Fixed Parameter Algorithms

Dániel Marx

Tel Aviv University, Israel

Open lectures for PhD students in computer science

December 12, 2009, Warsaw, Poland

Classical complexity

A brief review:

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

Can we say anything nontrivial about NP-hard problems?

Parameterized complexity

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

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

Parameterized complexity

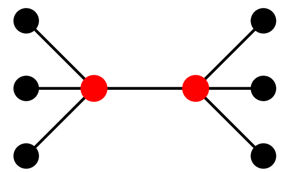
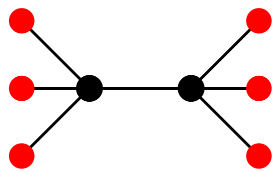
Main idea: Instead of expressing the running time as a function $T(n)$ of n, we express it as a function $T(n, k)$ of the input size n and some parameter k of the input.

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

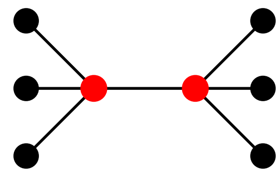
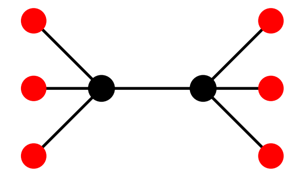
What can be the parameter k?

- The size k of the solution we are looking for.
- The maximum degree of the input graph.
- The diameter of the input graph.
- The length of clauses in the input Boolean formula.
- ...

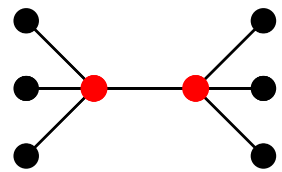
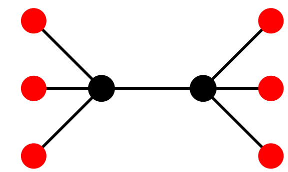
Parameterized complexity

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

Parameterized complexity

Problem:	MINIMUM VERTEX COVER	MAXIMUM INDEPENDENT SET
Input:	Graph G, integer k	Graph G, integer k
Question:	Is it possible to cover the edges with k vertices? 	Is it possible to find k independent vertices? 
Complexity:	NP-complete	NP-complete
Complete enumeration:	$O(n^k)$ possibilities	$O(n^k)$ possibilities

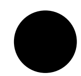
Parameterized complexity

Problem:	MINIMUM VERTEX COVER	MAXIMUM INDEPENDENT SET
Input:	Graph G, integer k	Graph G, integer k
Question:	Is it possible to cover the edges with k vertices? 	Is it possible to find k independent vertices? 
Complexity:	NP-complete	NP-complete
Complete enumeration:	$O(n^k)$ possibilities	$O(n^k)$ possibilities
	$O(2^k n^2)$ algorithm exists 	No $n^{o(k)}$ algorithm known 

Bounded search tree method

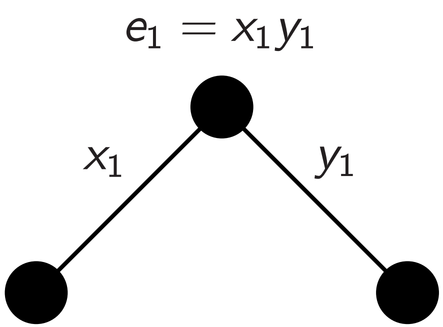
Algorithm for MINIMUM VERTEX COVER:

$e_1 = x_1 y_1$



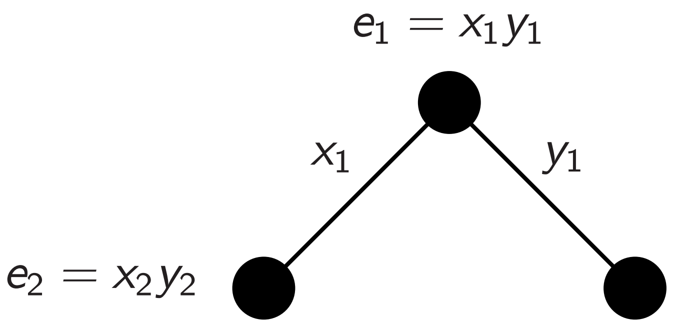
Bounded search tree method

Algorithm for MINIMUM VERTEX COVER:



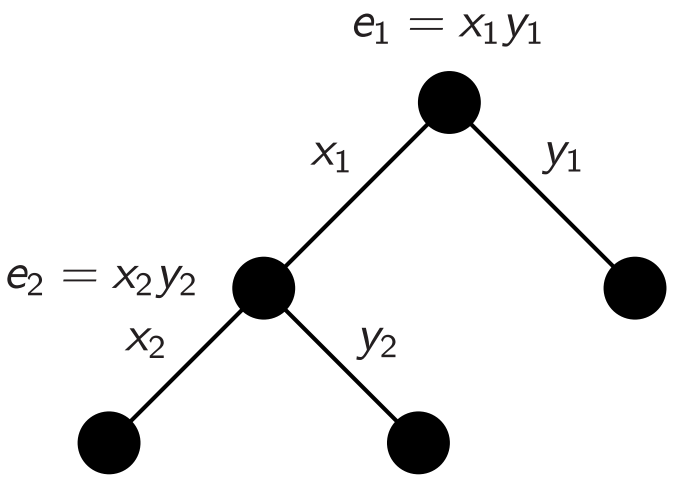
Bounded search tree method

Algorithm for MINIMUM VERTEX COVER:



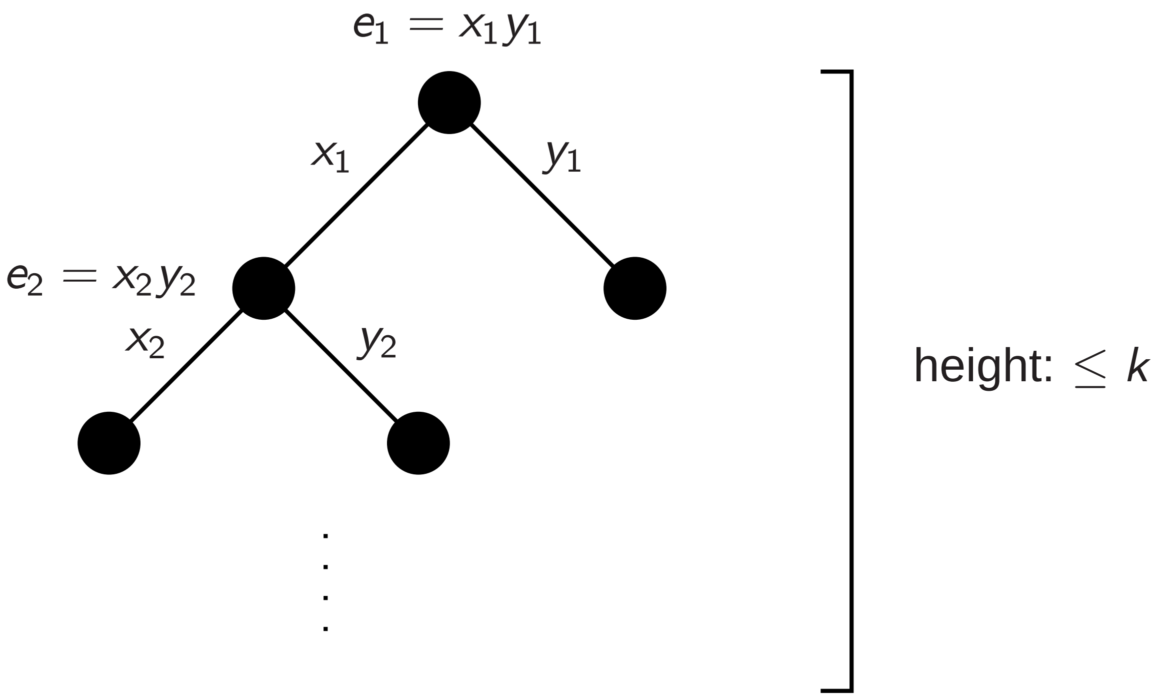
Bounded search tree method

Algorithm for MINIMUM VERTEX COVER:



Bounded search tree method

Algorithm for MINIMUM VERTEX COVER:



Height of the search tree is $\leq k \Rightarrow$ number of leaves is $\leq 2^k \Rightarrow$ complete search requires $2^k \cdot$ poly steps.

Fixed-parameter tractability

Definition: A **parameterization** of a decision problem is a function that assigns an integer parameter k to each input instance x.

The parameter can be

- explicit in the input (for example, if the parameter is the integer k appearing in the input (G, k) of VERTEX COVER), or
- implicit in the input (for example, if the parameter is the diameter d of the input graph G).

Main definition:

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

Fixed-parameter tractability

Definition: A **parameterization** of a decision problem is a function that assigns an integer parameter k to each input instance x.

Main definition:

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

Example: MINIMUM VERTEX COVER parameterized by the required size k is FPT: we have seen that it can be solved in time $O(2^k + n^2)$.

Better algorithms are known: e.g, $O(1.2832^k k + k|V|)$.

Main goal of parameterized complexity: to find FPT problems.

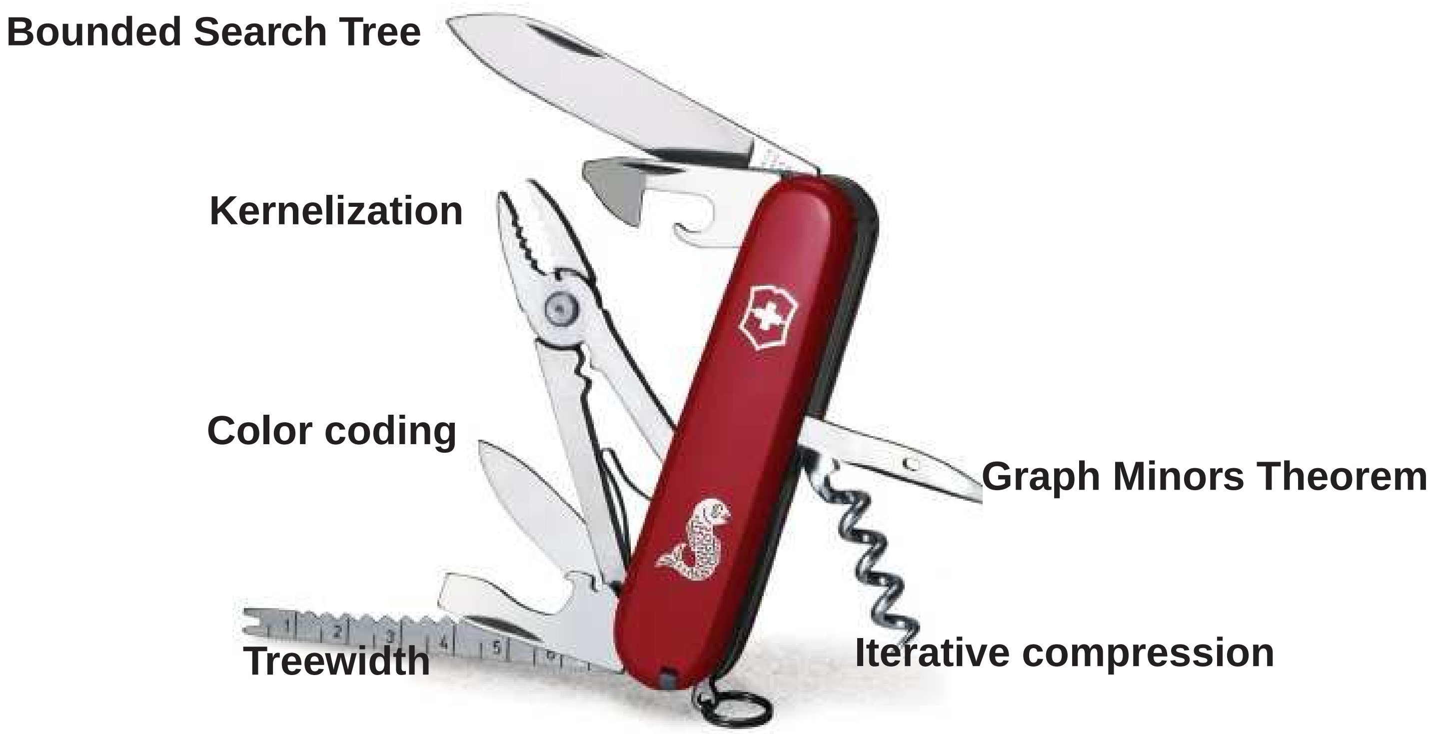
FPT problems

Examples of NP-hard problems that are FPT:

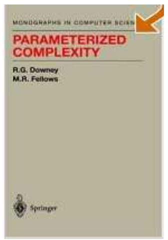
- Finding a vertex cover of size k.
- Finding a path of length k.
- Finding k disjoint triangles.
- Drawing the graph in the plane with k edge crossings.
- Finding disjoint paths that connect k pairs of points.
- …

FPT algorithmic techniques

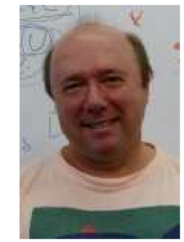
- Significant advances in the past 20 years or so (especially in recent years).
- Powerful toolbox for designing FPT algorithms:

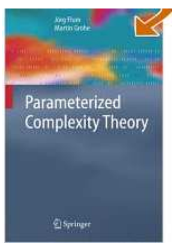


Books

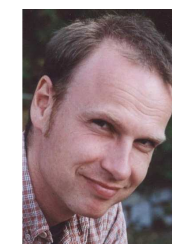


Downey-Fellows: Parameterized Complexity, Springer, 1999





Flum-Grohe: Parameterized Complexity Theory, Springer, 2006



Niedermeier: Invitation to Fixed-Parameter Algorithms, Oxford University Press, 2006.

Goals of the course

- Demonstrate techniques that were successfully used in the analysis of parameterized problems.
  - Determine quickly if a problem is FPT.
  - Design fast algorithms (improve the function $f(k)$).
- Introduce the basics of parameterized hardness theory (W[1]-hardness).

Notes

- **Warning:** The results presented for particular problems are not necessarily the best known results or the most useful approaches for these problems.
- Conventions:
  - Unless noted otherwise, k is the parameter.
  - O^* notation: $O^*(f(k))$ means $O(f(k) \cdot n^c)$ for some constant c.
  - Citations are mostly omitted (only for classical results).
  - We gloss over the difference between decision and search problems.

Kernelization

Kernelization

Definition: **Kernelization** is a polynomial-time transformation that maps an instance (I, k) to an instance (I', k') such that

- (I, k) is a yes-instance if and only if (I', k') is a yes-instance,
- $k' \leq k$, and
- $|I'| \leq f(k)$ for some function $f(k)$.

Kernelization

Definition: **Kernelization** is a polynomial-time transformation that maps an instance (I, k) to an instance (I', k') such that

- (I, k) is a yes-instance if and only if (I', k') is a yes-instance,
- $k' \leq k$, and
- $|I'| \leq f(k)$ for some function $f(k)$.

Simple fact: If a problem has a kernelization algorithm, then it is FPT.
Proof: Solve the instance (I', k') by brute force.

Kernelization

Definition: **Kernelization** is a polynomial-time transformation that maps an instance (I, k) to an instance (I', k') such that

- (I, k) is a yes-instance if and only if (I', k') is a yes-instance,
- $k' \leq k$, and
- $|I'| \leq f(k)$ for some function $f(k)$.

Simple fact: If a problem has a kernelization algorithm, then it is FPT.
Proof: Solve the instance (I', k') by brute force.

Converse: Every FPT problem has a kernelization algorithm.
Proof: Suppose there is an $f(k)n^c$ algorithm for the problem.

- If $f(k) \leq n$, then solve the instance in time $f(k)n^c \leq n^{c+1}$, and output a trivial yes- or no-instance.
- If $n < f(k)$, then we are done: a kernel of size $f(k)$ is obtained.

Kernelization for VERTEX COVER

General strategy: We devise a list of reduction rules, and show that if none of the rules can be applied and the size of the instance is still larger than $f(k)$, then the answer is trivial.

Reduction rules for VERTEX COVER instance (G, k):

Rule 1: If v is an isolated vertex $\Rightarrow (G \setminus v, k)$
Rule 2: If $d(v) > k \Rightarrow (G \setminus v, k-1)$

Kernelization for VERTEX COVER

General strategy: We devise a list of reduction rules, and show that if none of the rules can be applied and the size of the instance is still larger than $f(k)$, then the answer is trivial.

Reduction rules for VERTEX COVER instance (G, k):

Rule 1: If v is an isolated vertex $\Rightarrow (G \setminus v, k)$
Rule 2: If $d(v) > k \Rightarrow (G \setminus v, k-1)$

If neither Rule 1 nor Rule 2 can be applied:

- If $|V(G)| > k(k+1) \Rightarrow$ There is no solution (every vertex should be the neighbor of at least one vertex of the cover).
- Otherwise, $|V(G)| \leq k(k+1)$ and we have a $k(k+1)$ vertex kernel.

Kernelization for VERTEX COVER

Let us add a third rule:

Rule 1: If v is an isolated vertex $\Rightarrow (G \setminus v, k)$

Rule 2: If $d(v) > k \Rightarrow (G \setminus v, k-1)$

Rule 3: If $d(v) = 1$, then we can assume that its neighbor u is in the solution $\Rightarrow (G \setminus (u \cup v), k-1)$.

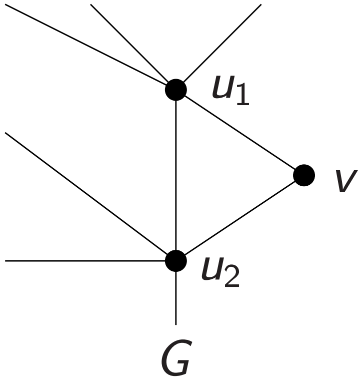
If none of the rules can be applied, then every vertex has degree at least 2.
$\Rightarrow |V(G)| \leq |E(G)|$

- If $|E(G)| > k^2 \Rightarrow$ There is no solution (each vertex of the solution can cover at most k edges).
- Otherwise, $|V(G)| \leq |E(G)| \leq k^2$ and we have a k^2 vertex kernel.

Kernelization for VERTEX COVER

Let us add a fourth rule:

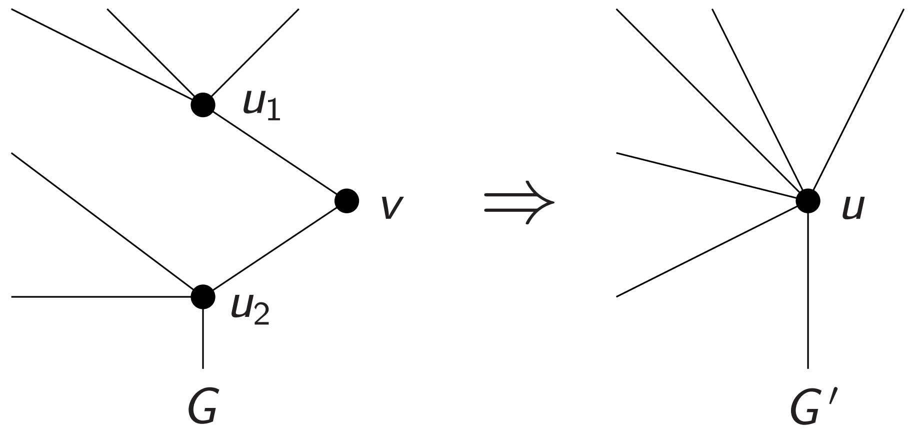
Rule 4a: If v has degree 2, and its neighbors u_1 and u_2 are adjacent, then we can assume that u_1, u_2 are in the solution $\Rightarrow (G \setminus \{u_1, u_2, v\}, k-2)$.



Kernelization for VERTEX COVER

Let us add a fourth rule:

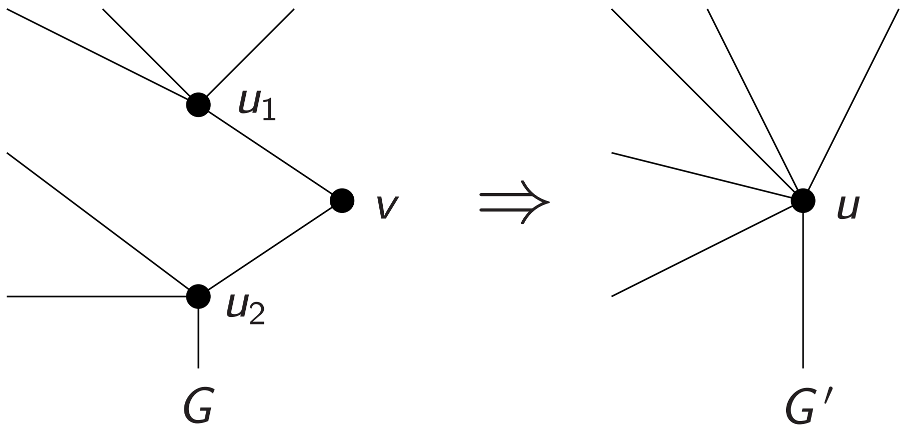
Rule 4b: If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting v $\Rightarrow (G', k-1)$.



Kernelization for Vertex Cover

Let us add a fourth rule:

Rule 4b: If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.



Correctness:

Let S' be a vertex cover of size $k-1$ for G'.

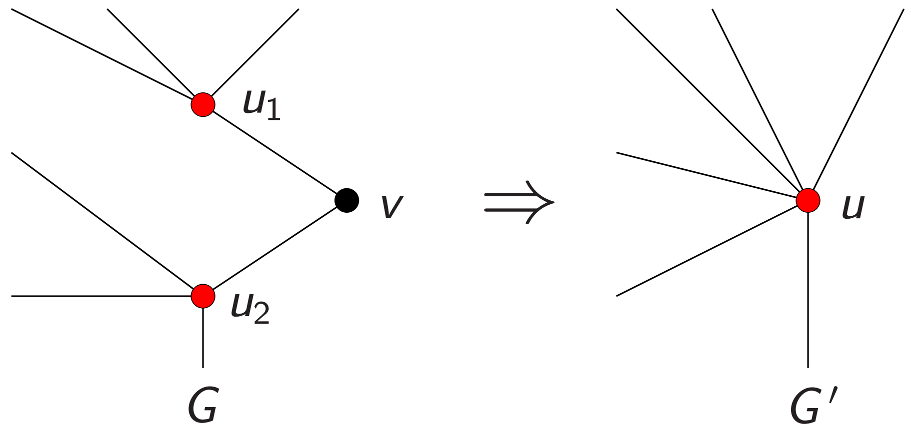
If $u \in S \Rightarrow (S' \setminus u) \cup \{u_1, u_2\}$ is a vertex cover of size k for G.

If $u \notin S \Rightarrow S' \cup v$ is a vertex cover of size k for G.

Kernelization for VERTEX COVER

Let us add a fourth rule:

> **Rule 4b:** If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.



Correctness:

Let S' be a vertex cover of size $k-1$ for G'.

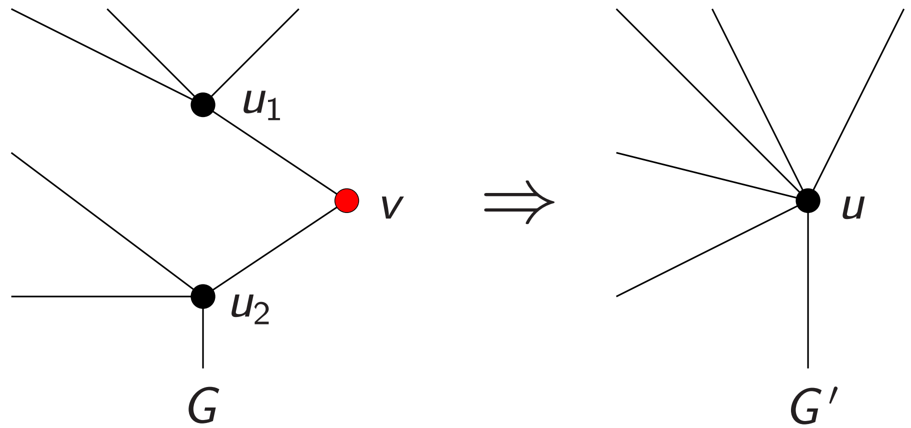
If $u \in S \Rightarrow (S' \setminus u) \cup \{u_1, u_2\}$ is a vertex cover of size k for G.

If $u \notin S \Rightarrow S' \cup v$ is a vertex cover of size k for G.

Kernelization for VERTEX COVER

Let us add a fourth rule:

Rule 4b: If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.



Correctness:

Let S' be a vertex cover of size $k-1$ for G'.

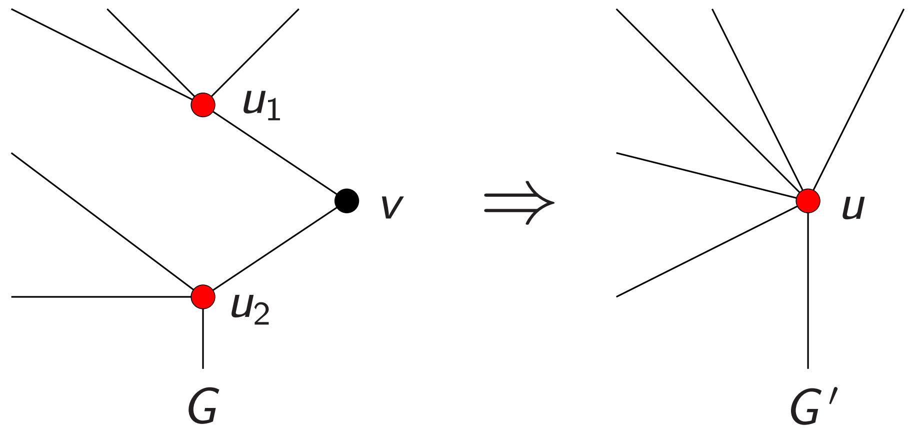
If $u \in S \Rightarrow (S' \setminus u) \cup \{u_1, u_2\}$ is a vertex cover of size k for G.

If $u \notin S \Rightarrow S' \cup v$ is a vertex cover of size k for G.

Kernelization for Vertex Cover

Let us add a fourth rule:

Rule 4b: If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.



Correctness:

Let S be a vertex cover of size k for G.

If $u_1, u_2 \in S \Rightarrow (S \setminus \{u_1, u_2, v\}) \cup u$ is a vertex cover of size $k-1$ for G'.

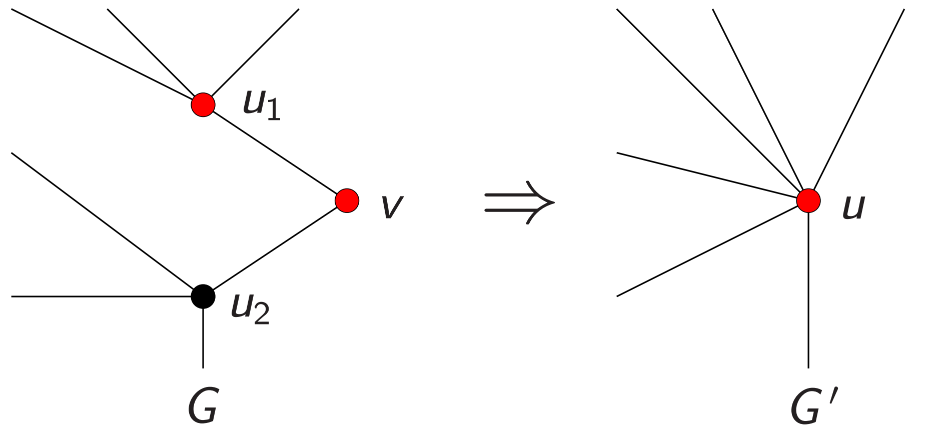
If exactly one of u_1 and u_2 is in S, then $v \in S \Rightarrow (S \setminus \{u_1, u_2, v\}) \cup u$ is a vertex cover of size $k-1$ for G'.

If $u_1, u_2 \notin S$, then $v \in S \Rightarrow (S \setminus v)$ is a vertex cover of size $k-1$ for G'.

Kernelization for **VERTEX COVER**

Let us add a fourth rule:

> **Rule 4b:** If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.



Correctness:

Let S be a vertex cover of size k for G.

If $u_1, u_2 \in S \Rightarrow (S \setminus \{u_1, u_2, v\}) \cup u$ is a vertex cover of size $k-1$ for G'.

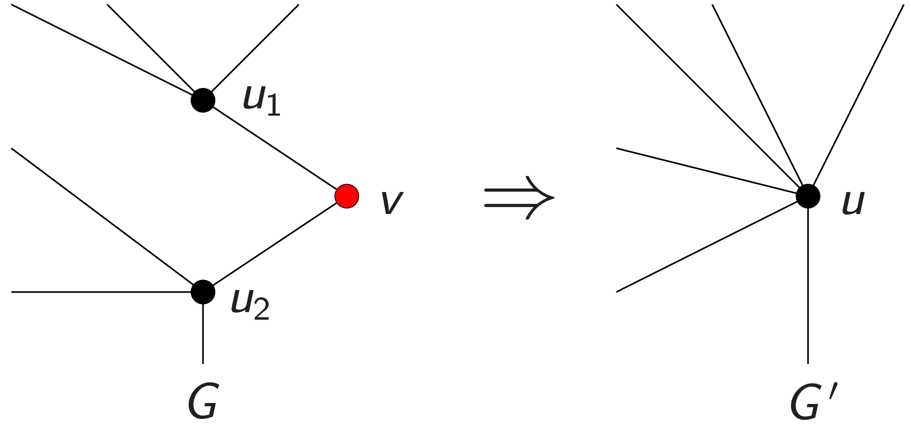
If exactly one of u_1 and u_2 is in S, then $v \in S \Rightarrow (S \setminus \{u_1, u_2, v\}) \cup u$ is a vertex cover of size $k-1$ for G'.

If $u_1, u_2 \notin S$, then $v \in S \Rightarrow (S \setminus v)$ is a vertex cover of size $k-1$ for G'.

Kernelization for VERTEX COVER

Let us add a fourth rule:

Rule 4b: If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.



Correctness:

Let S be a vertex cover of size k for G.

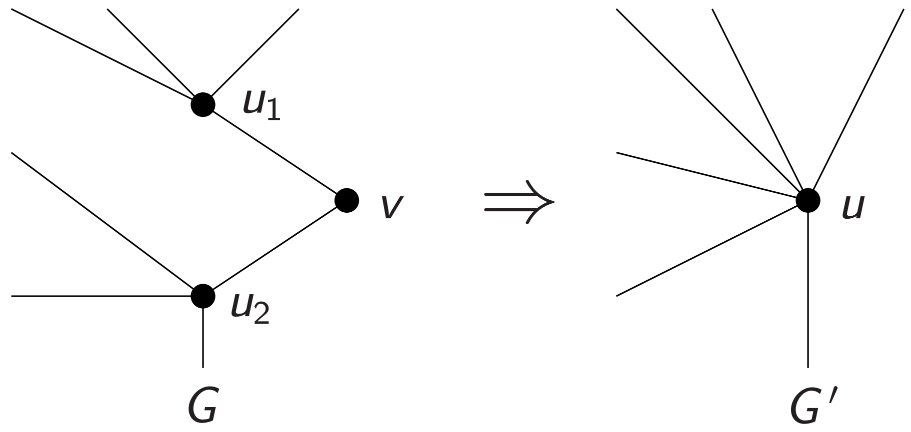
If $u_1, u_2 \in S \Rightarrow (S \setminus \{u_1, u_2, v\}) \cup u$ is a vertex cover of size $k-1$ for G'.
If exactly one of u_1 and u_2 is in S, then $v \in S \Rightarrow (S \setminus \{u_1, u_2, v\}) \cup u$ is a vertex cover of size $k-1$ for G'.

If $u_1, u_2 \notin S$, then $v \in S \Rightarrow (S \setminus v)$ is a vertex cover of size $k-1$ for G'.

Kernelization for Vertex Cover

Let us add a fourth rule:

> **Rule 4b:** If v has degree 2, then G' is obtained by identifying the two neighbors of v and deleting $v \Rightarrow (G', k-1)$.

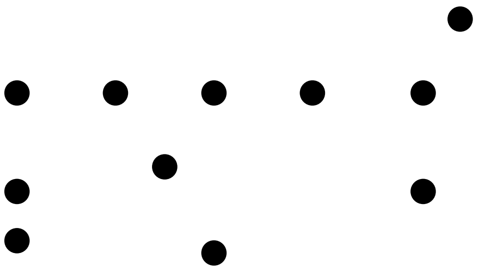


Kernel size:

- If $|E(G)| > k^2 \Rightarrow$ There is no solution (each vertex of the solution can cover at most k edges).
- Otherwise, $|V(G)| \leq 2|E(G)|/3 \leq \frac{2}{3}k^2$ and we have a $\frac{2}{3}k^2$ vertex kernel.

COVERING POINTS WITH LINES

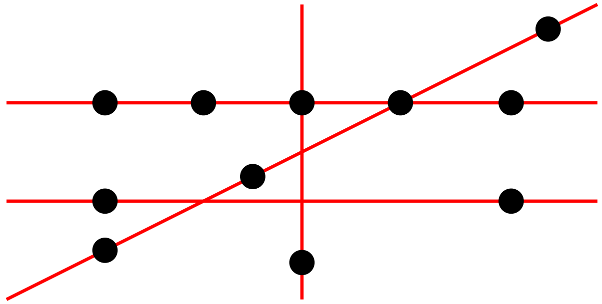
Task: Given a set P of n points in the plane and an integer k, find k lines that cover all the points.



Note: We can assume that every line of the solution covers at least 2 points, thus there are at most n^2 candidate lines.

COVERING POINTS WITH LINES

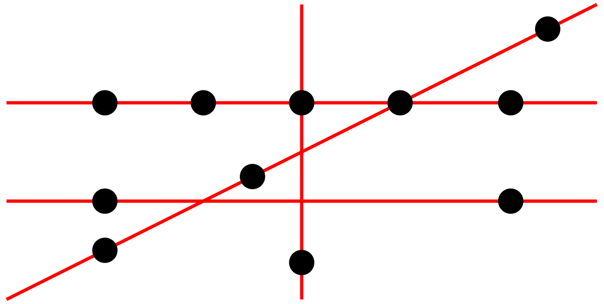
Task: Given a set P of n points in the plane and an integer k, find k lines that cover all the points.



Note: We can assume that every line of the solution covers at least 2 points, thus there are at most n^2 candidate lines.

Covering Points with Lines

Task: Given a set P of n points in the plane and an integer k, find k lines that cover all the points.



Note: We can assume that every line of the solution covers at least 2 points, thus there are at most n^2 candidate lines.

Reduction Rule:

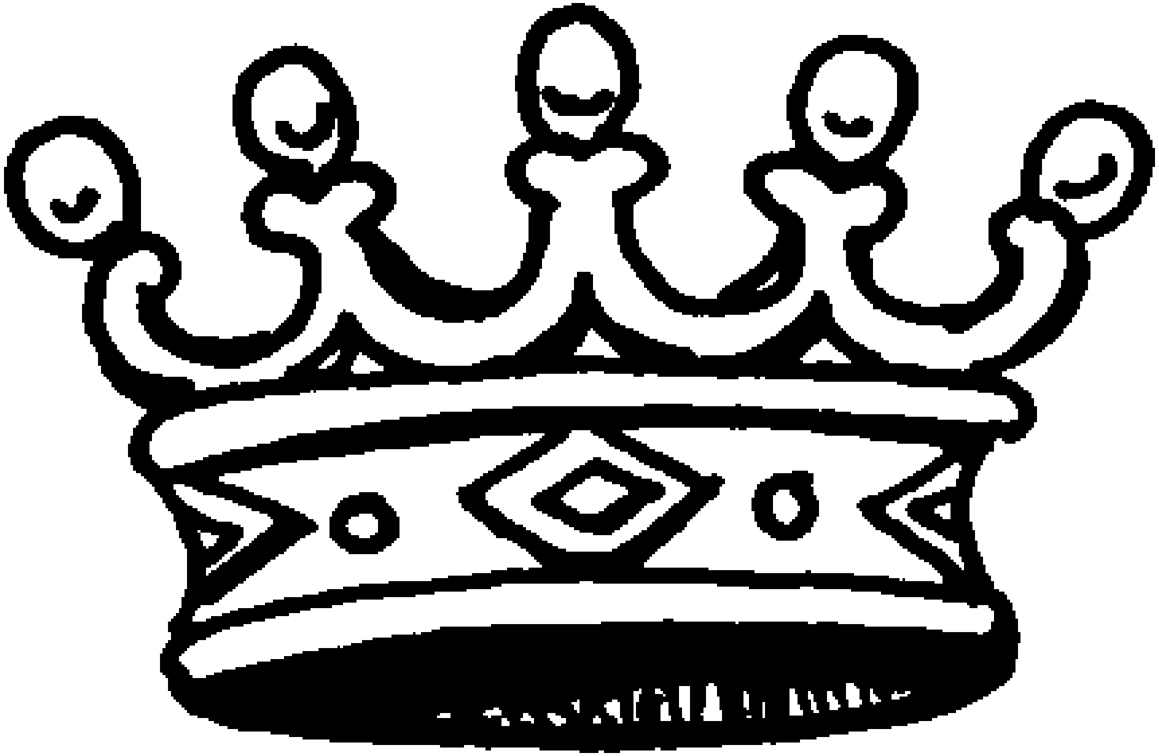
If a candidate line covers a set S of more than k points $\Rightarrow$ $(P \setminus S, k-1)$.

If this rule cannot be applied and there are still more than k^2 points, then there is no solution $\Rightarrow$ Kernel with at most k^2 points.

Kernelization

- Kernelization can be thought of as a polynomial-time preprocessing before attacking the problem with whatever method we have. “It does no harm” to try kernelization.
- Some kernelizations use lots of simple reduction rules and require a complicated analysis to bound the kernel size…
- … while other kernelizations are based on surprising nice tricks (Next: Crown Reduction and the Sunflower Lemma).
- Possibility to prove lower bounds.

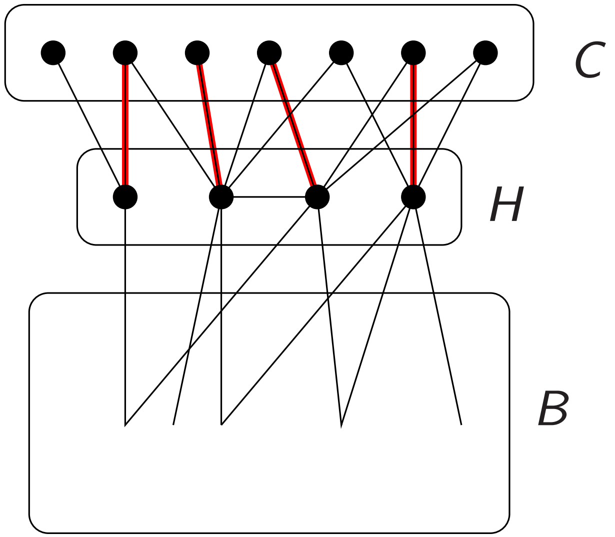
Crown Reduction



Crown Reduction

Definition: A **crown decomposition** is a partition $C \cup H \cup B$ of the vertices such that

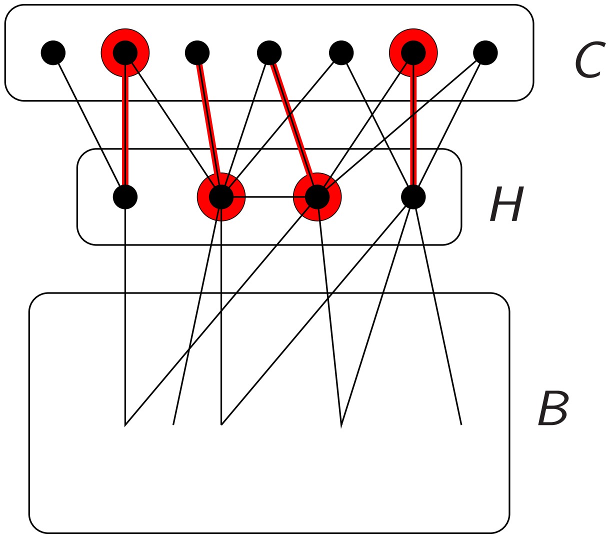
- C is an independent set,
- there is no edge between C and B,
- there is a matching between C and H that covers H.



Crown Reduction

Definition: A **crown decomposition** is a partition $C \cup H \cup B$ of the vertices such that

- C is an independent set,
- there is no edge between C and B,
- there is a matching between C and H that covers H.



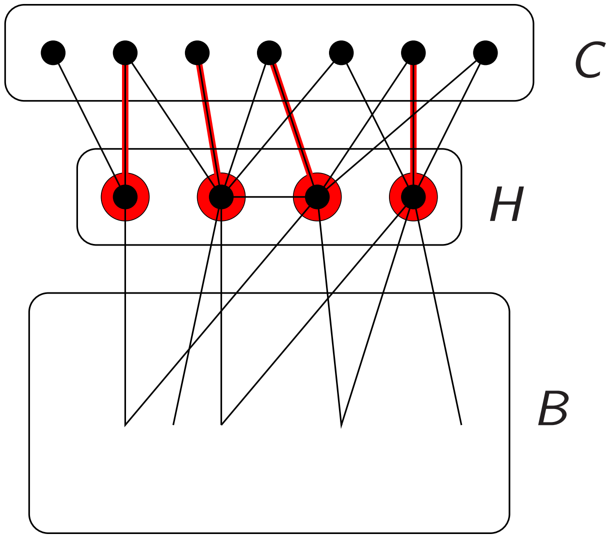
Crown rule for VERTEX COVER:

The matching needs to be covered and we can assume that it is covered by H (makes no sense to use vertices of C)

$\Rightarrow (G \setminus (H \cup C), k - |H|)$.

Crown Reduction

Definition: A **crown decomposition** is a partition $C \cup H \cup B$ of the vertices such that



- C is an independent set,
- there is no edge between C and B,
- there is a matching between C and H that covers H.

Crown rule for VERTEX COVER:

The matching needs to be covered and we can assume that it is covered by H (makes no sense to use vertices of C)

$\Rightarrow (G \setminus (H \cup C), k - |H|)$.

Crown Reduction

Key lemma:

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Crown Reduction

Key lemma:

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$, $\Rightarrow$ No solution!
- find a crown decomposition, $\Rightarrow$ Reduce!
- or conclude that the graph has at most $3k$ vertices.

$\Rightarrow 3k$ vertex kernel!

This gives a 3k vertex kernel for VERTEX COVER.

Proof

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

For the proof, we need the classical Kőnig's Theorem.

$\tau(G)$: size of the minimum vertex cover
$\nu(G)$: size of the maximum matching (independent set of edges)

Theorem: [Kőnig, 1931] If G is **bipartite,** then

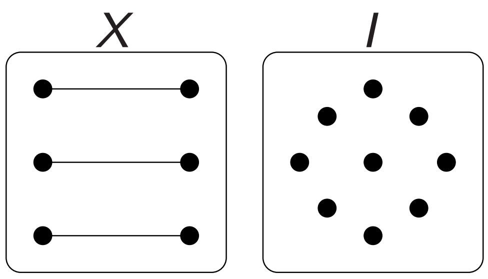
$$\tau(G) = \nu(G)$$

Proof

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Proof: Find (greedily) a maximal matching; if its size is at least $k+1$, then we are done. The rest of the graph is an independent set I.

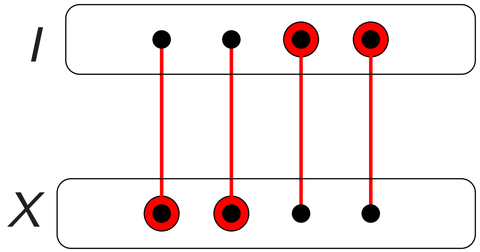


Proof

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Proof: Find (greedily) a maximal matching; if its size is at least $k+1$, then we are done. The rest of the graph is an independent set I.



Find a maximum matching/minimum vertex cover in the bipartite graph between X and I.

Proof

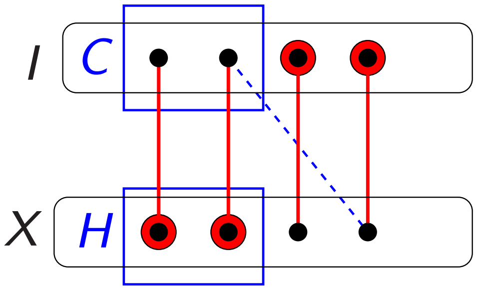
Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k + 1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Proof:

Case 1: The minimum vertex cover contains at least one vertex of X

$\Rightarrow$ There is a crown decomposition.



Proof

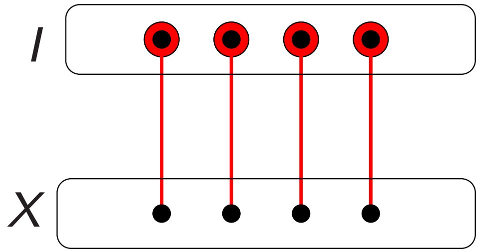
Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Proof:

Case 1: The minimum vertex cover contains at least one vertex of X

$\Rightarrow$ There is a crown decomposition.



Case 2: The minimum vertex cover contains only vertices of I $\Rightarrow$ It contains every vertex of I

$\Rightarrow$ There are at most $2k + k$ vertices.

DUAL OF VERTEX COLORING

Parameteric dual of k-COLORING. Also known as SAVING k COLORS.

Task: Given a graph G and an integer k, find a vertex coloring with $|V(G)| - k$ colors.

Crown rule for DUAL OF VERTEX COLORING:

DUAL OF VERTEX COLORING

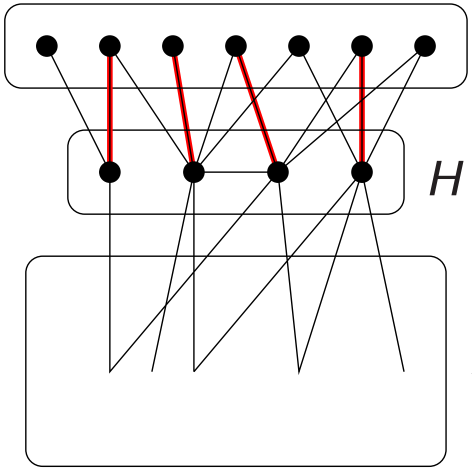
Parameteric dual of k-COLORING. Also known as SAVING k COLORS.

Task: Given a graph G and an integer k, find a vertex coloring with $|V(G)| - k$ colors.

Crown rule for DUAL OF VERTEX COLORING:

Suppose there is a crown decomposition for the **complement graph** $\overline{G}$.

- C is a clique in G: each vertex needs a distinct color.
- Because of the matching, it is possible to color H using only these $|C|$ colors.
- These colors cannot be used for B.
- $(G \setminus (H \cup C), k - |H|)$



DUAL OF VERTEX COLORING

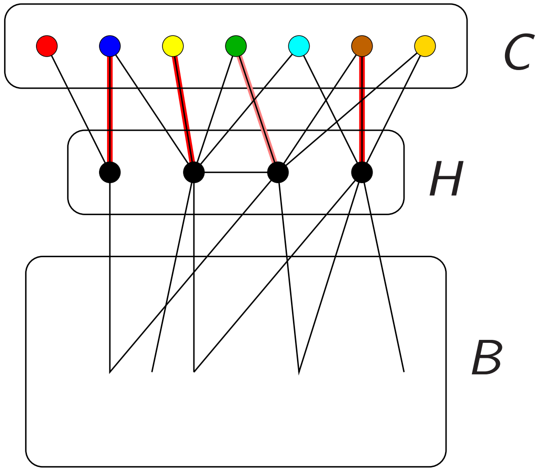
Parameteric dual of k-COLORING. Also known as SAVING k COLORS.

Task: Given a graph G and an integer k, find a vertex coloring with $|V(G)| - k$ colors.

Crown rule for DUAL OF VERTEX COLORING:

Suppose there is a crown decomposition for the **complement graph** $\overline{G}$.

- C is a clique in G: each vertex needs a distinct color.
- Because of the matching, it is possible to color H using only these $|C|$ colors.
- These colors cannot be used for B.
- $(G \setminus (H \cup C), k - |H|)$



DUAL OF VERTEX COLORING

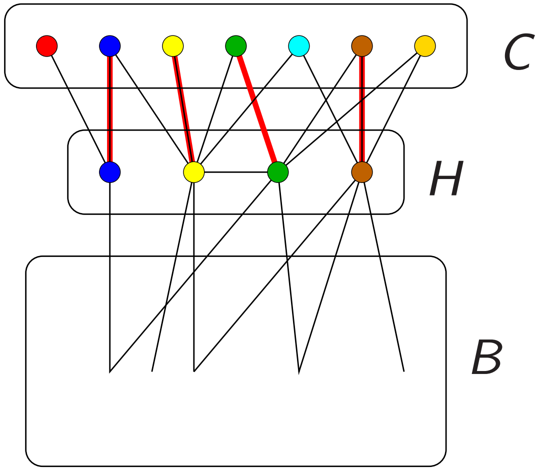
Parameteric dual of k-COLORING. Also known as SAVING k COLORS.

Task: Given a graph G and an integer k, find a vertex coloring with $|V(G)| - k$ colors.

Crown rule for DUAL OF VERTEX COLORING:

Suppose there is a crown decomposition for the **complement graph** $\overline{G}$.

- C is a clique in G: each vertex needs a distinct color.
- Because of the matching, it is possible to color H using only these $|C|$ colors.
- These colors cannot be used for B.
- $(G \setminus (H \cup C), k - |H|)$



Crown Reduction for DUAL OF VERTEX COLORING

Use the key lemma for the complement $\overline{G}$ of G:

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$, $\Rightarrow$ YES: we can save k colors!
- find a crown decomposition, $\Rightarrow$ Reduce!
- or conclude that the graph has at most $3k$ vertices.

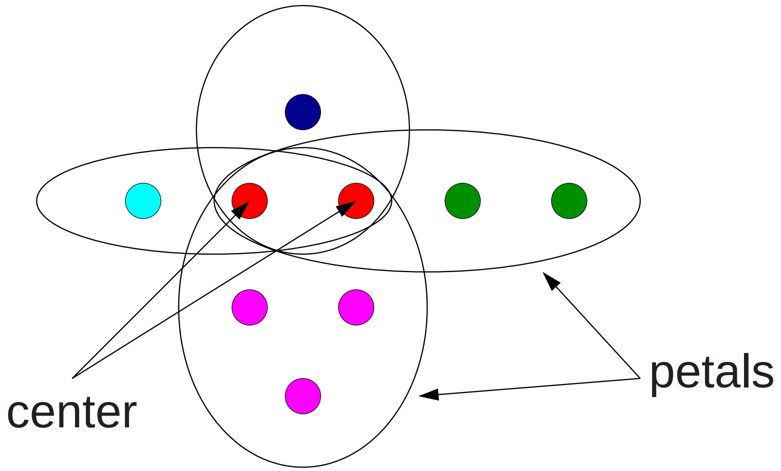
  $\Rightarrow$ $3k$ vertex kernel!

This gives a 3k vertex kernel for DUAL OF VERTEX COLORING.

Sunflower Lemma

Sunflower lemma

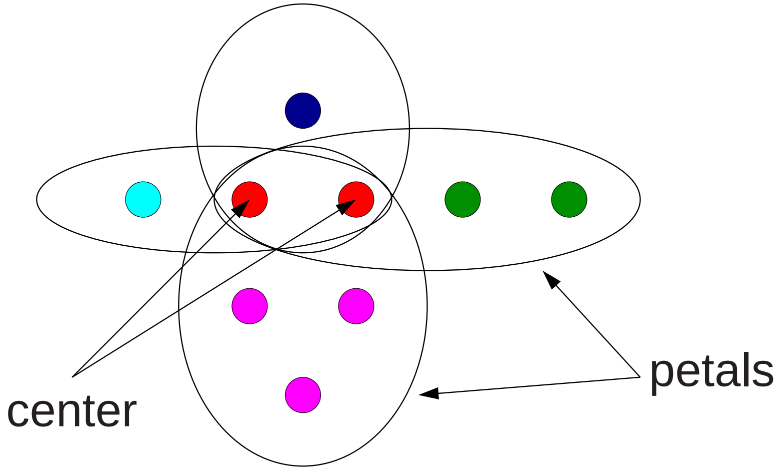
Definition: Sets S_1, S_2, ... , S_k form a **sunflower** if the sets $S_i \setminus (S_1 \cap S_2 \cap \cdots \cap S_k)$ are disjoint.



Lemma: [Erdős and Rado, 1960] If the size of a set system is greater than $(p-1)^d \cdot d!$ and it contains only sets of size at most d, then the system contains a sunflower with p petals. Furthermore, in this case such a sunflower can be found in polynomial time.

Sunflowers and d-HITTING SET

d-HITTING SET: Given a collection $\mathcal{S}$ of sets of size at most d and an integer k, find a set S of k elements that intersects every set of $\mathcal{S}$.



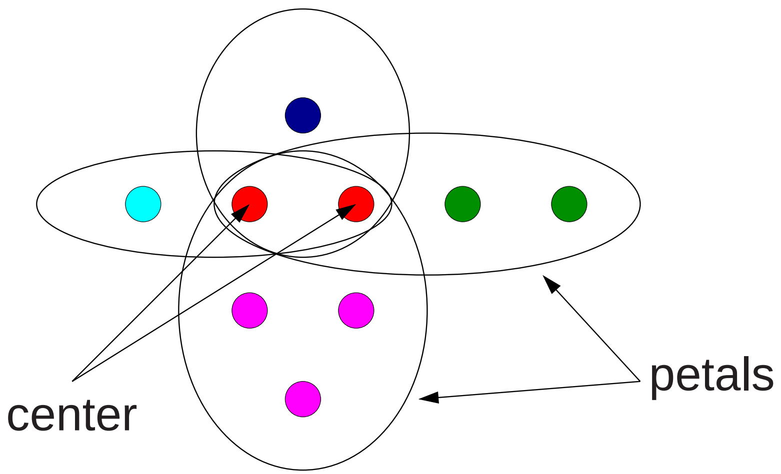
Reduction Rule: If $k+1$ sets form a sunflower, then remove these sets from $\mathcal{S}$ and add the center C to $\mathcal{S}$ (S does not hit one of the petals, thus it has to hit the center).

Note: if the center is empty (the sets are disjoint), then there is no solution.

If the rule cannot be applied, then there are at most $O(k^d)$ sets.

Sunflowers and d-HITTING SET

d-HITTING SET: Given a collection $\mathcal{S}$ of sets of size at most d and an integer k, find a set S of k elements that intersects every set of $\mathcal{S}$.



Reduction Rule (variant): Suppose more than $k+1$ sets form a sunflower.

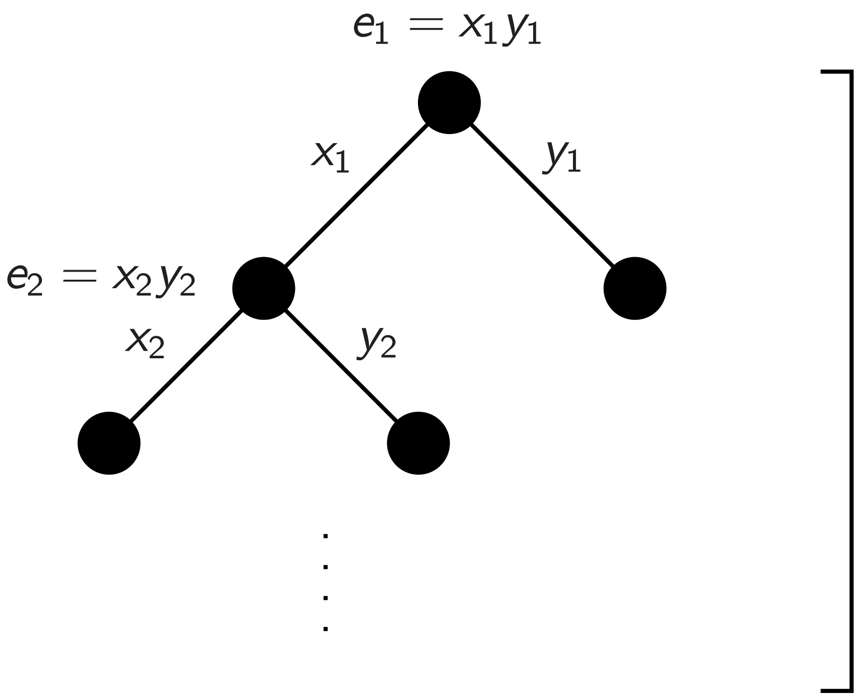
- If the sets are disjoint $\Rightarrow$ No solution.
- Otherwise, keep only $k+1$ of the sets.

If the rule cannot be applied, then there are at most $O(k^d)$ sets.

Branching and bounded search trees

Bounded search tree method

Recall how we solved MINIMUM VERTEX COVER:



height: $\leq k$

Bounded search tree method

We solve the problem by one or more **branching rules.**

Each rule makes a “guess” in such a way that at least one guess will lead to a correct solution.

If we have branching rules such that

- each rule branches into at most $b(k)$ directions, and
- applying a rule decreases the parameter,

then the problem can be solved in time $O^*(b(k)^k)$.

In many cases, this crude upper bound can be improved by better analysis.

VERTEX COVER

Improved algorithm for VERTEX COVER.

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

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

VERTEX COVER

Improved algorithm for VERTEX COVER.

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

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

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

Better analysis

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

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

There is a standard technique for bounding such functions asymptotically.

Better analysis

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

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

There is a standard technique for bounding such functions asymptotically.

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

What values of c are good? We need:

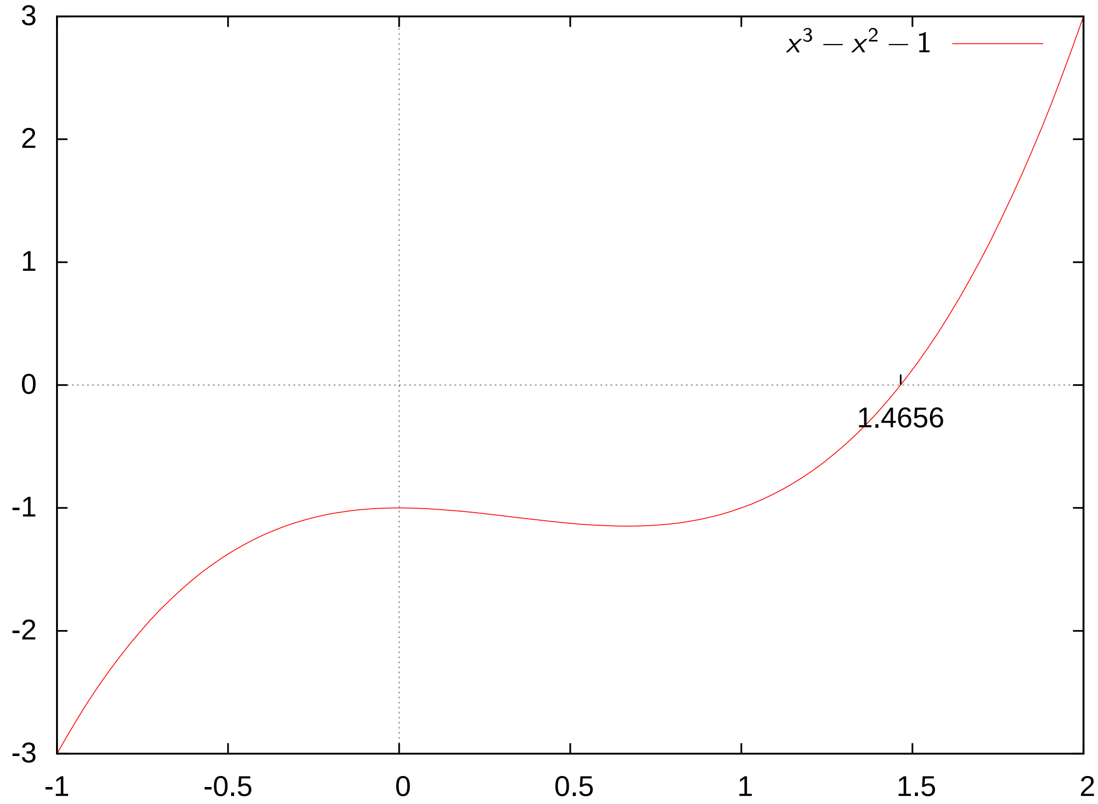
$$c^k \geq c^{k-1} + c^{k-3}$$

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

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

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

Better analysis



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

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

Better analysis

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

Is this bound tight? There are two questions:

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

Branching vectors

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

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

Branching vectors

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

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

The value $c > 1$ has to satisfy:

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

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

It is hard to compare branching vectors intuitively.

Branching vectors

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

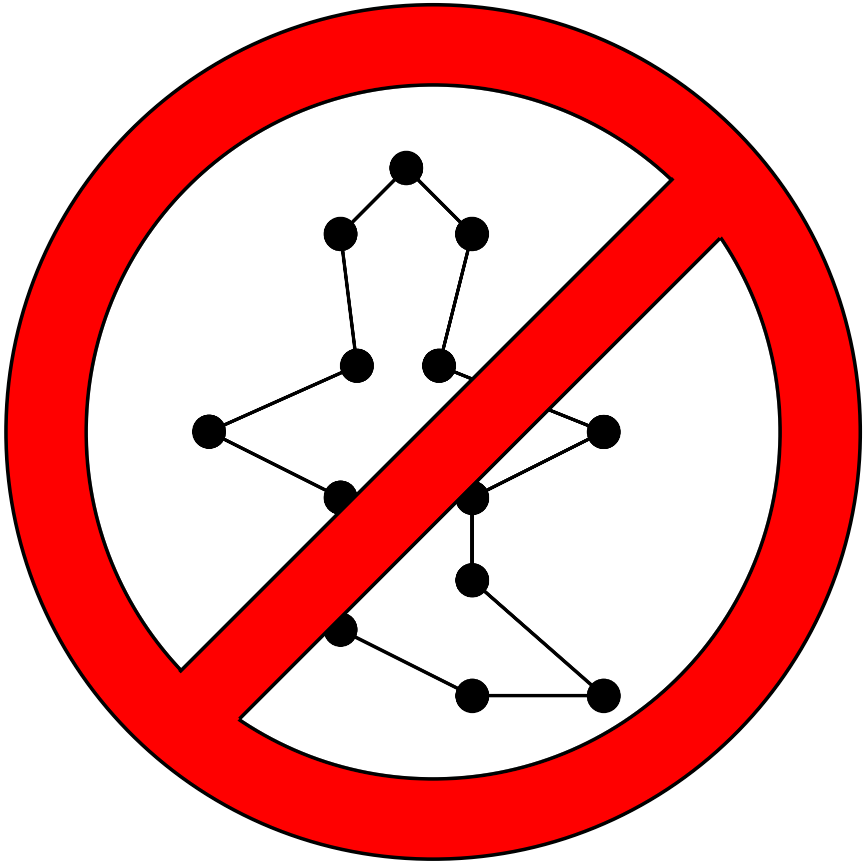
$$t(k) \leq t(k-i) + t(k-j) \Rightarrow c^k \geq c^{k-i} + c^{k-j}$$

$$c^j - c^{j-i} - 1 \geq 0$$

We compute the unique positive root.

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

Forbidden subgraphs



Forbidden subgraphs

General problem class: Given a graph G and an integer k, transform G with at most k modifications (add/remove vertices/edges) into a graph having property $\mathcal{P}$.

Example:

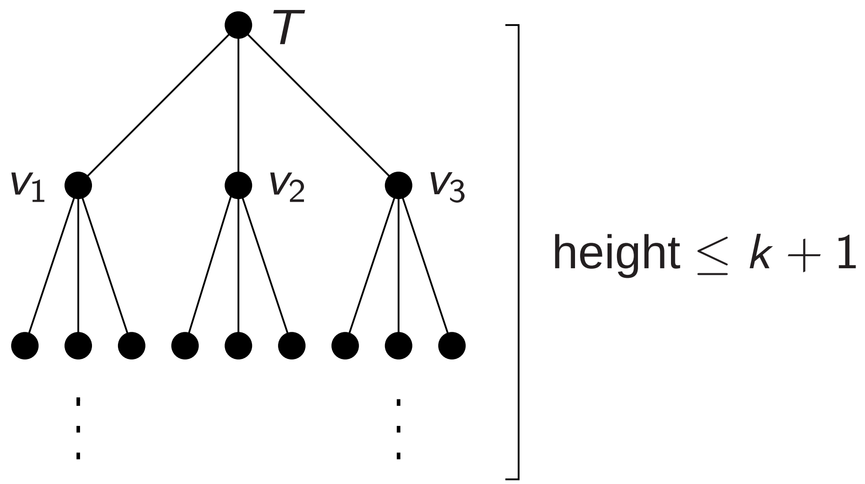
TRIANGLE DELETION: make the graph triangle-free by deleting at most k vertices.

Branching algorithm:

- If the graph is triangle-free, then we are done.
- **Branching rule:** If there is a triangle $v_1 v_2 v_3$, then at least one of v_1, v_2, v_3 has to be deleted $\Rightarrow$ We branch into 3 directions.

TRIANGLE DELETION

Search tree:



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

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

Hereditary properties

Definition: A graph property $\mathcal{P}$ is **hereditary** if for every $G \in \mathcal{P}$ and induced subgraph G' of G, we have $G' \in \mathcal{P}$ as well.

Examples: triangle-free, bipartite, interval graph, planar

Observation: Every hereditary property $\mathcal{P}$ can be characterized by a (finite or infinite) set $\mathcal{F}$ of forbidden induced subgraphs:

$$G \in \mathcal{P} \iff \forall H \in \mathcal{F},\, H \not\subseteq_{\text{ind}} G$$

Hereditary properties

Definition: A graph property $\mathcal{P}$ is **hereditary** if for every $G \in \mathcal{P}$ and induced subgraph G' of G, we have $G' \in \mathcal{P}$ as well.

Examples: triangle-free, bipartite, interval graph, planar

Observation: Every hereditary property $\mathcal{P}$ can be characterized by a (finite or infinite) set $\mathcal{F}$ of forbidden induced subgraphs:

$$G \in \mathcal{P} \Longleftrightarrow \forall H \in \mathcal{F}, H \not\subseteq_{\text{ind}} G$$

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

Hereditary properties

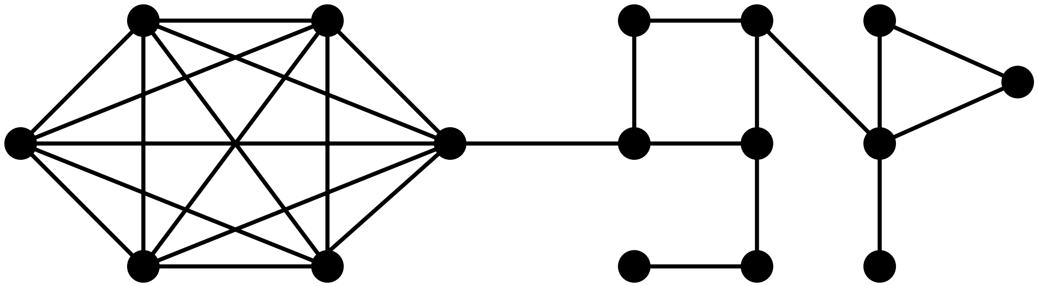
Theorem: If $\mathcal{P}$ is hereditary and can be characterized by a **finite** set $\mathcal{F}$ of forbidden induced subgraphs, then the graph modification problems corresponding to $\mathcal{P}$ are FPT.

Proof:

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

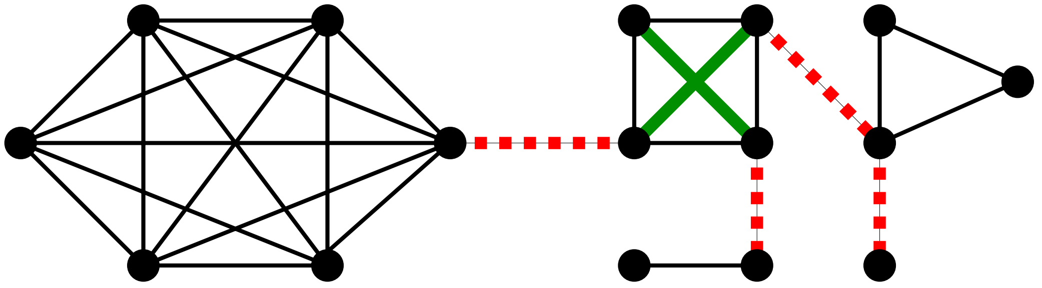
CLUSTER EDITING

Task: Given a graph G and an integer k, add/remove at most k edges such that every component is a clique in the resulting graph.



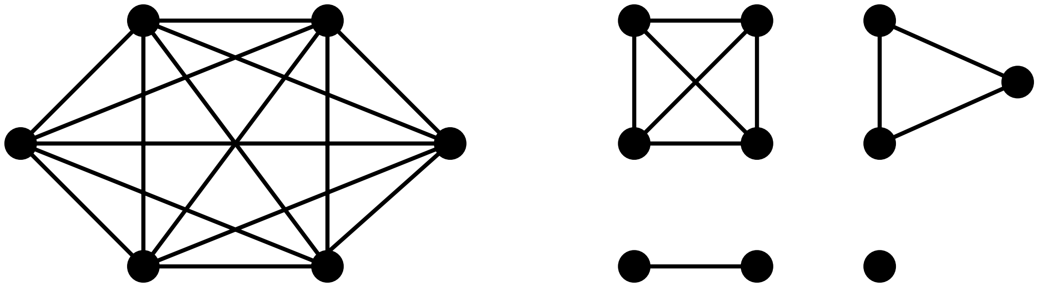
CLUSTER EDITING

Task: Given a graph G and an integer k, add/remove at most k edges such that every component is a clique in the resulting graph.



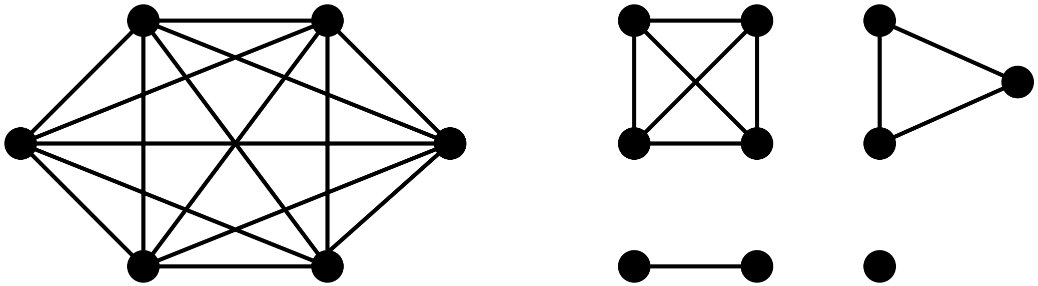
CLUSTER EDITING

Task: Given a graph G and an integer k, add/remove at most k edges such that every component is a clique in the resulting graph.



CLUSTER EDITING

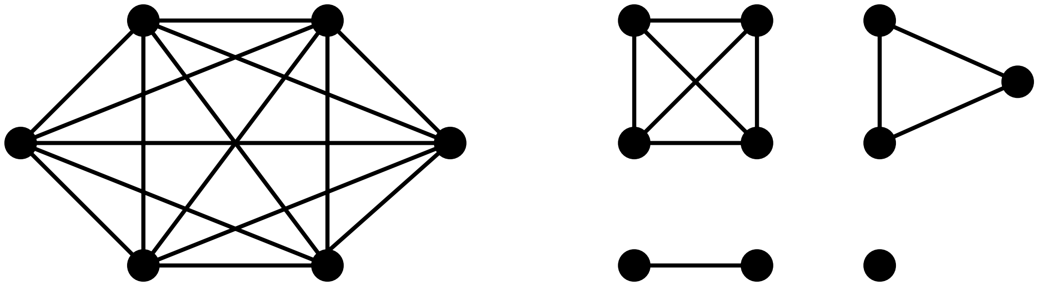
Task: Given a graph G and an integer k, add/remove at most k edges such that every component is a clique in the resulting graph.



Property $\mathcal{P}$: every component is a clique.

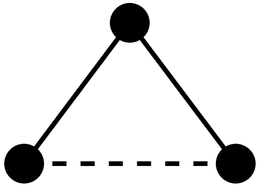
CLUSTER EDITING

Task: Given a graph G and an integer k, add/remove at most k edges such that every component is a clique in the resulting graph.



Property $\mathcal{P}$: every component is a clique.

Forbidden induced subgraph:



$O^*(3^k)$ time algorithm.

CHORDAL COMPLETION

Definition: A graph is **chordal** if it does not contain an induced cycle of length greater than 3.

CHORDAL COMPLETION: Given a graph G and an integer k, add at most k edges to G to make it a chordal graph.

CHORDAL COMPLETION

Definition: A graph is **chordal** if it does not contain an induced cycle of length greater than 3.

CHORDAL COMPLETION: Given a graph G and an integer k, add at most k edges to G to make it a chordal graph.

The forbidden induced subgraphs are the cycles of length greater 3
$\Rightarrow$ Not a finite set!

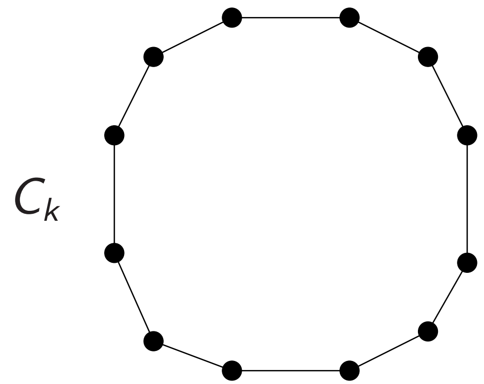
CHORDAL COMPLETION

Definition: A graph is **chordal** if it does not contain an induced cycle of length greater than 3.

CHORDAL COMPLETION: Given a graph G and an integer k, add at most k edges to G to make it a chordal graph.

The forbidden induced subgraphs are the cycles of length greater 3
$\Rightarrow$ Not a finite set!

Lemma: At least $k-3$ edges are needed to make a k-cycle chordal.
Proof: By induction. $k=3$ is trivial.



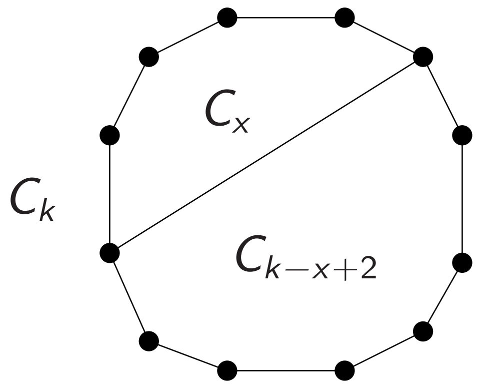
CHORDAL COMPLETION

Definition: A graph is **chordal** if it does not contain an induced cycle of length greater than 3.

CHORDAL COMPLETION: Given a graph G and an integer k, add at most k edges to G to make it a chordal graph.

The forbidden induced subgraphs are the cycles of length greater 3
$\Rightarrow$ Not a finite set!

Lemma: At least $k-3$ edges are needed to make a k-cycle chordal.
Proof: By induction. $k=3$ is trivial.



C_x: $x-3$ edges
C_{k-x+2}: $k-x-1$ edges
C_k: $(x-3)+(k-x-1)+1=k-3$ edges

Chordal Completion

Algorithm:

- Find an induced cycle C of length at least 4 (can be done in polynomial time).
- If no such cycle exists $\Rightarrow$ Done!
- If C has more than $k+3$ vertices $\Rightarrow$ No solution!
- Otherwise, one of the

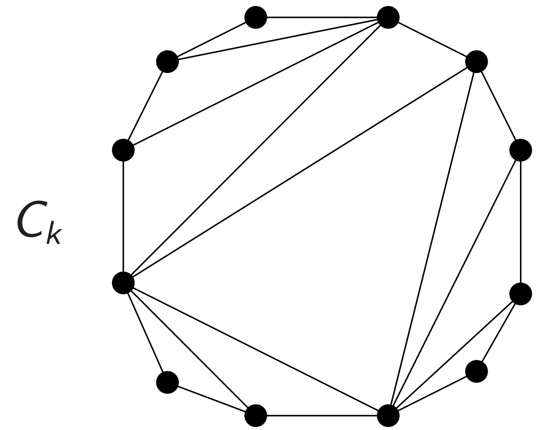
$$\binom{|C|}{2} - |C| \leq (k+3)(k+2)/2 - k = O(k^2)$$

  missing edges has to be added $\Rightarrow$ Branch!

Size of the search tree is $k^{O(k)}$.

Chordal Completion – *more efficiently*

Definition: Triangulation of a cycle.



Lemma: Every chordal supergraph of a cycle C contains a triangulation of the cycle C.

Lemma: The number of ways a cycle of length k can be triangulated is exactly the $(k-2)$-nd Catalan number

$$C_{k-2} = \frac{1}{k-1}\binom{2(k-2)}{k-2} \leq 4^{k-3}.$$

CHORDAL COMPLETION – *more efficiently*

Algorithm:

- Find an induced cycle C of length at least 4 (can be done in polynomial time).
- If no such cycle exists $\Rightarrow$ Done!
- If C has more than $k + 3$ vertices $\Rightarrow$ No solution!
- Otherwise, one of the $\leq 4^{|C|-3}$ triangulations has to be in the solution $\Rightarrow$ Branch!

Claim: Search tree has at most $T_k = 4^k$ leaves.

Proof: By induction. Number of leaves is at most

$$T_k \leq 4^{|C|-3} \cdot T_{k-(|C|-3)} \leq 4^{|C|-3} \cdot 4^{k-(|C|-3)} = 4^k.$$

Iterative compression

Iterative compression

- A surprising small, but very powerful trick.
- Most useful for deletion problems: delete k things to achieve some property.
- Demonstration: ODD CYCLE TRANSVERSAL aka BIPARTITE DELETION aka GRAPH BIPARTIZATION: Given a graph G and an integer k, delete k vertices to make the graph bipartite.
- Forbidden induced subgraphs: odd cycles. There is no bound on the size of odd cycles.

BIPARTITE DELETION

Solution based on iterative compression:

- **Step 1:**

  Solve the **annotated problem** for bipartite graphs:

  > Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.

BIPARTITE DELETION

Solution based on iterative compression:

- **Step 1:**

  Solve the **annotated problem** for bipartite graphs:

  Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.

- **Step 2:**

  Solve the **compression problem** for general graphs:

  Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.

BIPARTITE DELETION

Solution based on iterative compression:

- **Step 1:**

  Solve the **annotated problem** for bipartite graphs:

  > Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.

- **Step 2:**

  Solve the **compression problem** for general graphs:

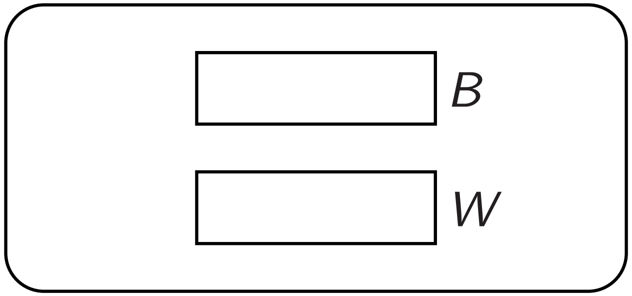
  > Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.

- **Step 3:**

  Apply the magic of iterative compression...

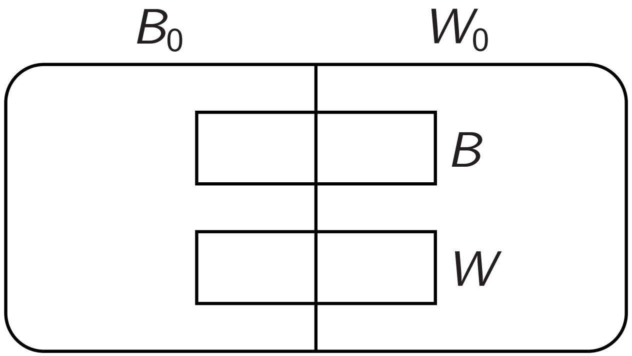
Step 1: The annotated problem

Given a bipartite graph G, two sets B, $W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.



Step 1: The annotated problem

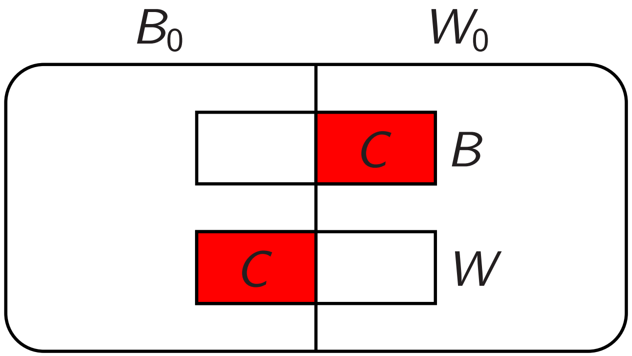
Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.



Find an arbitrary 2-coloring (B_0, W_0) of G.

Step 1: The annotated problem

Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.

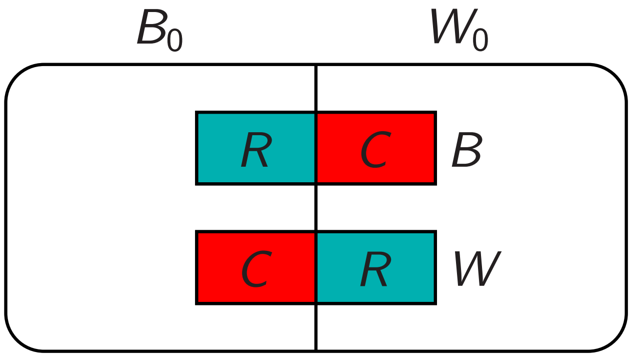


Find an arbitrary 2-coloring (B_0, W_0) of G.

$C := (B_0 \cap W) \cup (W_0 \cap B)$ should change color, while $R := (B_0 \cap B) \cup (W_0 \cap W)$ should remain the same color.

Step 1: The annotated problem

Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.



Find an arbitrary 2-coloring (B_0, W_0) of G.
$C := (B_0 \cap W) \cup (W_0 \cap B)$ should change color, while $R := (B_0 \cap B) \cup (W_0 \cap W)$ should remain the same color.

Lemma: $G \setminus S$ has the required 2-coloring if and only if S separates C and R, i.e., no component of $G \setminus S$ contains vertices from both $C \setminus S$ and $R \setminus S$.

Step 1: The annotated problem

Lemma: $G \setminus S$ has the required 2-coloring if and only if S separates C and R, i.e., no component of $G \setminus S$ contains vertices from both $C \setminus S$ and $R \setminus S$.

Proof:

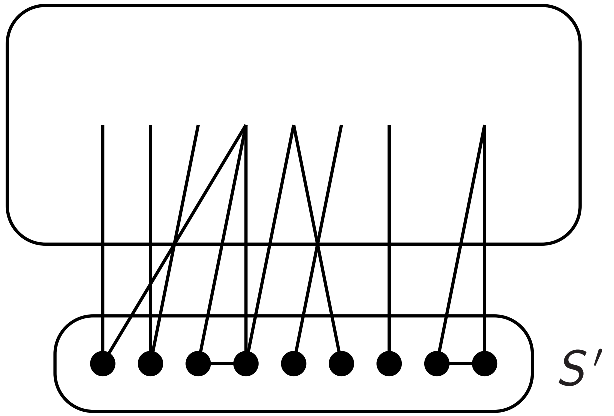
$\Rightarrow$ In a 2-coloring of $G \setminus S$, each vertex either remained the same color or changed color. Adjacent vertices do the same, thus every component either changed or remained.

$\Leftarrow$ Flip the coloring of those components of $G \setminus S$ that contain vertices from $C \setminus S$. No vertex of R is flipped.

Algorithm: Using max-flow min-cut techniques, we can check if there is a set S that separates C and R. It can be done in time $O(k|E(G)|)$ using k iterations of the Ford-Fulkerson algorithm.

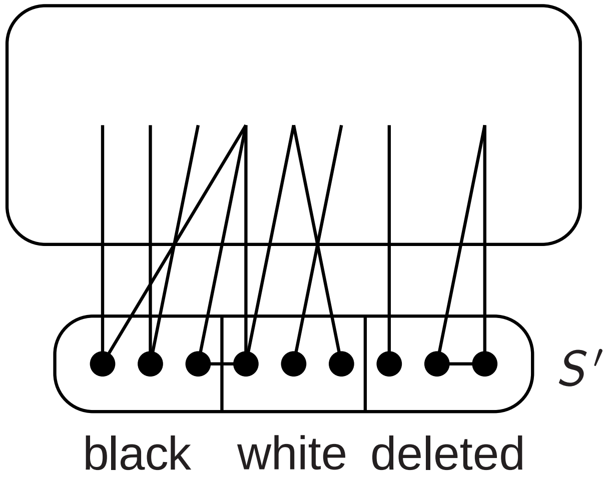
Step 2: The compression problem

Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.



Step 2: The compression problem

Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.

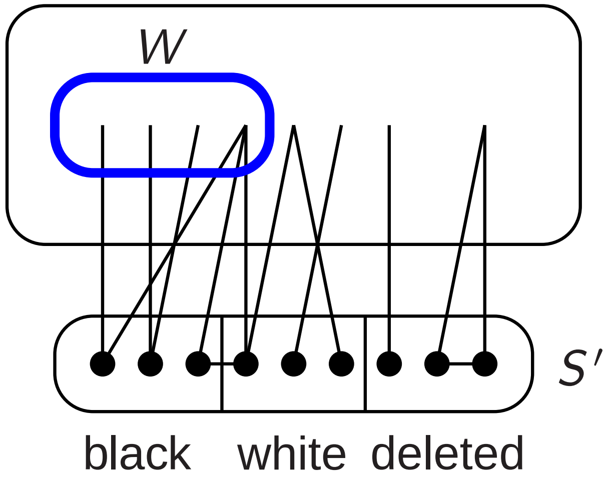


Branch into 3^{k+1} cases: each vertex of S' is either black, white, or deleted.

Trivial check: no edge between two black or two white vertices.

Step 2: The compression problem

Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.



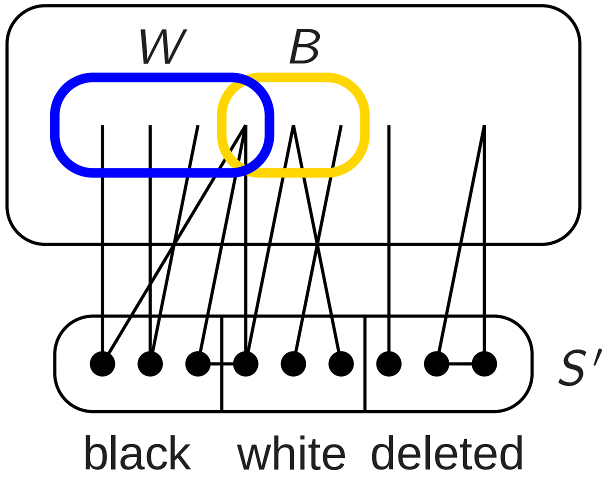
Branch into 3^{k+1} cases: each vertex of S' is either black, white, or deleted.

Trivial check: no edge between two black or two white vertices.

Neighbors of the black vertices in S' should be white and the neighbors of the white vertices in S' should be black.

Step 2: The compression problem

Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.



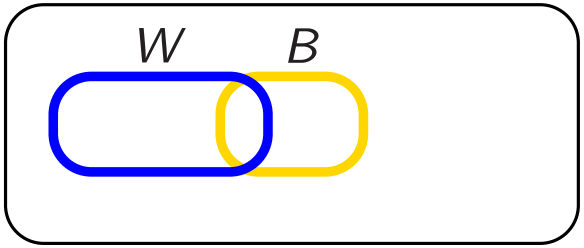
Branch into 3^{k+1} cases: each vertex of S' is either black, white, or deleted.

Trivial check: no edge between two black or two white vertices.

Neighbors of the black vertices in S' should be white and the neighbors of the white vertices in S' should be black.

Step 2: The compression problem

Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.



The vertices of S' can be disregarded. Thus we need to solve the annotated problem on the bipartite graph $G \setminus S'$.

Running time: $O(3^k \cdot k|E(G)|)$ time.

Step 3: Iterative compression

How do we get a solution of size $k+1$?

Step 3: Iterative compression

How do we get a solution of size $k + 1$?

We get it for free!

Step 3: Iterative compression

How do we get a solution of size $k+1$?

We get it for free!

Let $V(G) = \{v_1, \ldots, v_n\}$ and let G_i be the graph induced by $\{v_1, \ldots, v_i\}$.

For every i, we find a set S_i of size k such that $G_i \setminus S_i$ is bipartite.

- For G_k, the set $S_k = \{v_1, \ldots, v_k\}$ is a trivial solution.
- If S_{i-1} is known, then $S_{i-1} \cup \{v_i\}$ is a set of size $k+1$ whose deletion makes G_i bipartite $\Rightarrow$ We can use the compression algorithm to find a suitable S_i in time $O(3^k \cdot k|E(G_i)|)$.

Step 3: Iterative Compression

Bipartite-Deletion(G, k)

1. $S_k = \{v_1, \ldots, v_k\}$
2. for $i := k + 1$ to n
3. Invariant: $G_{i-1} \setminus S_{i-1}$ is bipartite.
4. Call Compression$(G_i, S_{i-1} \cup \{v_i\})$
5. If the answer is “NO” $\Rightarrow$ return “NO”
6. If the answer is a set $X \Rightarrow S_i := X$
7. Return the set S_n

Running time: the compression algorithm is called n times and everything else can be done in linear time

$\Rightarrow O(3^k \cdot k|V(G)| \cdot |E(G)|)$ time algorithm.

Graph Minors

Neil Robertson

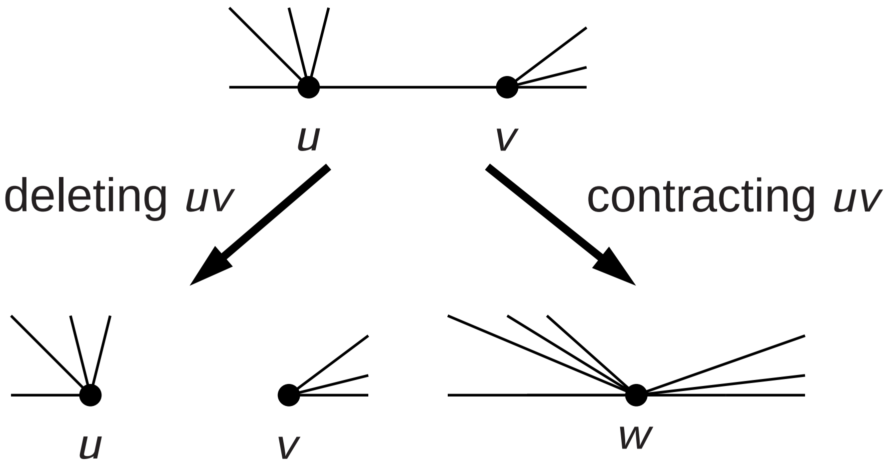
Paul Seymour

Graph Minors

- Some consequences of the Graph Minors Theorem give a quick way of showing that certain problems are FPT.
- However, the function $f(k)$ in the resulting FPT algorithms can be HUGE, completely impractical.
- History: motivation for FPT.
- Parts and ingredients of the theory are useful for algorithm design.
- New algorithmic results are still being developed.

Graph Minors

Definition: Graph H is a **minor** G ($H \leq G$) if H can be obtained from G by deleting edges, deleting vertices, and contracting edges.

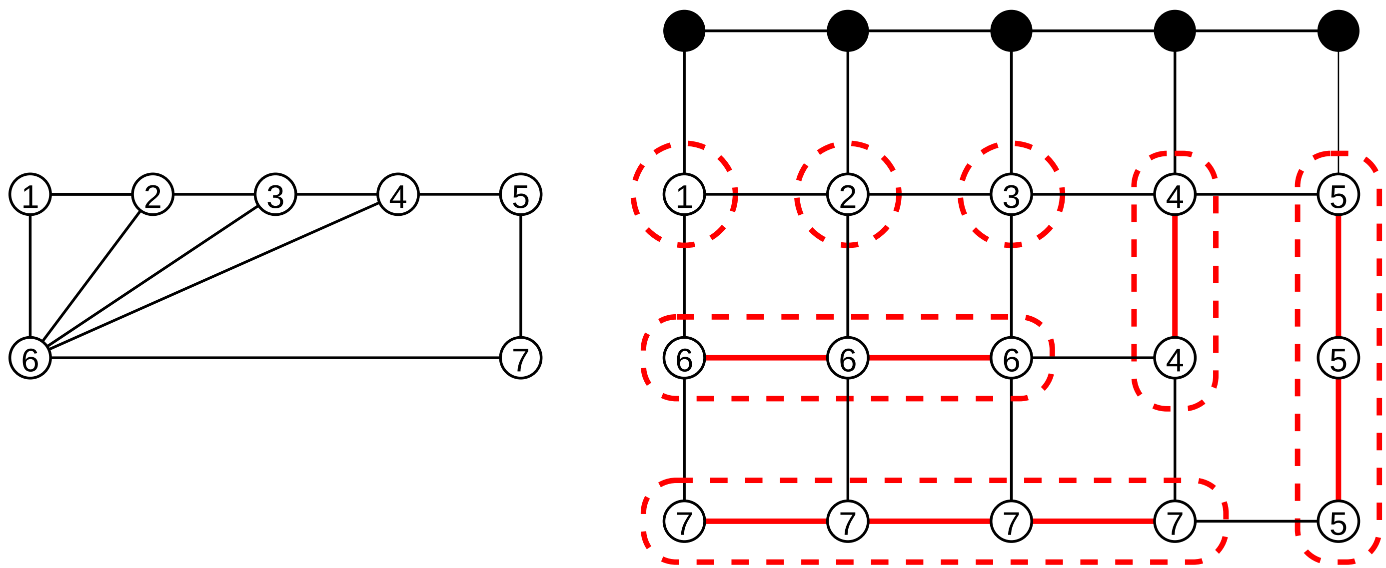


Example: A triangle is a minor of a graph G if and only if G has a cycle (i.e., it is not a forest).

Graph minors

Equivalent definition: Graph H is a **minor** of G if there is a mapping ϕ that maps each vertex of H to a connected subset of G such that

- $\phi(u)$ and $\phi(v)$ are disjoint if $u \neq v$, and
- if $uv \in E(G)$, then there is an edge between $\phi(u)$ and $\phi(v)$.



Minor closed properties

Definition: A set $\mathcal{G}$ of graphs is **minor closed** if whenever $G \in \mathcal{G}$ and $H \leq G$, then $H \in \mathcal{G}$ as well.

Examples of minor closed properties:

planar graphs

acyclic graphs (forests)

graphs having no cycle longer than k

empty graphs

Examples of not minor closed properties:

complete graphs

regular graphs

bipartite graphs

Forbidden minors

Let $\mathcal{G}$ be a minor closed set and let $\mathcal{F}$ be the set of "minimal bad graphs": $H \in \mathcal{F}$ if $H \notin \mathcal{G}$, but every proper minor of H is in $\mathcal{G}$.

Characterization by forbidden minors:

$$G \in \mathcal{G} \iff \forall H \in \mathcal{F}, H \not\leq G$$

The set $\mathcal{F}$ is the **obstruction set** of property $\mathcal{G}$.

Forbidden minors

Let $\mathcal{G}$ be a minor closed set and let $\mathcal{F}$ be the set of "minimal bad graphs": $H \in \mathcal{F}$ if $H \notin \mathcal{G}$, but every proper minor of H is in $\mathcal{G}$.

Characterization by forbidden minors:

$$G \in \mathcal{G} \iff \forall H \in \mathcal{F}, H \not\leq G$$

The set $\mathcal{F}$ is the **obstruction set** of property $\mathcal{G}$.

Theorem: [Wagner] A graph is planar if and only if it does not have a K_5 or $K_{3,3}$ minor.

In other words: the obstruction set of planarity is $\mathcal{F} = \{K_5, K_{3,3}\}$.

Does every minor closed property have such a finite characterization?

Graph Minors Theorem

Theorem: [Robertson and Seymour] Every minor closed property $\mathcal{G}$ has a finite obstruction set.

Note: The proof is contained in the paper series “Graph Minors I–XX”.

Note: The size of the obstruction set can be astronomical even for simple properties.

Graph Minors Theorem

Theorem: [Robertson and Seymour] Every minor closed property $\mathcal{G}$ has a finite obstruction set.

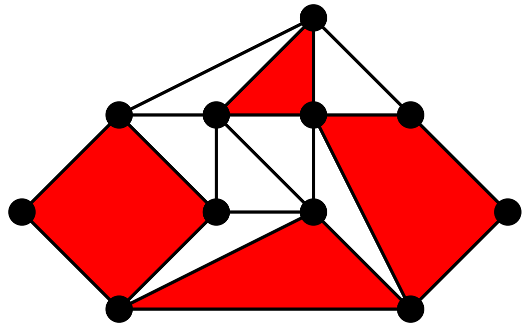
Note: The proof is contained in the paper series "Graph Minors I–XX".
Note: The size of the obstruction set can be astronomical even for simple properties.

Theorem: [Robertson and Seymour] For every fixed graph H, there is an $O(n^3)$ time algorithm for testing whether H is a minor of the given graph G.

Corollary: For every minor closed property $\mathcal{G}$, there is an $O(n^3)$ time algorithm for testing whether a given graph G is in $\mathcal{G}$.

Applications

PLANAR FACE COVER: Given a graph G and an integer k, find an embedding of planar graph G such that there are k faces that cover all the vertices.



One line argument:

For every fixed k, the class $\mathcal{G}_k$ of graphs of yes-instances is minor closed.

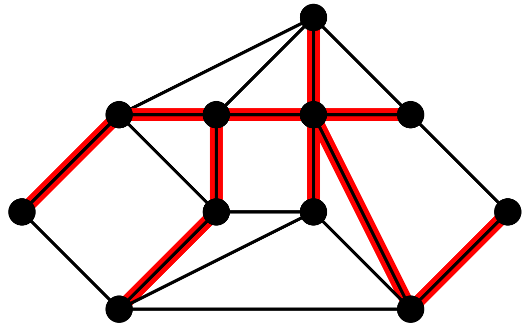
$$\Downarrow$$

For every fixed k, there is a $O(n^3)$ time algorithm for PLANAR FACE COVER.

Note: non-uniform FPT.

Applications

k-LEAF SPANNING TREE: Given a graph G and an integer k, find a spanning tree with **at least** k leaves.



Technical modification: Is there such a spanning tree for at least one component of G?

One line argument:

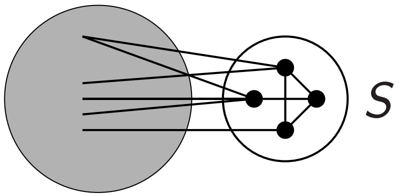
For every fixed k, the class $\mathcal{G}_k$ of no-instances is minor closed.

$$\Downarrow$$

For every fixed k, k-LEAF SPANNING TREE can be solved in time $O(n^3)$.

$\mathcal{G} + k$ vertices

Let $\mathcal{G}$ be a graph property, and let $\mathcal{G} + kv$ contain graph G if there is a set $S \subseteq V(G)$ of k vertices such that $G \setminus S \in \mathcal{G}$.

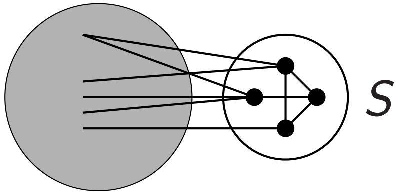


Lemma: If $\mathcal{G}$ is minor closed, then $\mathcal{G} + kv$ is minor closed for every fixed k.

$\Rightarrow$ It is (nonuniform) FPT to decide if G can be transformed into a member of $\mathcal{G}$ by deleting k vertices.

$\mathcal{G} + k$ vertices

Let $\mathcal{G}$ be a graph property, and let $\mathcal{G} + kv$ contain graph G if there is a set $S \subseteq V(G)$ of k vertices such that $G \setminus S \in \mathcal{G}$.



Lemma: If $\mathcal{G}$ is minor closed, then $\mathcal{G} + kv$ is minor closed for every fixed k.
$\Rightarrow$ It is (nonuniform) FPT to decide if G can be transformed into a member of $\mathcal{G}$ by deleting k vertices.

- If $\mathcal{G} =$ forests $\Rightarrow \mathcal{G} + kv =$ graphs that can be made acyclic by the deletion of k vertices $\Rightarrow$ FEEDBACK VERTEX SET is FPT.
- If $\mathcal{G} =$ planar graphs $\Rightarrow \mathcal{G} + kv =$ graphs that can be made planar by the deletion of k vertices (k-apex graphs) $\Rightarrow$ k-APEX GRAPH is FPT.
- If $\mathcal{G} =$ empty graphs $\Rightarrow \mathcal{G} + kv =$ graphs with vertex cover number at most k $\Rightarrow$ VERTEX COVER is FPT.

Color coding

Color coding

- Works best when we need to ensure that a small number of “things” are disjoint.
- We demonstrate it on the problem of finding an s-t path of length **exactly** k.
- Randomized algorithm, but can be derandomized using a standard technique.
- Very robust technique, we can use it as an “opening step” when investigating a new problem.

k-PATH

Task: Given a graph G, an integer k, two vertices s, t, find a **simple** s-t path with exactly k internal vertices.

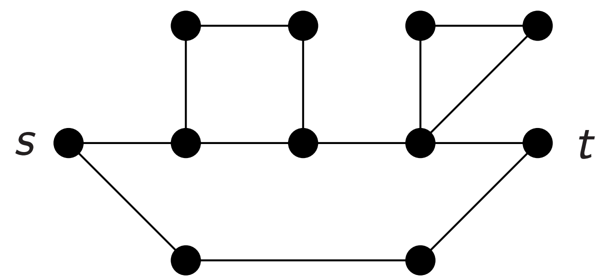
Note: Finding such a **walk** can be done easily in polynomial time.

Note: The problem is clearly NP-hard, as it contains the s-t HAMILTONIAN PATH problem.

The k-PATH algorithm can be used to check if there is a cycle of length exactly k in the graph.

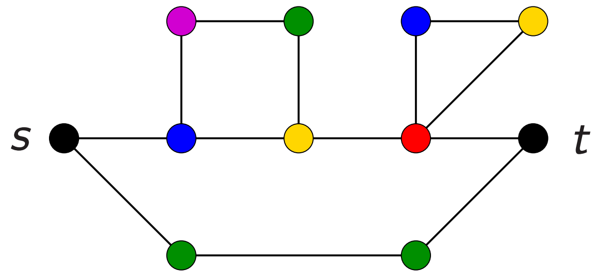
k-PATH

- Assign colors from $[k]$ to vertices $V(G) \setminus \{s, t\}$ uniformly and independently at random.



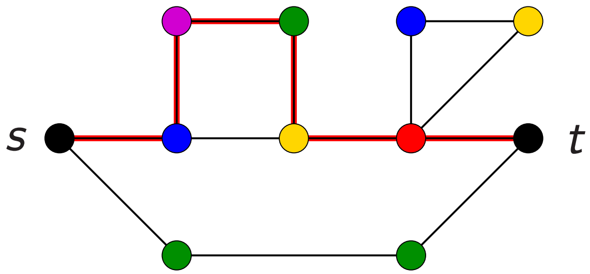
k-PATH

- Assign colors from $[k]$ to vertices $V(G) \setminus \{s, t\}$ uniformly and independently at random.



k-PATH

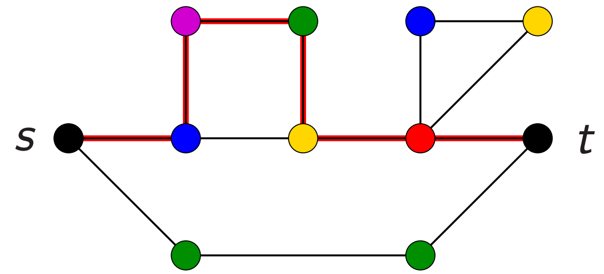
- Assign colors from $[k]$ to vertices $V(G) \setminus \{s, t\}$ uniformly and independently at random.



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

k-PATH

- Assign colors from $[k]$ to vertices $V(G) \setminus \{s, t\}$ uniformly and independently at random.



- Check if there is a **colorful** s-t path: a path where each color appears exactly once on the internal vertices; output “YES” or “NO”.
  - If there is no s-t k-path: no such colorful path exists $\Rightarrow$ “NO”.
  - If there is an s-t k-path: the probability that such a path is colorful is

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

thus the algorithm outputs “YES” with at least that probability.

Error probability

- **Useful fact:** If the probability of success is at least p, then the probability that the algorithm **does not** say “YES” after $1/p$ repetitions is at most

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

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

Error probability

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

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

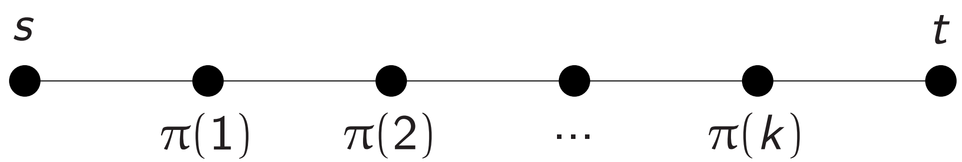
- Thus if $p > e^{-k}$, then error probability is at most $1/e$ after e^k repetitions.
- Repeating the whole algorithm a constant number of times can make the error probability an arbitrary small constant.
- For example, by trying $100 \cdot e^k$ random colorings, the probability of a wrong answer is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

It remains to see how a colorful s-t path can be found.

Method 1: Trying all permutations.
Method 2: Dynamic programming.

Method 1: Trying all permutations

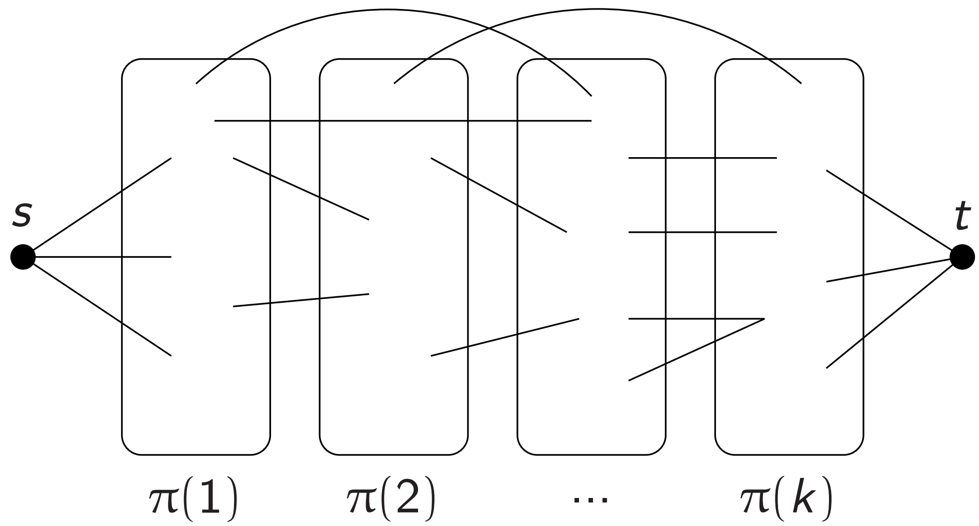
The colors encountered on a colorful s-t path form a permutation π of $\{1, 2, \ldots, k\}$:



We try all possible $k!$ permutations. For a fixed π, it is easy to check if there is a path with this order of colors.

Method 1: Trying all permutations

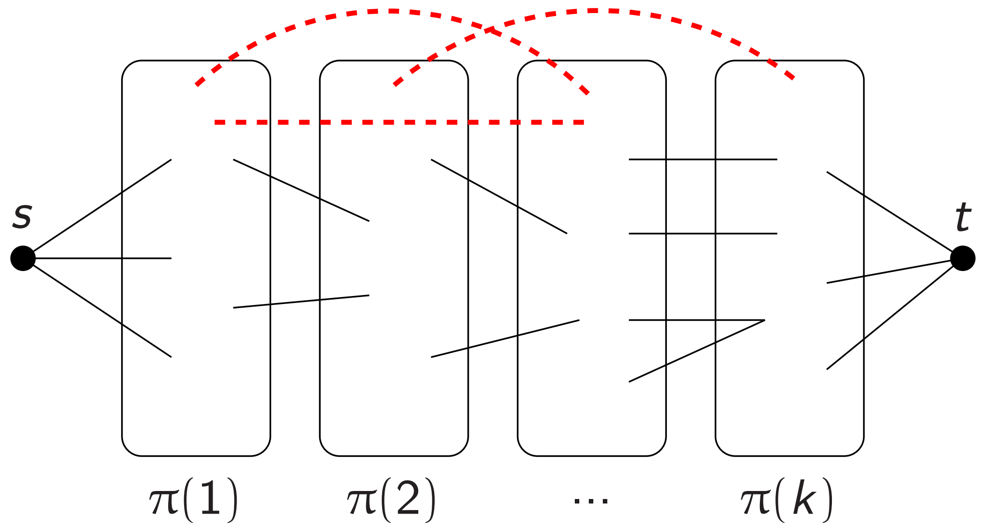
We try all possible $k!$ permutations. For a fixed π, it is easy to check if there is a path with this order of colors.



- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed.
- All we need to check if there is a directed s-t path.
- Running time is $O(k! \cdot |E(G)|)$.

Method 1: Trying all permutations

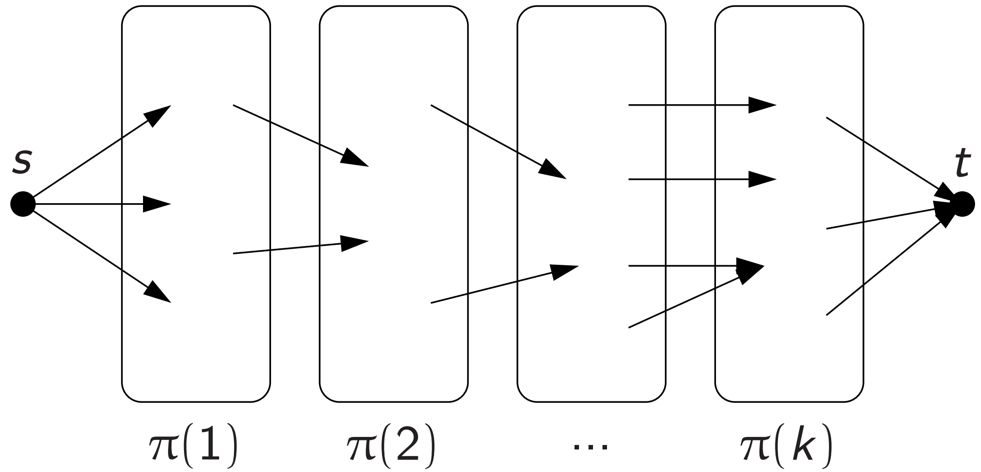
We try all possible $k!$ permutations. For a fixed π, it is easy to check if there is a path with this order of colors.



- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed.
- All we need to check if there is a directed s-t path.
- Running time is $O(k! \cdot |E(G)|)$.

Method 1: Trying all permutations

We try all possible $k!$ permutations. For a fixed π, it is easy to check if there is a path with this order of colors.



- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed.
- All we need to check if there is a directed s-t path.
- Running time is $O(k! \cdot |E(G)|)$.

Method 2: Dynamic Programming

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

$$\boxed{\begin{array}{c} x(v, C) = \text{TRUE for some } v \in V(G) \text{ and } C \subseteq [k] \\ \Updownarrow \\ \text{There is an } s\text{-}v \text{ path where each color in } C \text{ appears exactly once} \\ \text{and no other color appears.} \end{array}}$$

Method 2: Dynamic Programming

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

$$\boxed{\begin{array}{c} x(v, C) = \text{TRUE for some } v \in V(G) \text{ and } C \subseteq [k] \\ \Updownarrow \\ \text{There is an } s\text{-}v \text{ path where each color in } C \text{ appears exactly once} \\ \text{and no other color appears.} \end{array}}$$

Clearly, $x(s, \emptyset) = \text{TRUE}$. Recurrence for vertex v with color r:

$$x(v, C) = \bigvee_{uv \in E(G)} x(u, C \setminus \{r\})$$

Method 2: Dynamic Programming

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

$$\boxed{\begin{array}{c} x(v, C) = \text{TRUE for some } v \in V(G) \text{ and } C \subseteq [k] \\ \Updownarrow \\ \text{There is an } s\text{-}v \text{ path where each color in } C \text{ appears exactly once} \\ \text{and no other color appears.} \end{array}}$$

Clearly, $x(s, \emptyset) = \text{TRUE}$. Recurrence for vertex v with color r:

$$x(v, C) = \bigvee_{uv \in E(G)} x(u, C \setminus \{r\})$$

If we know every $x(v, C)$ with $|C| = i$, then we can determine every $x(v, C)$ with $|C| = i + 1$ $\Rightarrow$ All the values can be determined in time $O(2^k \cdot |E(G)|)$.

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

Derandomization

Using Method 2, we obtain a $O^*((2e)^k)$ time algorithm with constant error probability. How to make it deterministic?

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

Derandomization

Using Method 2, we obtain a $O^*((2e)^k)$ time algorithm with constant error probability. How to make it deterministic?

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

Instead of trying $O(e^k)$ random colorings, we go through a k-perfect family $\mathcal{H}$ of functions $V(G) \to [k]$. If there is a solution $\Rightarrow$ The internal vertices S are colorful for at least one $h \in \mathcal{H}$ $\Rightarrow$ Algorithm outputs “YES”.

Derandomization

Using Method 2, we obtain a $O^*((2e)^k)$ time algorithm with constant error probability. How to make it deterministic?

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

Instead of trying $O(e^k)$ random colorings, we go through a k-perfect family $\mathcal{H}$ of functions $V(G) \to [k]$. If there is a solution $\Rightarrow$ The internal vertices S are colorful for at least one $h \in \mathcal{H}$ $\Rightarrow$ Algorithm outputs "YES".

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

$\Rightarrow$ There is a **deterministic** $2^{O(k)} \cdot n^{O(1)}$ time algorithm for the k-PATH problem.

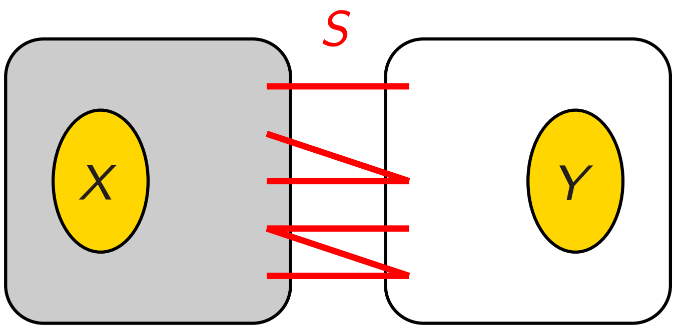
Cut problems

MULTIWAY CUT

Task: Given a graph G, a set T of vertices, and an integer k, find a set S of at most k edges that separates T (each component of $G \setminus S$ contains at most one vertex of T).

Polynomial for $|T| = 2$, but NP-hard for $|T| = 3$.

Theorem: MULTIWAY CUT is FPT parameterized by k.



$\delta(R)$: set of edges leaving R
$\lambda(X,Y)$: minimum number of edges in an (X,Y)-separator

Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

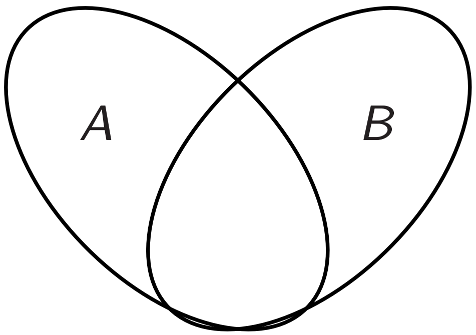
$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

Proof: Determine separately the contribution of the different types of edges.



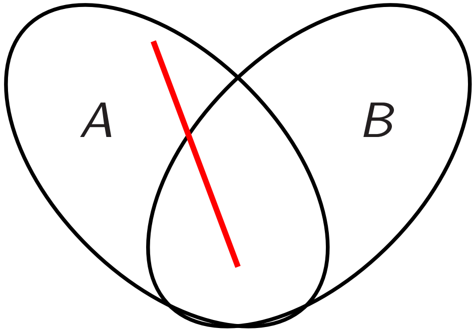
Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

$$0 \qquad\qquad 1 \qquad\qquad 1 \qquad\qquad 0$$

Proof: Determine separately the contribution of the different types of edges.



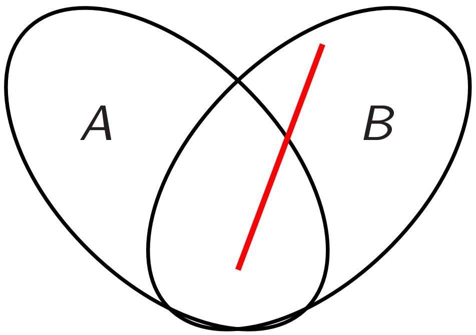
Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

$$1 \qquad\qquad 0 \qquad\qquad 1 \qquad\qquad 0$$

Proof: Determine separately the contribution of the different types of edges.



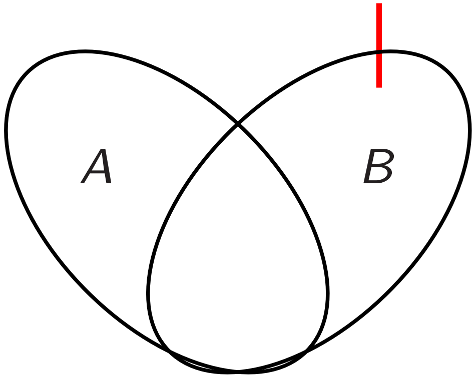
Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

$$0 \qquad\qquad 1 \qquad\qquad 0 \qquad\qquad 1$$

Proof: Determine separately the contribution of the different types of edges.



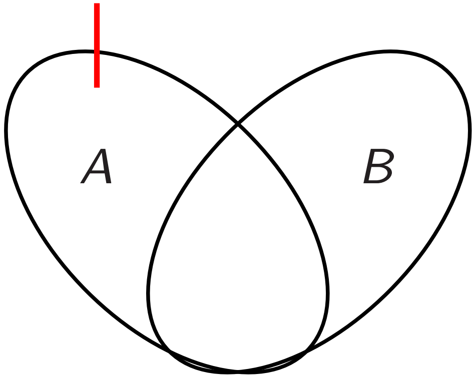
Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

$$1 \qquad\qquad 0 \qquad\qquad\qquad 0 \qquad\qquad\qquad 1$$

Proof: Determine separately the contribution of the different types of edges.



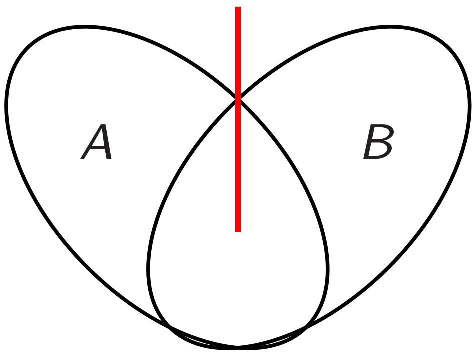
Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| + |\delta(B)| \geq |\delta(A \cap B)| + |\delta(A \cup B)|$$

$$1 \qquad\qquad 1 \qquad\qquad\qquad 1 \qquad\qquad\qquad 1$$

Proof: Determine separately the contribution of the different types of edges.



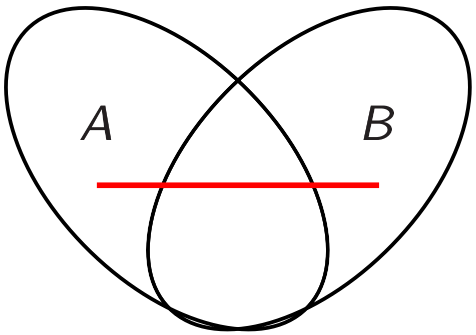
Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| \quad + \quad |\delta(B)| \quad \geq \quad |\delta(A \cap B)| \quad + \quad |\delta(A \cup B)|$$

$$1 \qquad\qquad 1 \qquad\qquad 0 \qquad\qquad 0$$

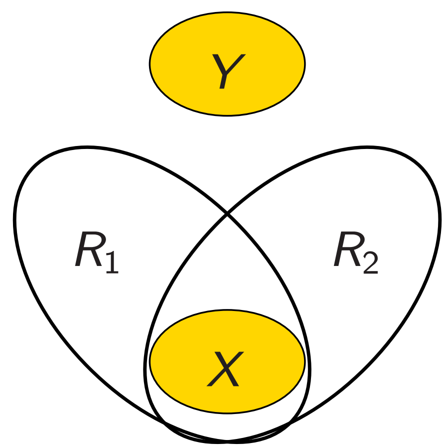
Proof: Determine separately the contribution of the different types of edges.



Submodularity

Consequence: There is a unique maximal $R_{\max} \supseteq X$ such that $\delta(R_{\max})$ is an (X, Y)-separator of size $\lambda(X, Y)$.

Proof: Let $R_1, R_2 \supseteq X$ be two sets such that $\delta(R_1), \delta(R_2)$ are (X, Y)-separators of size $\lambda := \lambda(X, Y)$.



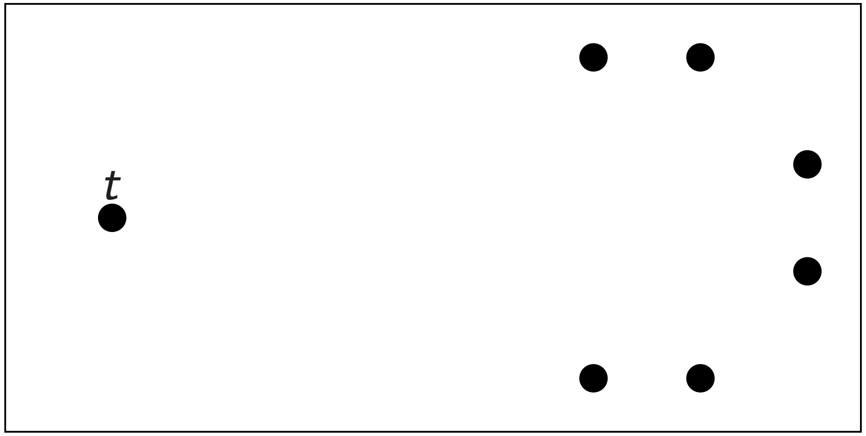
$$\begin{array}{ccccccc} |\delta(R_1)| & + & |\delta(R_2)| & \geq & |\delta(R_1 \cap R_2)| & + & |\delta(R_1 \cup R_2)| \\ \lambda & & \lambda & & \geq \lambda & & \end{array}$$

$\Rightarrow |\delta(R_1 \cup R_2)| \leq \lambda$

Note: Analogous result holds for a unique minimal $R_{\min}$.

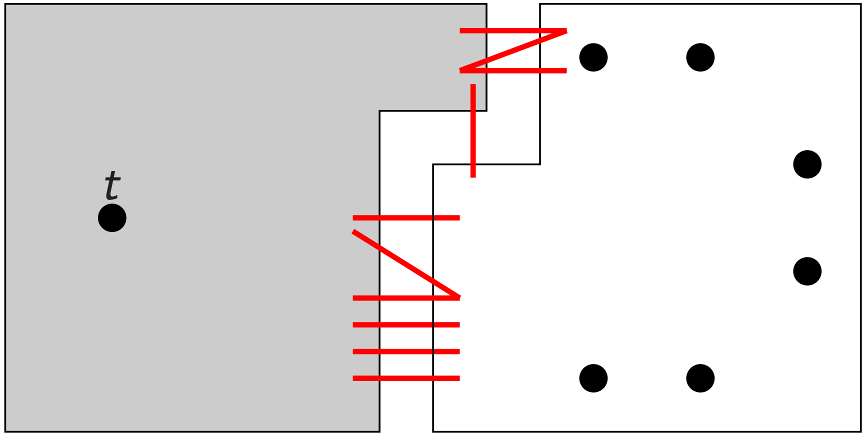
MULTIWAY CUT

Intuition: Consider a $t \in T$. A subset of the solution separates t and $T \setminus t$.



MULTIWAY CUT

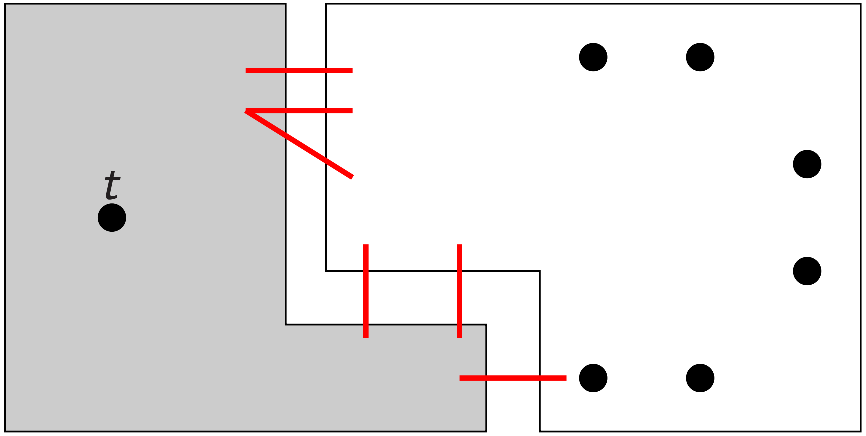
Intuition: Consider a $t \in T$. A subset of the solution separates t and $T \setminus t$.



There are many such separators.

MULTIWAY CUT

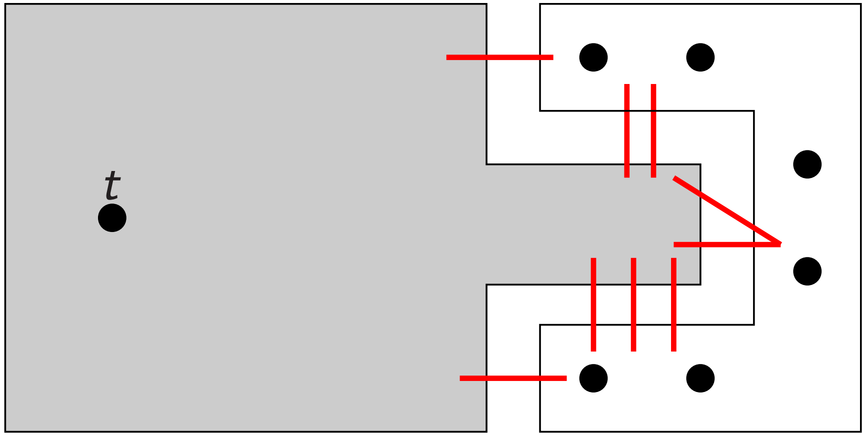
Intuition: Consider a $t \in T$. A subset of the solution separates t and $T \setminus t$.



There are many such separators.

MULTIWAY CUT

Intuition: Consider a $t \in T$. A subset of the solution separates t and $T \setminus t$.

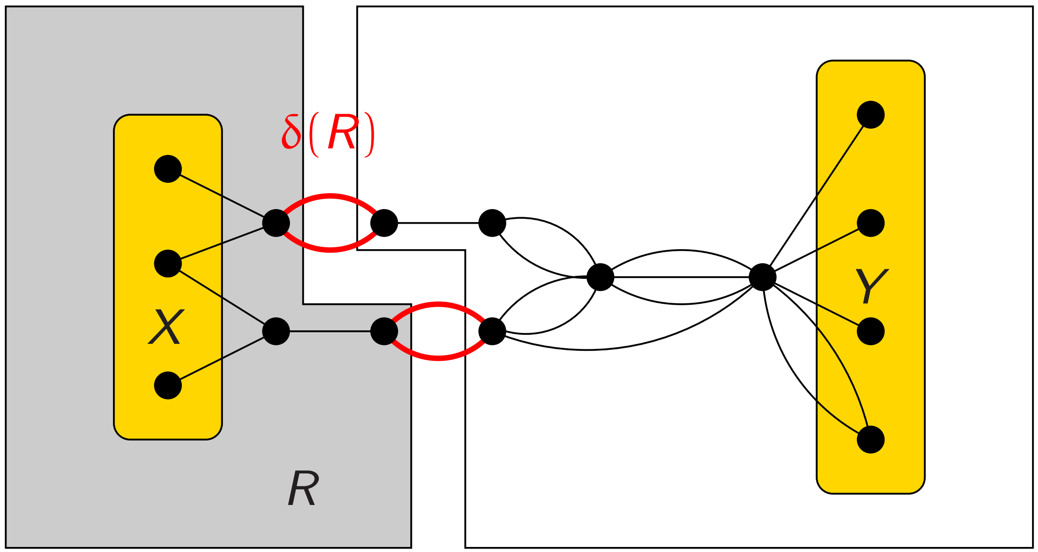


There are many such separators.

But a separator farther from t and closer to $T \setminus t$ seems to be more useful.

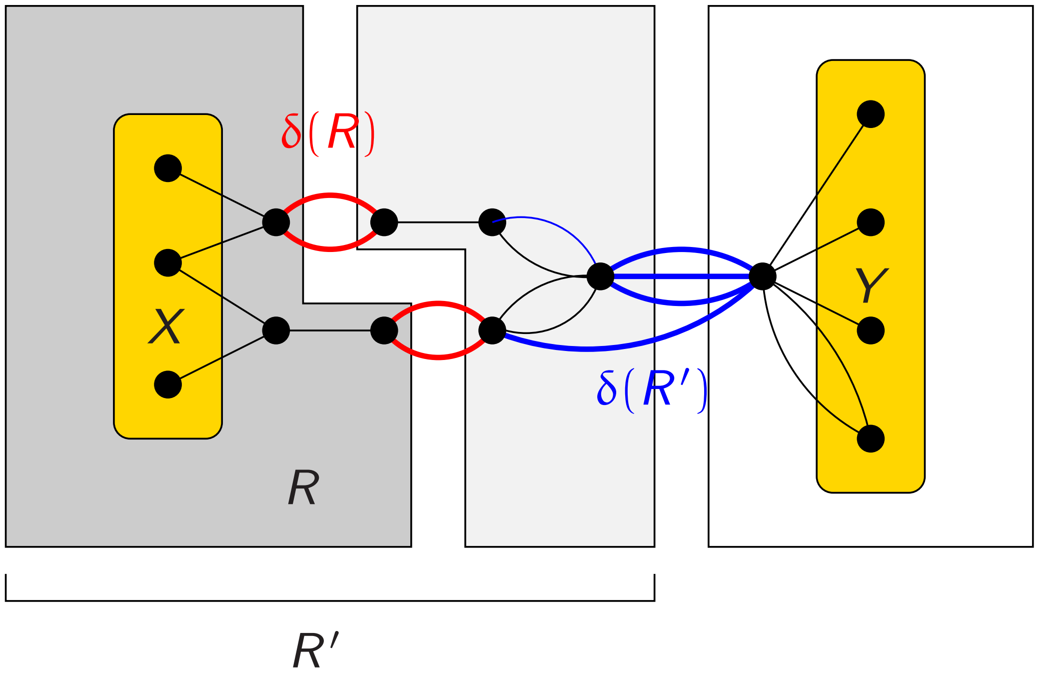
Important separators

Definition: An (X, Y)-separator $\delta(R)$ is **important** if there is no (X, Y)-separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \leq |\delta(R)|$.



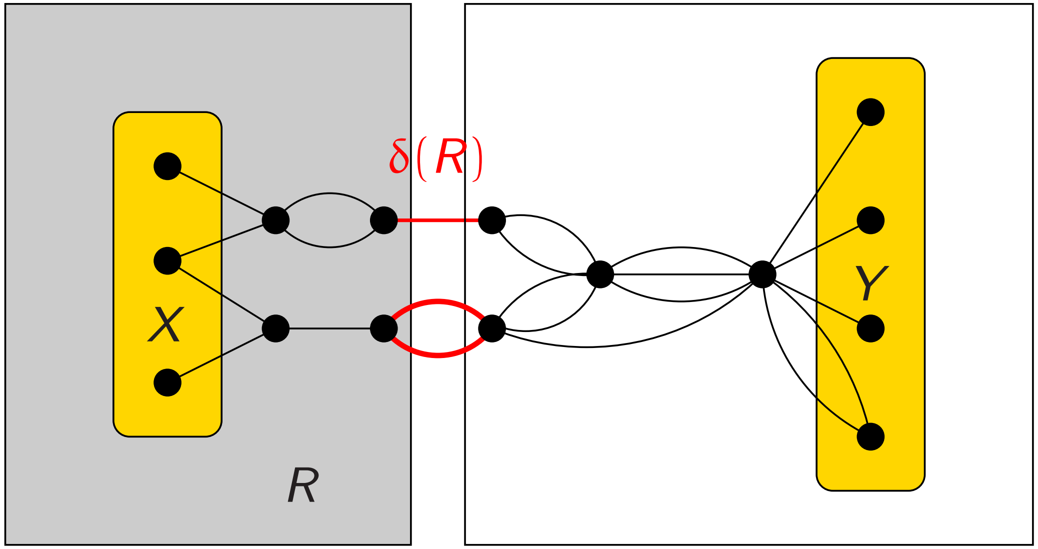
Important separators

Definition: An (X, Y)-separator $\delta(R)$ is **important** if there is no (X, Y)-separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \leq |\delta(R)|$.



Important separators

Definition: An (X, Y)-separator $\delta(R)$ is **important** if there is no (X, Y)-separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \le |\delta(R)|$.



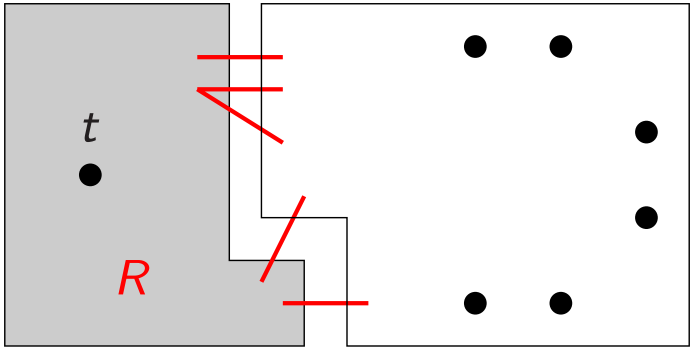
MULTIWAY CUT *and important separators*

Lemma: Let $t \in T$. The MULTIWAY CUT problem has a solution S that contains an important $(t, T \setminus t)$-separator.

MULTIWAY CUT *and important separators*

Lemma: Let $t \in T$. The MULTIWAY CUT problem has a solution S that contains an important $(t, T \setminus t)$-separator.

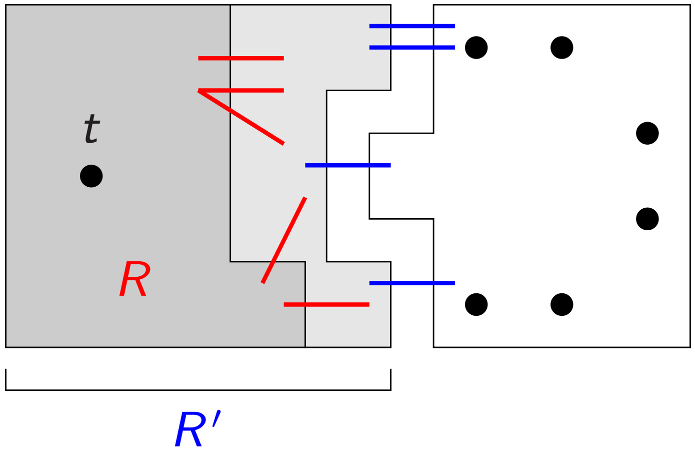
Proof: Let R be the vertices reachable from t in $G \setminus S$ for a solution S.



MULTIWAY CUT *and important separators*

Lemma: Let $t \in T$. The MULTIWAY CUT problem has a solution S that contains an important $(t, T \setminus t)$-separator.

Proof: Let R be the vertices reachable from t in $G \setminus S$ for a solution S.

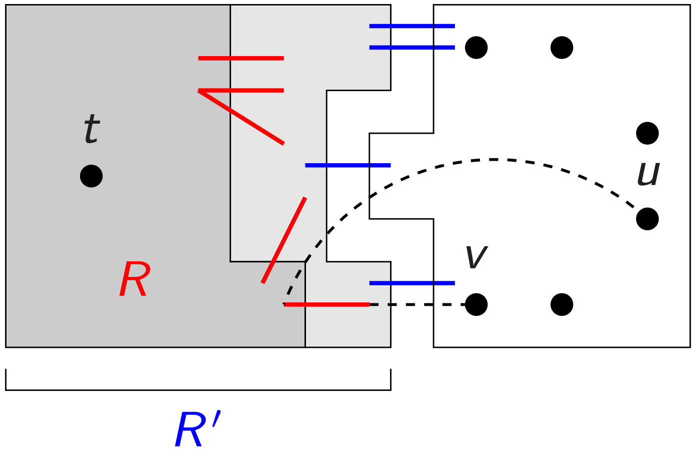


If $\delta(R)$ is not important, then there is an important separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \leq |\delta(R)|$. Replace S with $S' := (S \setminus \delta(R)) \cup \delta(R') \Rightarrow |S'| \leq |S|$

MULTIWAY CUT *and important separators*

Lemma: Let $t \in T$. The MULTIWAY CUT problem has a solution S that contains an important $(t, T \setminus t)$-separator.

Proof: Let R be the vertices reachable from t in $G \setminus S$ for a solution S.



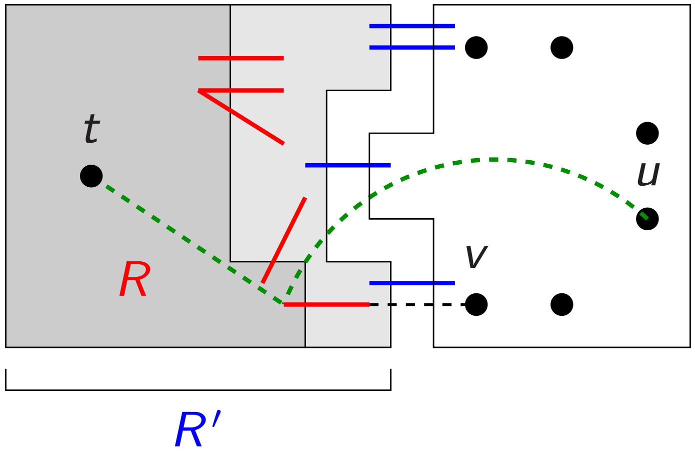
If $\delta(R)$ is not important, then there is an important separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \le |\delta(R)|$. Replace S with $S' := (S \setminus \delta(R)) \cup \delta(R') \Rightarrow |S'| \le |S|$

S' is a multiway cut: A u-v path in $G \setminus S'$ implies a u-t path, a contradiction.

MULTIWAY CUT *and important separators*

Lemma: Let $t \in T$. The MULTIWAY CUT problem has a solution S that contains an important $(t, T \setminus t)$-separator.

Proof: Let R be the vertices reachable from t in $G \setminus S$ for a solution S.



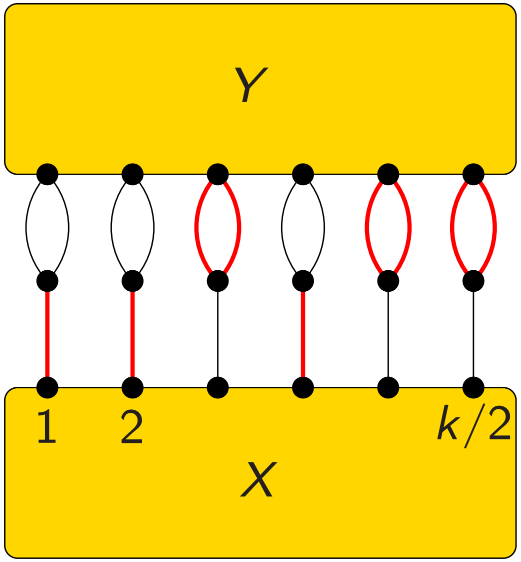
If $\delta(R)$ is not important, then there is an important separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \leq |\delta(R)|$. Replace S with $S' := (S \setminus \delta(R)) \cup \delta(R') \Rightarrow |S'| \leq |S|$

S' is a multiway cut: A u-v path in $G \setminus S'$ implies a u-t path, a contradiction.

Important separators

Lemma: There are at most 4^k important (X, Y)-separators of size at most k.

Example:



There are exactly $2^{k/2}$ important (X, Y)-separators of size at most k in this graph.

Important separators

Lemma: There are at most 4^k important (X, Y)-separators of size at most k.

Proof: First we show that $R_{\max} \subseteq R$ for every important separator $\delta(R)$.

$$\underbrace{|\delta(R_{\max})|}_{\lambda} + |\delta(R)| \geq \underbrace{|\delta(R_{\max} \cap R)|}_{\geq \lambda} + |\delta(R_{\max} \cup R)|$$

$$\Downarrow$$

$$|\delta(R_{\max} \cup R)| \leq |\delta(R)|$$

$$\Downarrow$$

If $R \neq R_{\max} \cup R$, then $\delta(R)$ is not important.

Thus the important (X, Y)- and $(R_{\max}, Y)$-separators are the same.

$\Rightarrow$ We can assume $X = R_{\max}$.

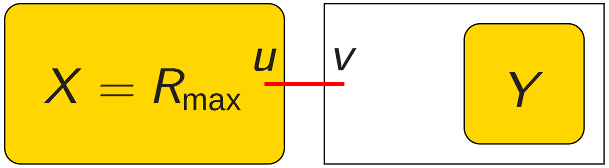
Important separators

Lemma: There are at most 4^k important (X, Y)-separators of size at most k.

Search tree algorithm for finding all these separators:

An (arbitrary) edge uv leaving $X = R_{\max}$ is either in the separator or not.

Branch 1: If $uv \in S$, then $S \setminus uv$ is an important (X, Y)-separator of size at most $k - 1$ in $G \setminus uv$.



Branch 2: If $uv \notin S$, then S is an important $(X \cup v, Y)$-separator of size at most k in G.

Important separators

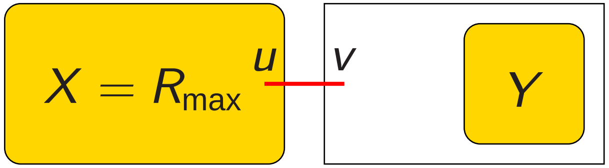
Lemma: There are at most 4^k important (X, Y)-separators of size at most k.

Search tree algorithm for finding all these separators:

An (arbitrary) edge uv leaving $X = R_{\max}$ is either in the separator or not.

Branch 1: If $uv \in S$, then $S \setminus uv$ is an important (X, Y)-separator of size at most $k - 1$ in $G \setminus uv$.

$\Rightarrow$ k decreases by one, λ decreases by at most 1.



Branch 2: If $uv \notin S$, then S is an important $(X \cup v, Y)$-separator of size at most k in G.

$\Rightarrow$ k remains the same, λ increases by 1.

The measure $2k - \lambda$ decreases in each step.

$\Rightarrow$ Height of the search tree $\leq 2k \Rightarrow \leq 2^{2k} = 4^k$ important separators.

Algorithm for MULTIWAY CUT

1. If every vertex of T is in a different component, then we are done.
2. Let $t \in T$ be a vertex with that is not separated from every $T \setminus t$.
3. Branch on a choice of an important $(t, T \setminus t)$ separator S of size at most k.
4. Set $G := G \setminus S$ and $k := k - |S|$.
5. Go to step 1.

We branch into at most 4^k directions at most k times.

Algorithm for MULTIWAY CUT

1. If every vertex of T is in a different component, then we are done.
2. Let $t \in T$ be a vertex with that is not separated from every $T \setminus t$.
3. Branch on a choice of an important $(t, T \setminus t)$ separator S of size at most k.
4. Set $G := G \setminus S$ and $k := k - |S|$.
5. Go to step 1.

We branch into at most 4^k directions at most k times.
Better estimate of the search tree size:

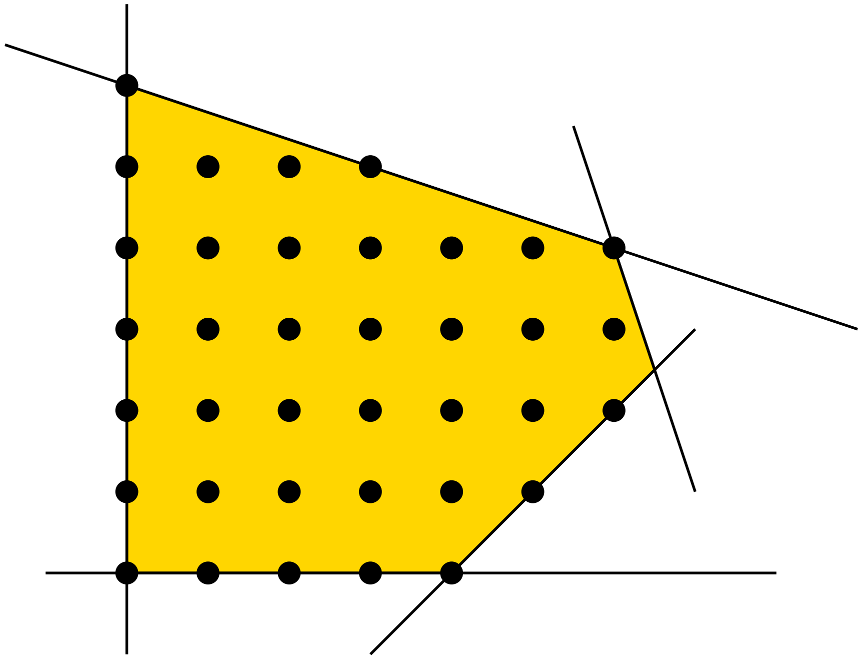
- When choosing the important separator, $2k - \lambda$ decreases at each branching, until λ reaches 0.
- When choosing the next vertex t, λ changes from 0 to positive, thus $2k - \lambda$ does not increase.

Size of the search tree is at most $2^{2k} = 4^k$.

Other separation problems

- Some other variants:
  - $|T|$ as a parameter
  - MULTITERMINAL CUT: pairs $(s_1, t_1), \ldots, (s_\ell, t_\ell)$ have to be separated.
  - Directed graphs
  - Planar graphs
- Useful for deletion-type problems such as DIRECTED FEEDBACK VERTEX SET (via iterative compression).
- Important separators: is it relevant for a given problem?

Integer Linear Programming



Integer Linear Programming

Linear Programming (LP): important tool in (continuous) combinatorial optimization. Sometimes very useful for discrete problems as well.

$$\max c_1x_1 + c_2x_2 + c_3x_3$$

$$\text{s.t.}$$

$$x_1 + 5x_2 - x_3 \leq 8$$

$$2x_1 - x_3 \leq 0$$

$$3x_2 + 10x_3 \leq 10$$

$$x_1, x_2, x_3 \in \mathbb{R}$$

Fact: It can be decided if there is a solution (feasibility) and an optimum solution can be found in polynomial time.

Integer Linear Programming

Integer Linear Programming (ILP): Same as LP, but we require that every x_i is integer.

Very powerful, able to model many NP-hard problems. (Of course, no polynomial-time algorithm is known.)

Theorem: ILP with p variables can be solved in time $p^{O(p)} \cdot n^{O(1)}$.

Closest String

Task: Given strings $s_1, \dots, s_k$ of length L over alphabet Σ, and an integer d, find a string s (of length L) such that $d(s, s_i) \leq d$ for every $1 \leq i \leq k$.

Note: $d(s, s_i)$ is the Hamming distance.

Theorem: Closest String parameterized by k is FPT.

Theorem: Closest String parameterized by d is FPT.

Theorem: Closest String parameterized by L is FPT.

Theorem: Closest String is NP-hard for $\Sigma = \{0, 1\}$.

Closest String

Task: Given strings $s_1, \ldots, s_k$ of length L over alphabet Σ, and an integer d, find a string s (of length L) such that $d(s, s_i) \leq d$ for every $1 \leq i \leq k$.

Note: $d(s, s_i)$ is the Hamming distance.

Theorem: Closest String parameterized by k is FPT.

Theorem: Closest String parameterized by d is FPT.

Theorem: Closest String parameterized by L is FPT.

Theorem: Closest String is NP-hard for $\Sigma = \{0, 1\}$.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	CBDCCACBB
s_2	ABDBCABDB
s_3	CDDBACCBD
s_4	DDABACCBD
s_5	ACDBDDCBC
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.

The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	**CB**D**C**CACB**B**
s_2	A**B**DBCA**BDB**
s_3	**C**DDB**AC**CBD
s_4	**D**D**A**B**AC**CBD
s_5	A**C**DB**DD**CB**C**
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.

The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	CBDCCACBB
s_2	ABDBCABDB
s_3	CDDBACCBD
s_4	DDABACCBD
s_5	ACDBDDCBC
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.

The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	CBDCCACBB
s_2	ABDBCABDB
s_3	CDDBACCBD
s_4	DDABACCBD
s_5	ACDBDDCBC
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.

The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	CBDCCACBB
s_2	ABDBCABDB
s_3	CDDBACCBD
s_4	DDABACCBD
s_5	ACDBDDCBC
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.

The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	CBDCCACBB
s_2	ABDBCABDB
s_3	CDDBACCBD
s_4	DDABACCBD
s_5	ACDBDDCBC
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.

The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

CLOSEST STRING

Each column can be described by a partition $\mathcal{P}$ of $[k]$.
The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Describing a solution: If C is a class of $\mathcal{P}$, let $x_{\mathcal{P},C}$ be the number of type $\mathcal{P}$ columns where the solution agrees with class C.

There is a solution iff the following ILP has a feasible solution:

$$\sum_{C \in \mathcal{P}} x_{\mathcal{P},C} \leq c_{\mathcal{P}} \qquad \forall \text{partition } \mathcal{P}$$

$$\sum_{i \notin C, C \in \mathcal{P}} x_{\mathcal{P},C} \leq d \qquad \forall 1 \leq i \leq k$$

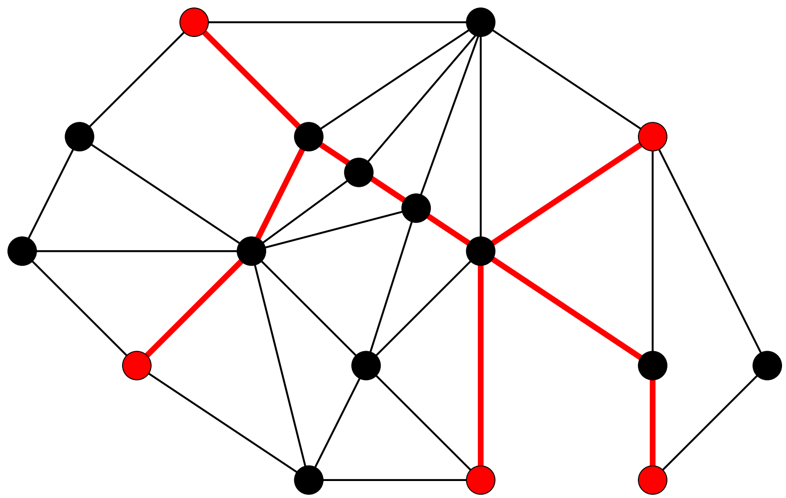
$$x_{\mathcal{P},C} \geq 0 \qquad \forall \mathcal{P}, C$$

Number of variables is $\leq B(k) \cdot k$, where $B(k)$ is the no. of partitions of $[k]$
$\Rightarrow$ The ILP algorithm solves the problem in time $f(k) \cdot n^{O(1)}$.

STEINER TREE

STEINER TREE

Task: Given a graph G with weighted edges and a set S of k vertices, find a tree T of minimum weight that contains S.



Known to be NP-hard. For fixed k, we can solve it in polynomial time: we can guess the Steiner points and the way they are connected.

Theorem: STEINER TREE is FPT parameterized by $k = |S|$.

STEINER TREE

Solution by dynamic programming. For $v \in V(G)$ and $X \subseteq S$,

$c(v, X) :=$ minimum cost of a Steiner tree of X that contains v

$d(u, v) :=$ distance of u and v

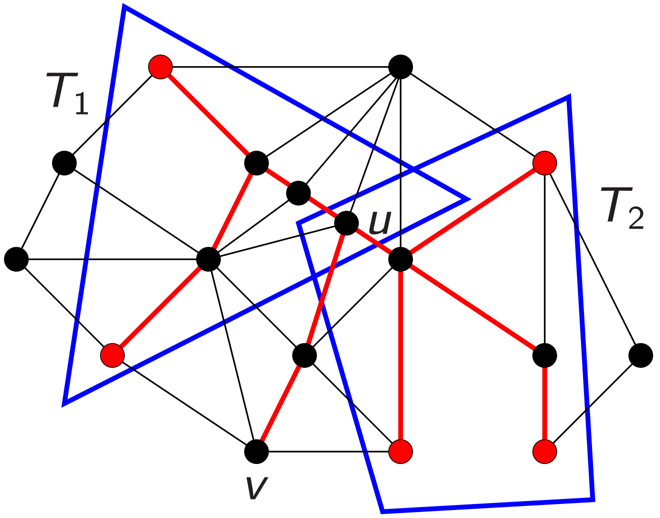
Recurrence relation:

$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus X') \setminus u) + d(u, v)$$

Recurrence relation:

$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus X') \setminus u) + d(u, v)$$

- $\leq$: A tree T_1 realizing $c(u, X' \setminus u)$, a tree T_2 realizing $c(u, (X \setminus X') \setminus u)$, and the path uv gives a (superset of a) Steiner tree of X containing v.

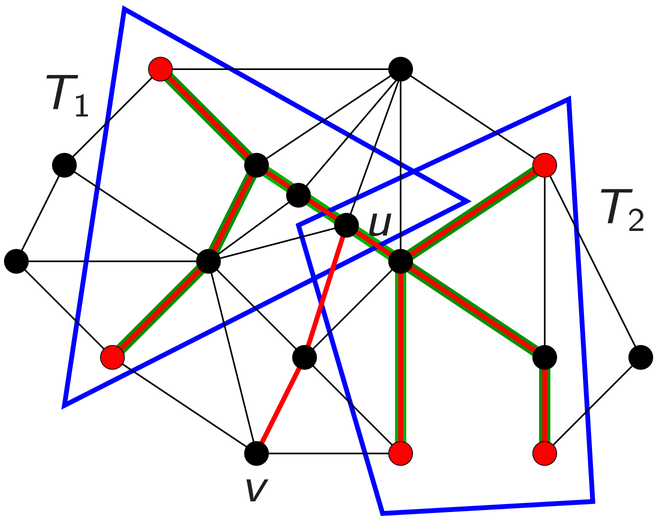


STEINER TREE

Recurrence relation:

$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus X') \setminus u) + d(u, v)$$

- $\geq$: Suppose T realizes $c(v, X)$, let T' be the minimum subtree containing X. Let u be a vertex of T' closest to v. If $|X| > 1$, then there is a component C of $T \setminus u$ that contains a subset $\emptyset \subset X' \subset X$ of terminals. Thus T is the disjoint union of a tree containing $X' \setminus u$ and u, a tree containing $(X \setminus X') \setminus u$ and u, and the path uv.



STEINER TREE

Recurrence relation:

$$c(v,X) = \min_{\substack{u\in V(G)\\ \emptyset\subset X'\subset X}} c(u,X'\setminus u) + c(u,(X\setminus u)\setminus X') + d(u,v)$$

Running time:

$2^k|V(G)|$ variables $c(v,X)$, determine them in increasing order of $|X|$. Variable $c(v,X)$ can be determined by considering $2^{|X|}$ cases. Total number of cases to consider:

$$\sum_{X\subseteq T} 2^{|X|} = \sum_{i=1}^{k} \binom{k}{i} 2^i \le (1+2)^k = 3^k.$$

Running time is $O^*(3^k)$.

Note: Running time can be reduced to $O^*(2^k)$ with clever techniques.

Conclusions

- Many nice techniques invented so far — and probably many more to come.
- A single technique might provide the key for several problems.
- How to find new techniques? By attacking the open problems!
- Next (January):
  - Treewidth
  - Hardness theory