Fixed-Parameter Algorithms

Rolf Niedermeier and Jiong Guo

Lehrstuhl Theoretische Informatik I / Komplexitätstheorie

Institut für Informatik

Friedrich-Schiller-Universität Jena

niedermr@minet.uni-jena.de guo@minet.uni-jena.de

http://theinf1.informatik.uni-jena.de/

Fixed-Parameter Algorithms

Literature:

- M. R. Fellows and R. G. Downey, *Parameterized Complexity*, Springer-Verlag, 1999.
- R. Niedermeier, *Invitation to Fixed-Parameter Algorithms*, Oxford University Press, 2005, almost in press.

Lecture is based on the second book.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

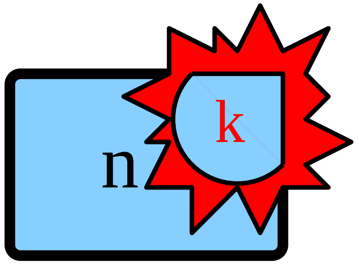
- Introduction to fixed-parameter algorithms
- Parameterized complexity theory—a primer
- VERTEX COVER—an illustrative example
- The art of problem parameterization

❷ Algorithmic methods

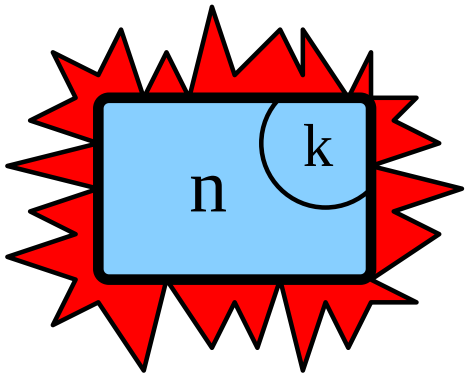
❸ Parameterized complexity theory

Introduction to fixed-parameter algorithms

Given a "combinatorially explosive" (NP-hard) problem with input size n, parameter value k, then the leitmotif is:



instead of



- Guaranteed optimality of the solution $\Uparrow$
- Provable upper bounds on the computational complexity $\Uparrow$
- Exponential running time $\Downarrow$

Introduction to fixed-parameter algorithms

Other approaches:

- Randomized algorithms
- Approximation algorithms
- Heuristics
- Average-case analysis
- New models of computing (DNA or quantum computing, ...)

Case Study 1: CNF-SATISFIABILITY

- Input: A boolean formula F in conjunctive normal form with n boolean variables and m clauses.
- Task: Determine whether or not there exists a truth assignment for the boolean variables in F such that F evaluates to true.

Example

$$(x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

Application: VLSI design, model checking, ...

Case Study 1: CNF-SATISFIABILITY

Parameter "clause size". 2-CNF-SATISFIABILITY is polynomial-time solvable; 3-CNF-SATISFIABILITY is NP-complete.

Parameter "number of variables". Solvable in 2^n steps.

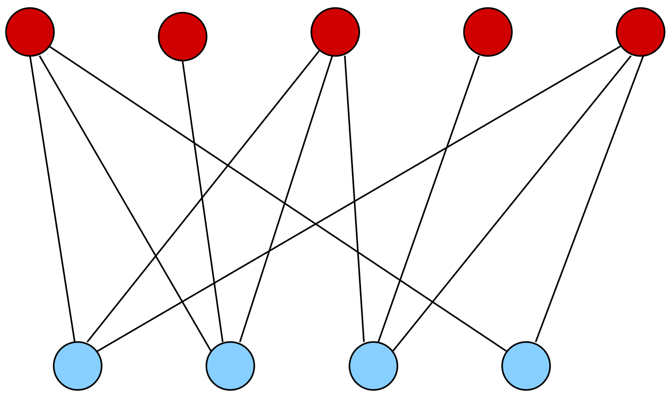
Parameter "number of clauses". Solvable in 1.24^m steps.

Parameter "formula length". Solvable in 1.08^ℓ steps where ℓ denotes the total length (that is, the number of literal occurrences in the formula) of F.

Case Study 2: Dominating Set in Bipartite Graphs

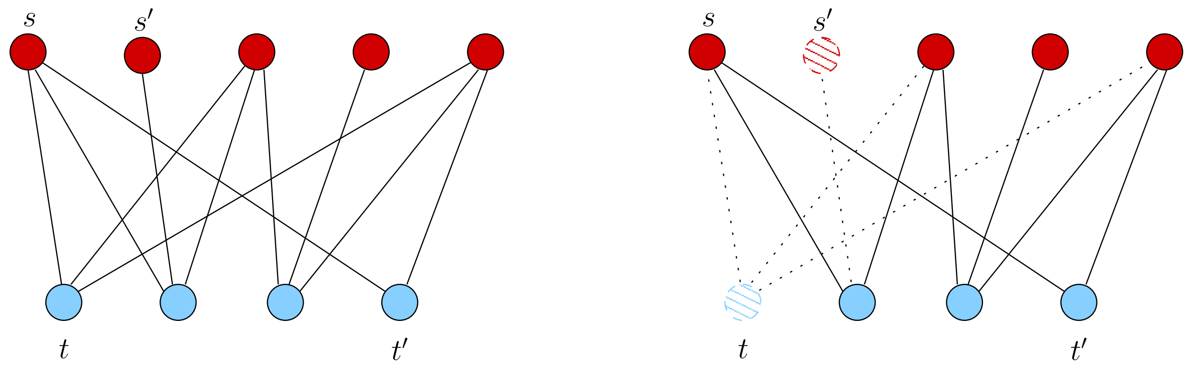
- Input: An undirected, bipartite graph G with disjoint vertex sets V_1 and V_2 and edge set E.
- Task: Find a minimum size set $S \subseteq V_2$ such that each vertex in V_1 has an adjacent edge connecting it to some vertex in S.

Example Railway optimization



Case Study 2: DOMINATING SET IN BIPARTITE GRAPHS

Example



Train Rule. For $t, t' \in V_1$: If $N(t') \subseteq N(t)$, then remove t.

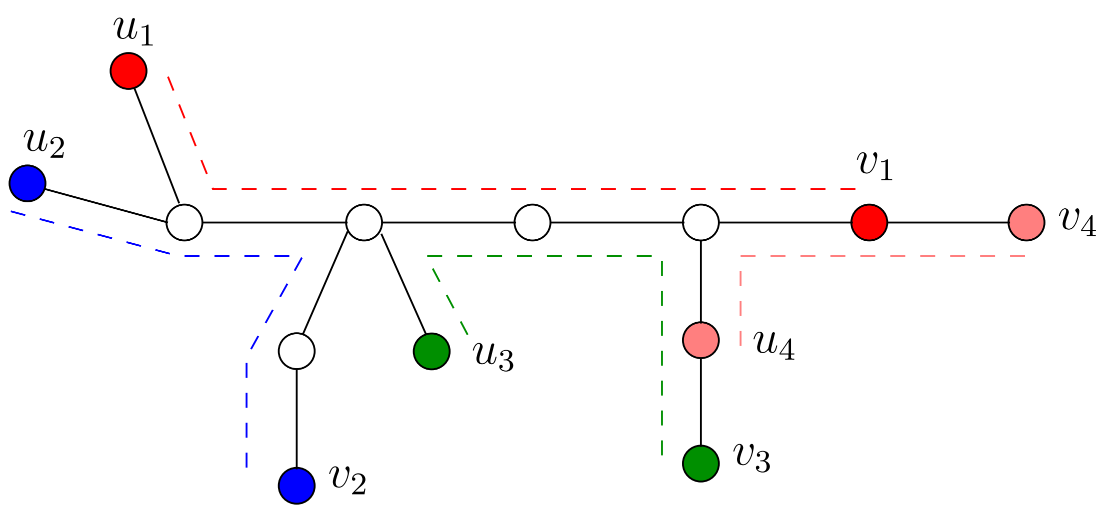
Station Rule. For $s, s' \in V_2$: if $N(s') \subseteq N(s)$, then remove s'.

Case Study 3: MULTICUT IN TREES

- Input: An undirected tree $T = (V, E)$, $n := |V|$, and $H = \{(u_i, v_i) \mid u_i, v_i \in V, u_i \neq v_i, 1 \leq i \leq m\}$.

- Task: Find a minimum size set $E' \subseteq E$ such that the removal of the edges in E' separates each pair of nodes in H.

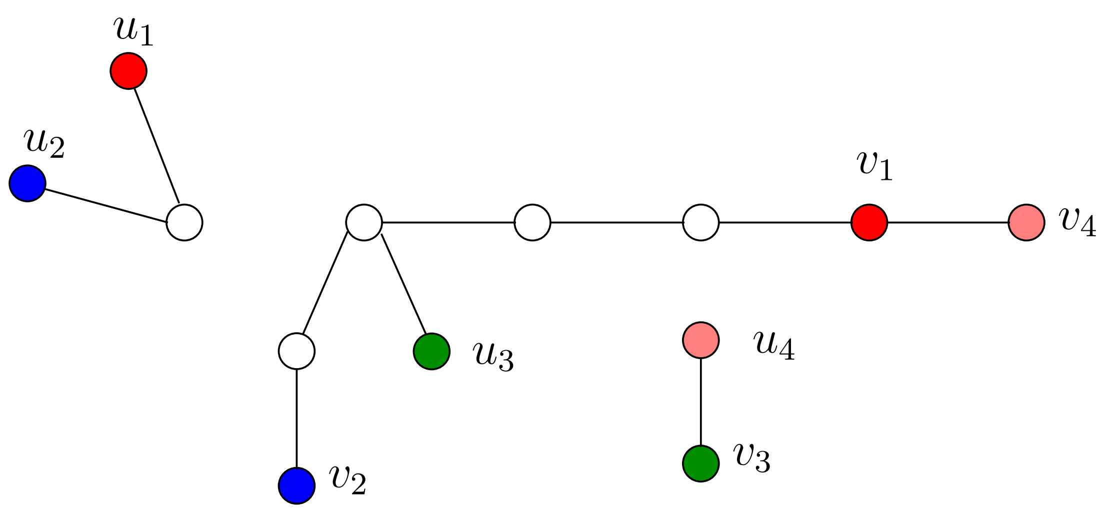
Example



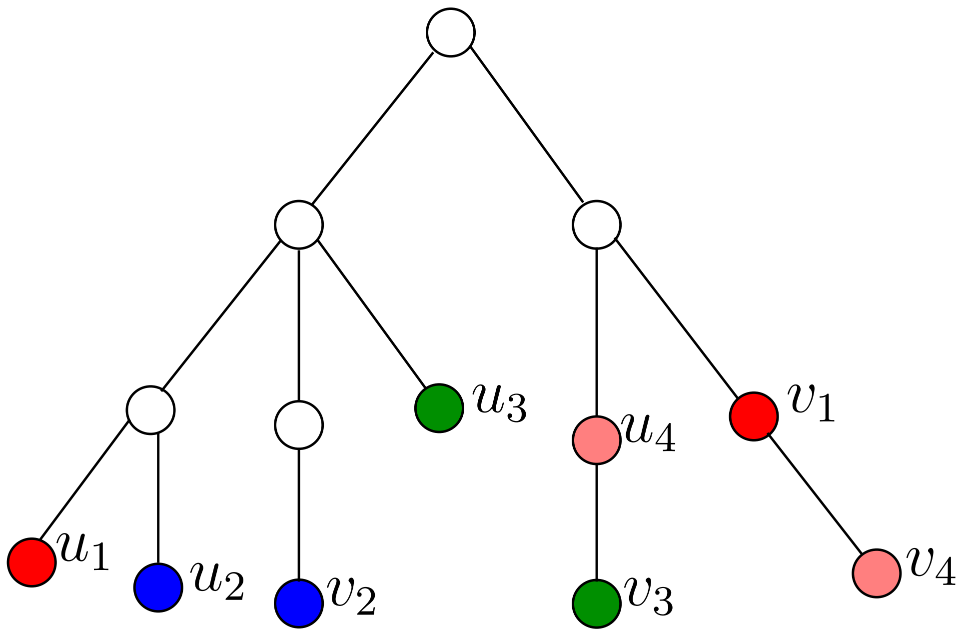
Case Study 3: Multicut in Trees

- Input: An undirected tree $T = (V, E)$, $n := |V|$, and $H = \{(u_i, v_i) \mid u_i, v_i \in V, u_i \neq v_i, 1 \leq i \leq m\}$.
- Task: Find a minimum size set $E' \subseteq E$ such that the removal of the edges in E' separates each pair of nodes in H.

Example



Case Study 3: MULTICUT IN TREES



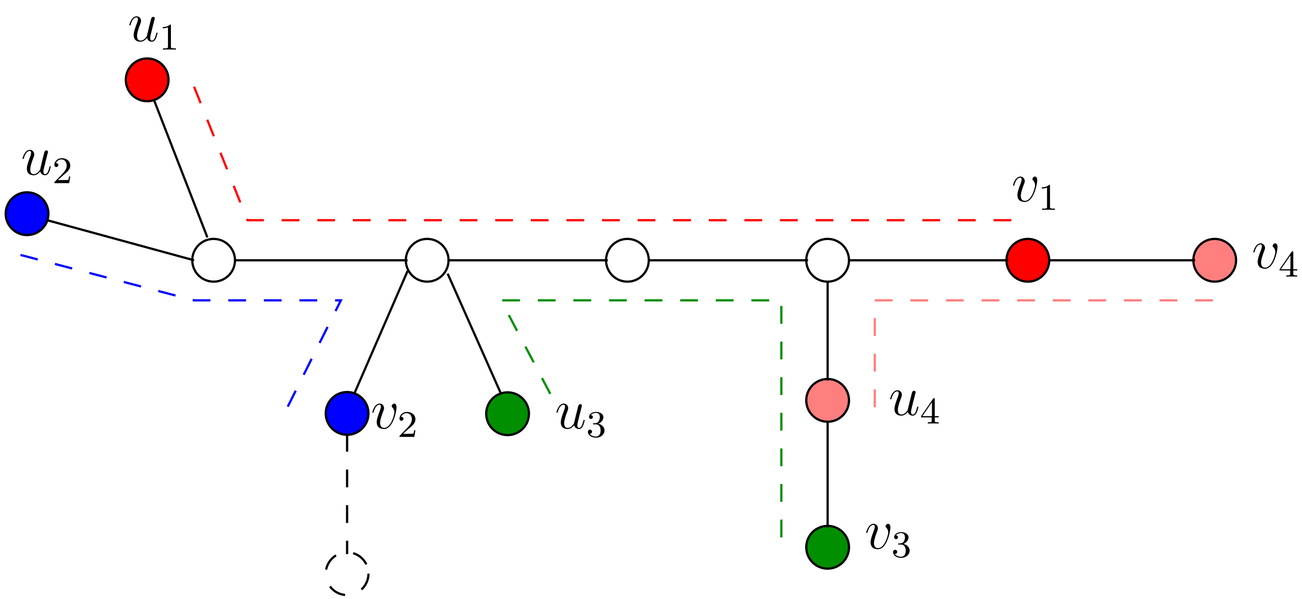
Idea: *Bottom-up* $+$ *Least common ancestor* ...

Search tree: 2^k ($k :=$ the number of edge deletions).

Case Study 3: MULTICUT IN TREES

Data reduction rules:

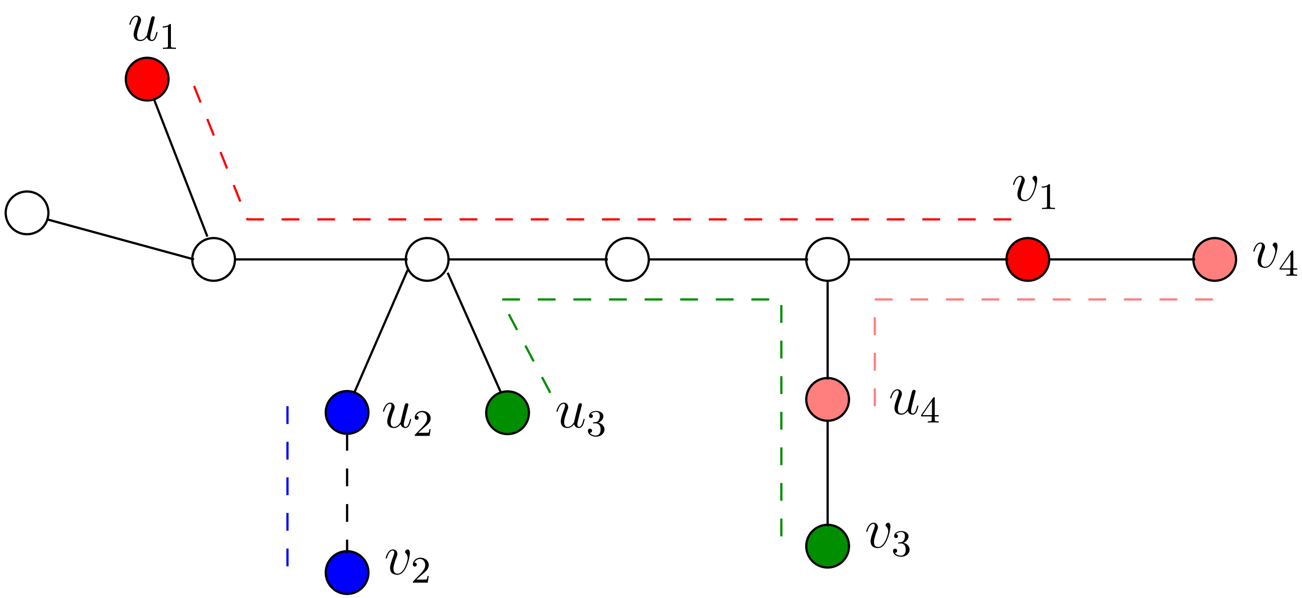
Idle edge



Case Study 3: Multicut in Trees

Data reduction rules:

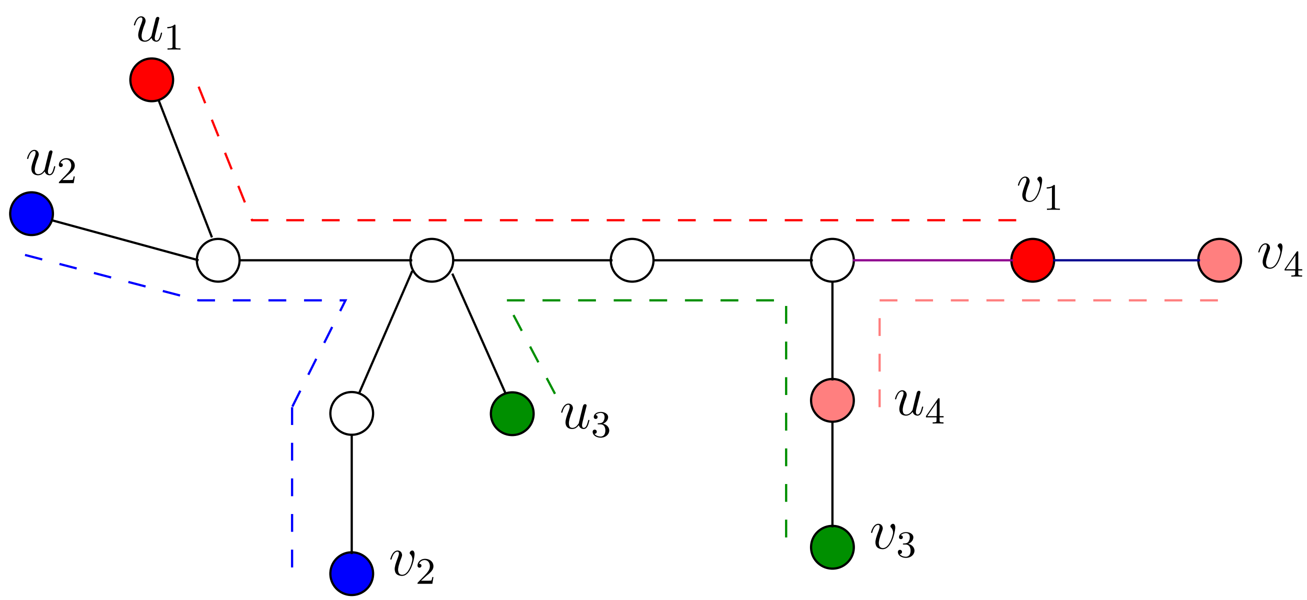
Unit path



Case Study 3: MULTICUT IN TREES

Data reduction rules:

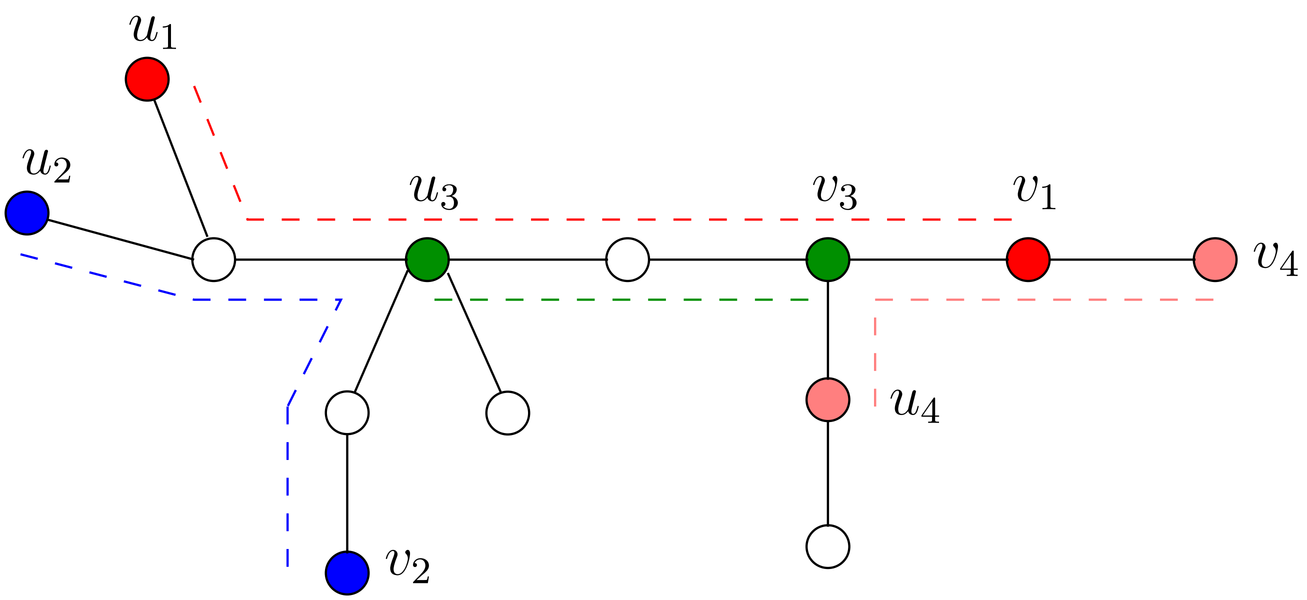
Dominated edge



Case Study 3: Multicut in Trees

Data reduction rules:

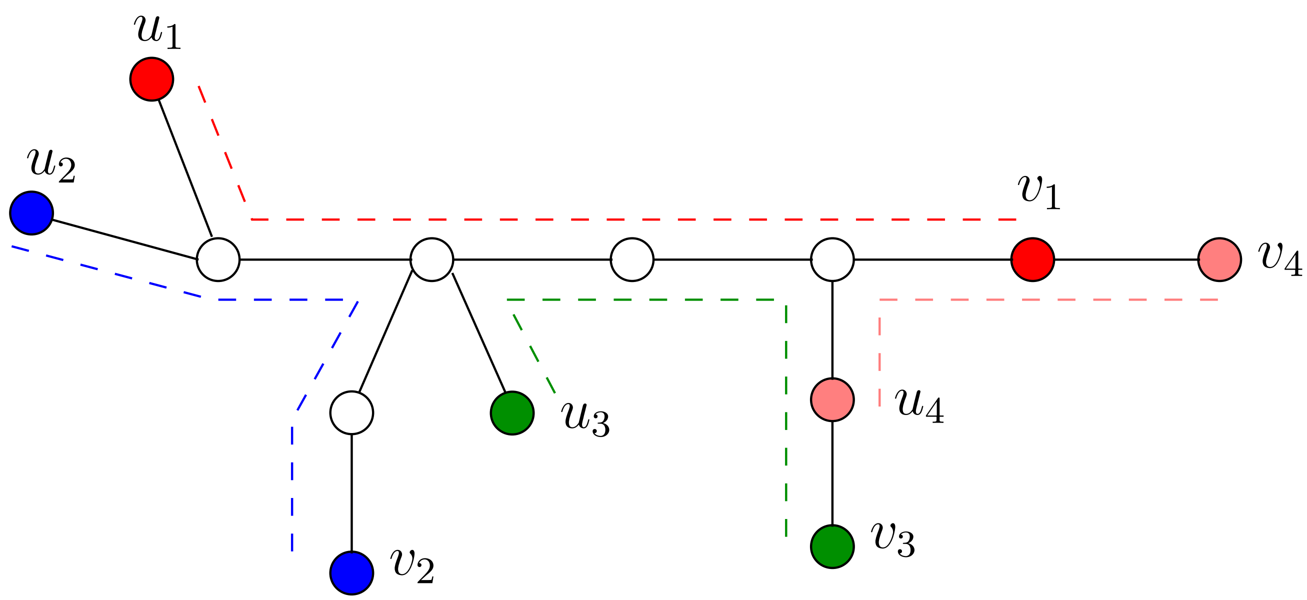
Dominated path



Case Study 3: MULTICUT IN TREES

Data reduction rules:

Disjoint paths



Case Study 3: MULTICUT IN TREES

Fundamental observation:

Preprocess the “raw input” using data reduction rules in order to simplify and shrink it.

Advantage:

Only the “really hard” “problem kernel” remains ...

Fixed-Parameter Algorithms

1. Basic ideas and foundations
   - Introduction to fixed-parameter algorithms
   - Parameterized complexity theory—a primer
   - VERTEX COVER—an illustrative example
   - The art of problem parameterization
2. Algorithmic methods
3. Parameterized complexity theory

Parameterized complexity theory—a primer

Definition

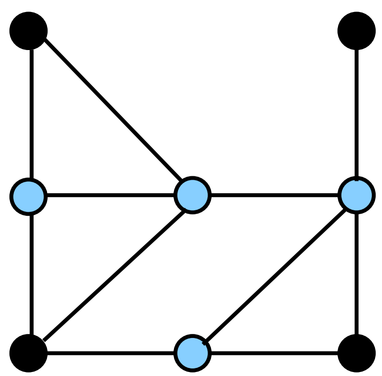
A *parameterized* problem is a language $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The second component is called the *parameter* of the problem.

Definition

A parameterized problem is *fixed-parameter tractable* if it can be determined in $f(k) \cdot |x|^{O(1)}$ time whether $(x, k) \in L$, where f is a computable function only depending on k. The corresponding complexity class is called *FPT*.

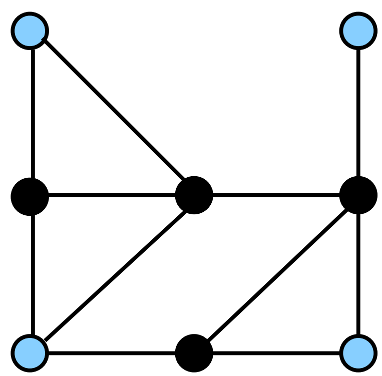
Parameterized complexity theory—a primer

$$\text{FPT} \subseteq \overbrace{\text{W[1]} \subseteq \text{W[2]} \subseteq \ldots \subseteq \text{W[P]}}^{\text{Presumably fixed-parameter intractable}}$$



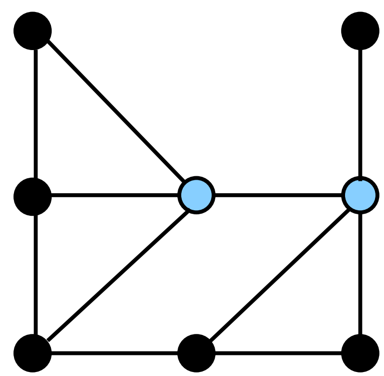
VERTEX COVER

FPT



INDEPENDENT SET

"W[1]–complete"



DOMINATING SET

"W[2]–complete"

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

- Introduction to fixed-parameter algorithms
- Parameterized complexity theory—a primer
- VERTEX COVER—an illustrative example
- The art of problem parameterization

❷ Algorithmic methods

❸ Parameterized complexity theory

VERTEX COVER—an illustrative example

- ☞ Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- ☞ Task: Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

Solution methods:

- ✘ Search tree: combinatorial explosion smaller than 1.28^k.
- ✘ Data reduction by preprocessing: techniques by Buss, Nemhauser-Trotter.

VERTEX COVER—an illustrative example

- Parameterizing
- Specializing
- Generalizing
- Counting or enumerating
- Lower bounds
- Implementing and applying
- Using VERTEX COVER structure for other problems

VERTEX COVER—an illustrative example

- Parameterizing
  - Size of the vertex cover set;
  - Dual parameterization: INDEPENDENT SET;
  - Parameterizing above guaranteed values: planar graphs;
  - Structure of the input graph: treewidth $\omega \leadsto$ combinatorial explosion 2^ω.

Vertex Cover—an illustrative example

- Parameterizing
- Specializing: special graph classes—planar graphs $O(c^{\sqrt{k}} + kn)$.
- Generalizing
- Counting or enumerating
- Lower bounds
- Implementing and applying
- Using Vertex Cover structure for other problems

Vertex Cover—an illustrative example

- Parameterizing
- Specializing
- Generalizing: Weighted Vertex Cover, Capacitated Vertex Cover, Hitting Set, ...
- Counting or enumerating
- Lower bounds
- Implementing and applying
- Using Vertex Cover structure for other problems

VERTEX COVER—an illustrative example

- Parameterizing
- Specializing
- Generalizing
- Counting or enumerating:
  - Counting: combinatorial explosion $O(1.47^k)$.
  - Enumerating: combinatorial explosion $O(2^k)$.

VERTEX COVER—an illustrative example

- Parameterizing
- Specializing
- Generalizing
- Counting or enumerating
- Lower bounds: widely open!
- Implementing and applying
- Using VERTEX COVER structure for other problems

VERTEX COVER—an illustrative example

- Parameterizing
- Specializing
- Generalizing
- Counting or enumerating
- Lower bounds
- Implementing and applying: re-engineering case distinctions, parallelization, ...
- Using VERTEX COVER structure for other problems

VERTEX COVER—an illustrative example

- Parameterizing
- Specializing
- Generalizing
- Counting or enumerating
- Lower bounds
- Implementing and applying
- Using VERTEX COVER structure for other problems: solve related problems using an optimal VERTEX COVER solution.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

- Introduction to fixed-parameter algorithms
- Parameterized complexity theory—a primer
- VERTEX COVER—an illustrative example
- The art of problem parameterization

❷ Algorithmic methods

❸ Parameterized complexity theory

The art of problem parameterization

- Parameter really small?
- Guaranteed parameter value?
- More than one obvious parameterization?
- Close to “trivial” problem instances?

Fixed-Parameter Algorithms

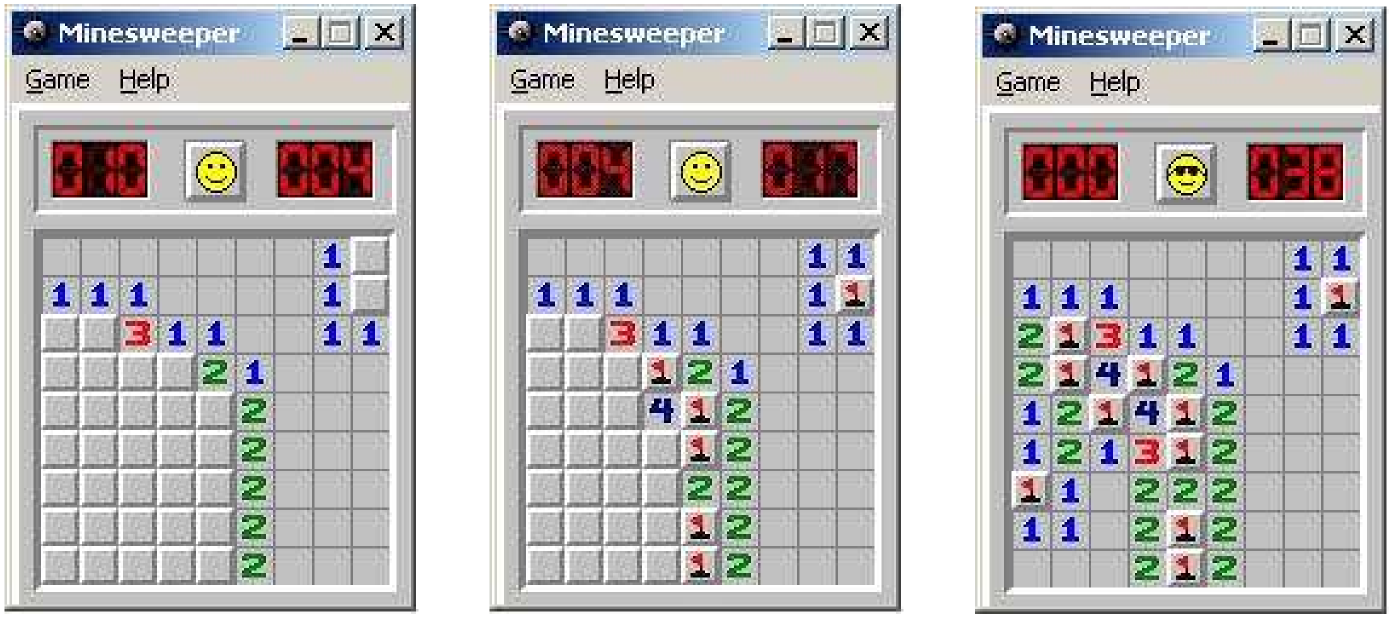
❶ Basic ideas and foundations

❷ Algorithmic methods

- ➩ Data reduction and problem kernels
- ➩ Depth-bounded search trees
- ➩ Some advanced techniques
- ➩ Dynamic programming
- ➩ Tree decompositions of graphs

❸ Parameterized complexity theory

Data reduction and problem kernels



Data reduction and problem kernels

VERTEX COVER

- Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- Task: Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

Buss' reduction to a problem kernel: If a vertex has more than k adjacent edges, this particular vertex has to be part of every vertex cover of size at most k.

Data reduction and problem kernels

VERTEX COVER: Buss' reduction to a problem kernel

- All vertices with more than k neighbors are added to the vertex cover.
- In the resulting graph each vertex can have at most k neighbors. Then if the remaining graph has a vertex cover of size k, then it contains at most $k^2 + k$ vertices and at most k^2 edges.

Data reduction and problem kernels

Definition

Let L be a parameterized problem, that is, L consists of (I, k), where I is the problem instance and k is the parameter. *Reduction to a problem kernel* then means to replace instance (I, k) by a "reduced" instance (I', k') (called *problem kernel*) such that

1. $k' \leq k,\ |I'| \leq g(k)$ for some function g only depending on k,
2. $(I, k) \in L$ iff $(I', k') \in L$, and
3. the reduction from (I, k) to (I', k') has to be computable in polynomial time.

Data reduction and problem kernels

MAXIMUM SATISFIABILITY

- Input: A boolean formula F in conjunctive normal form consisting of m clauses and a nonnegative integer k.
- Task: Find a truth assignment satisfying at least k clauses.

Example

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

Data reduction and problem kernels

MAXIMUM SATISFIABILITY

Observation 1:

If $k \leq \lceil m/2 \rceil$, then the desired truth assignment trivially exists: Take a random truth assignment. If it satisfies at least k clauses then we are done. Otherwise, "flipping" each bit in this truth assignment to its opposite value yields a new truth assignment that now satisfies at $\lceil m/2 \rceil$ clauses.

Data reduction and problem kernels

MAXIMUM SATISFIABILITY

Partition the clauses of F into F_l and F_s:

- F_l: long clauses containing at least k literals;
- F_s: short clauses containing less than k literals.

Let $L :=$ number of long clauses.

Observation 2:

If $L \geq k$, then at least k clauses can be satisfied.

Data reduction and problem kernels

MAXIMUM SATISFIABILITY

Observation 3:

(F, k) is a yes-instance iff $(F_s, k - L)$ is a yes-instance.

Theorem

MAXIMUM SATISFIABILITY has a problem kernel of size $O(k^2)$, and it can be found in linear time.

Data reduction and problem kernels: 3-HITTING SET

☞ Input: A collection $\mathcal{C}$ of subsets of size at most three of a finite set S and a nonnegative integer k.

☞ Task: Find a subset $H \subseteq S$ with $|H| \leq k$ such that H contains at least one element from each subset in $\mathcal{C}$.

$\boxed{\text{Example}}$ $S := \{s_1, s_2, \ldots, s_9\}, k = 3.$

$$\mathcal{C} := \{ \quad \{s_1, s_2, s_3\}, \quad \{s_1, s_4, s_7\}, \quad \{s_2, s_4, s_5\}, \quad \{s_3, s_5, s_7\}, \quad \{s_5, s_6, s_8\}, \quad \{s_6, s_9\} \quad \}$$

Data reduction and problem kernels: 3-HITTING SET

Example $S := \{s_1, s_2, \ldots, s_9\}, k = 3.$

$\mathcal{C} := \{$

$\{s_1, s_2, s_3\},$

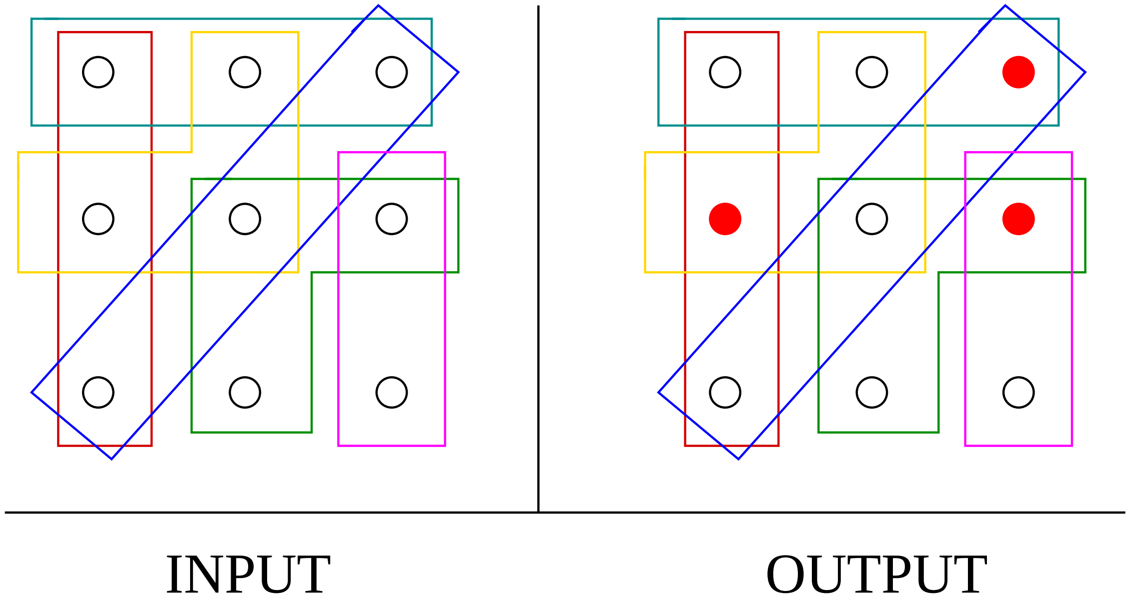
$\{s_1, s_4, s_7\},$

$\{s_2, s_4, s_5\},$

$\{s_3, s_5, s_7\},$

$\{s_5, s_6, s_8\},$

$\{s_6, s_9\}\ \}$



$$H = \{s_3, s_4, s_6\}.$$

Data reduction and problem kernels: 3-HITTING SET

Data reduction rule 1 For every pair of subsets $C_i, C_j \in \mathcal{C}$:
If $C_i \subseteq C_j$, then remove C_j from $\mathcal{C}$.

Data reduction rule 2 For every pair of elements $s_i, s_j \in S$ with $i < j$:
If there are more than k size-three subsets in $\mathcal{C}$ that contain both s_i and s_j, then remove all these size-three subsets from $\mathcal{C}$ and add subset $\{s_i, s_j\}$ to $\mathcal{C}$.

Data reduction and problem kernels: 3-HITTING SET

Data reduction rule 3 For every element $s \in S$:

If there are more than k^2 size-three or more than k size-two subsets in $\mathcal{C}$ that contain s, then remove all these subsets from $\mathcal{C}$, add s to H, and $k := k - 1$.

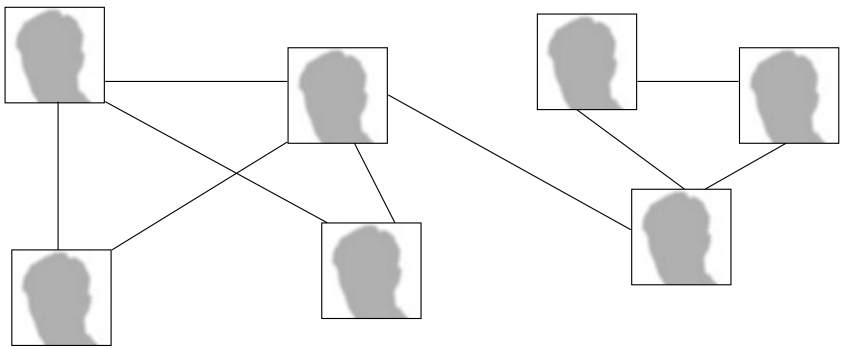
Theorem

3-HITTING SET has a problem kernel with $|\mathcal{C}| = O(k^3)$. It can be found in $O(|S| + |\mathcal{C}|)$ time.

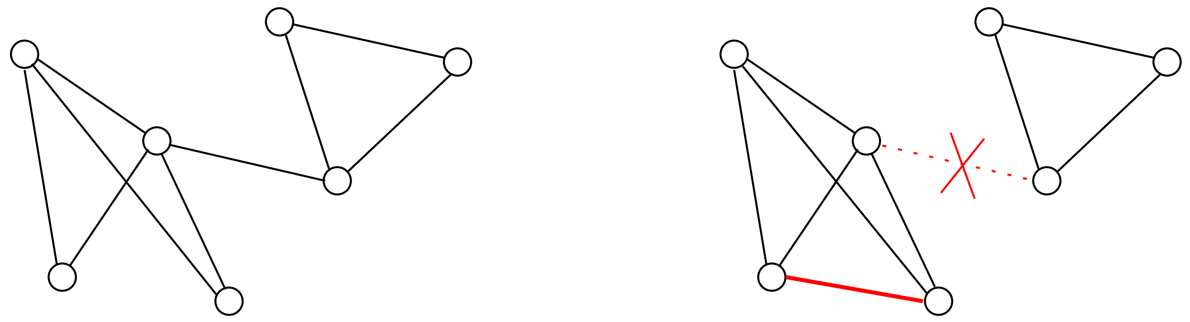
Data reduction and problem kernels: CLUSTER EDITING

- Input: A graph G and a nonnegative integer k.
- Task: Find out whether we can transform G, by deleting or adding at most k edges, into a graph that consists of a disjoint union of cliques.

Applications: Machine Learning, Clustering gene expression data



Data reduction and problem kernels: CLUSTER EDITING



Define Table T which has an entry for every pair of vertices $u, v \in V$.

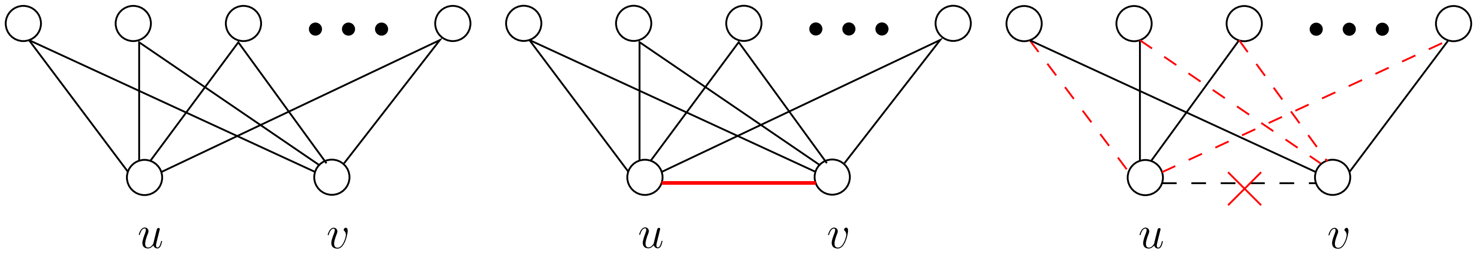
This entry is either empty or takes one of the following two values:

- "permanent": $\{u, v\} \in E$ and it is not allowed to delete $\{u, v\}$;
- "forbidden": $\{u, v\} \notin E$ and it is not allowed to add $\{u, v\}$;

Data reduction and problem kernels: CLUSTER EDITING

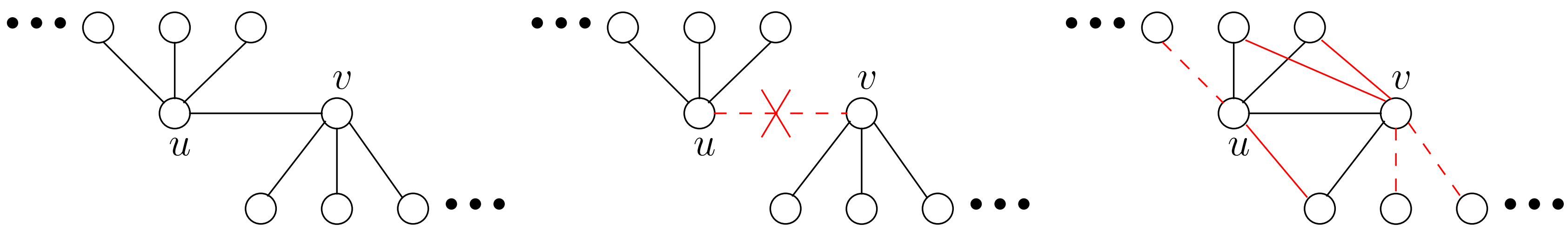
Data reduction rule 1 For every pair of vertices $u, v \in V$:

(1) If u and v have more than k common neighbors, then $\{u, v\}$ must be in E and we set $T[u, v] :=$ permanent. If $\{u, v\} \notin E$, we add it to E.



Data reduction rule 1 For every pair of vertices $u, v \in V$:

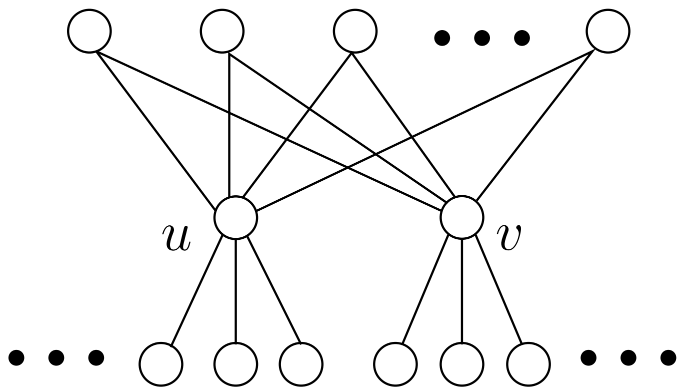
(2) If u and v have more than k non-common neighbors, then $\{u, v\}$ must not be in E and we set $T[u, v] :=$ forbidden. If $\{u, v\} \in E$, we delete it.



Data reduction and problem kernels: CLUSTER EDITING

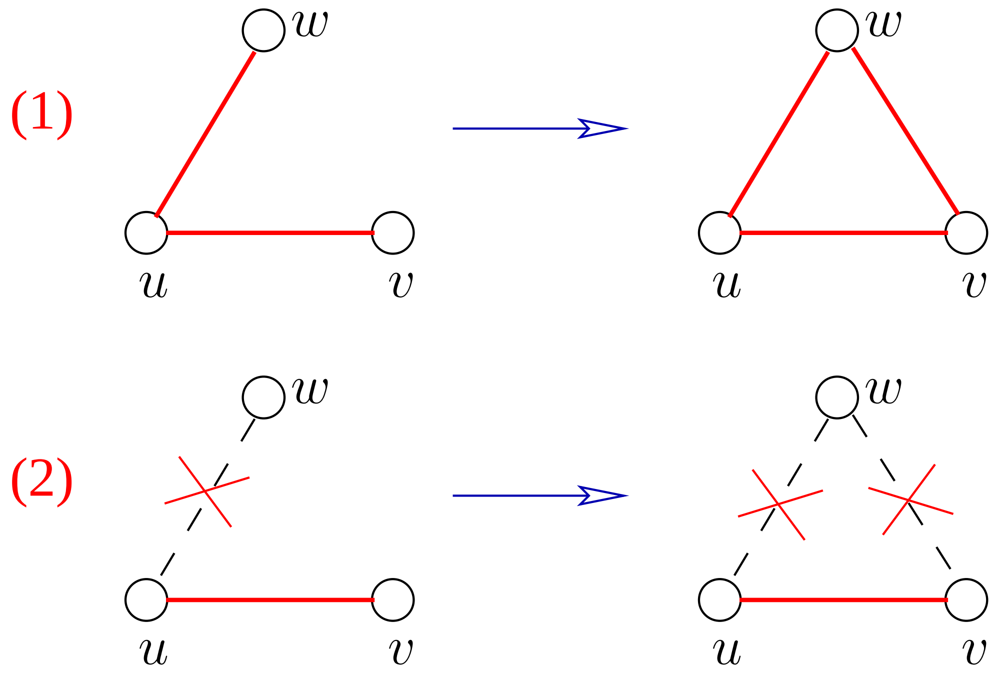
Data reduction rule 1 For every pair of vertices $u, v \in V$:

(3) If u and v have both more than k common and more than k non-common neighbors, then the given instance has no solution.



Data reduction and problem kernels: CLUSTER EDITING

Data reduction rule 2 For every three vertices $u, v, w \in V$:



Data reduction and problem kernels: CLUSTER EDITING

Data reduction rule 3

Delete connected components which are cliques.

Theorem

CLUSTER EDITING has a problem kernel with a graph that contains at most $O(k^2)$ vertices and $O(k^3)$ edges. It can be found in $O(n^3)$ time.

Data reduction and problem kernels: Vertex Cover

- Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- Task: Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

Buss' reduction leads to a size-$O(k^2)$ problem kernel.

Improved bound on the kernel size: $|V| \leq 2k$.

VERTEX COVER: Kernelization Based on Matching

Kernelization based on matching:

- Input: An undirected graph $G = (V, E)$.

  Step (1): Construct a bipartite graph $B = (V, V', E_B)$
  where $E_B := \{\{x, y'\}, \{x', y\} \mid \{x, y\} \in E\}$
  and $V' := \{x' \mid x \in V\}$.

  Step (2): Compute an optimal vertex cover C_B of B.

- Output: $C_0 := \{x \mid x \in C_B$ and $x' \in C_B\}$,
  $V_0 := \{x \mid$ either $x \in C_B$ or $x' \in C_B\}$, and
  $I_0 := V \setminus (V_0 \cup C_0)$.

VERTEX COVER: Kernelization Based on Matching

How to do Step (2)?

- An optimal vertex cover of a bipartite graph can be determined by computing a maximum matching using standard methods in $O(\sqrt{n} \cdot m)$ time.
- A maximum matching is a maximum cardinality set of edges in a graph such that no two edges in this set share an endpoint.
- For bipartite graphs the size of a maximum matching coincides with the size of a minimum vertex cover of the graph (König, 1931).

VERTEX COVER: Kernelization Based on Matching

☞ Input: An undirected graph $G = (V, E)$.

☞ Output: $C_0 := \{x \mid x \in C_B \text{ and } x' \in C_B\}$,
$V_0 := \{x \mid \text{ either } x \in C_B \text{ or } x' \in C_B\}$, and
$I_0 := V \setminus (V_0 \cup C_0)$.

NT–Theorem [Nemhauser and Trotter]

1. If $D \subseteq V_0$ is a vertex cover of the induced graph $G[V_0]$, then $C := C_0 \cup D$ is a vertex cover of G.
2. There is a minimum vertex cover of G which comprises C_0.
3. The induced subgraph $G[V_0]$ has a minimum vertex cover of size at least $|V_0|/2$.

VERTEX COVER: Kernelization Based on Matching

Application of NT–Theorem:

Theorem

Let $(G = (V, E), k)$ be an input instance of VERTEX COVER. In $O(k \cdot |V| + k^3)$ time we can compute a reduced instance $(G' = (V', E'), k')$ with $|V'| \leq 2k$ and $k' \leq k$ such that G admits a vertex cover of size k iff G' admits a vertex cover of size k'.

Proof: Blackboard.

VERTEX COVER: Kernelization Based on Matching

Few remarks:

- ✗ It is hard to improve the $2k$ bound.
- ✗ NT–Theorem can be generalized to find minimum *weighted* vertex covers for positive real-valued vertex weights.
- ✗ NT–Theorem makes no use of the parameter value k.

VERTEX COVER: Kernelization Based on LP

An alternative route to a $2k$-vertices problem kernel: state the optimization version of VERTEX COVER as an integer linear program.

Integer Linear Program (ILP) for VERTEX COVER

$$\begin{array}{rl} \text{Minimize} & \sum_{v \in V} x_v \\ \text{subject to} & x_u + x_v \geq 1, \quad \forall e = \{u, v\} \in E \\ & x_v \in \{0, 1\}, \quad \forall v \in V \end{array}$$

- $x_v = 1$: v is in the vertex cover;
- $x_v = 0$: v is not in the vertex cover.

VERTEX COVER: Kernelization Based on LP

Since integer linear programming is generally intractable (the corresponding decision problem is NP-complete), we relax the integer programming formulation to polynomial-time solvable linear programming.

LP-relaxation

$$\begin{array}{rl} \text{Minimize} & \sum_{v \in V} x_v \\ \text{subject to} & x_u + x_v \geq 1, \quad \forall e = \{u, v\} \in E \\ & 0 \leq x_v \leq 1, \quad \forall v \in V \end{array}$$

VERTEX COVER: Kernelization Based on LP

Minimize $\sum_{v \in V} x_v$

subject to

$$x_u + x_v \geq 1, \quad \forall \{u, v\} \in E$$

$$0 \leq x_v \leq 1, \quad \forall v \in V$$

$$C_0 := \{v \in V \mid x_v > 0.5\}$$

$$V_0 := \{v \in V \mid x_v = 0.5\}$$

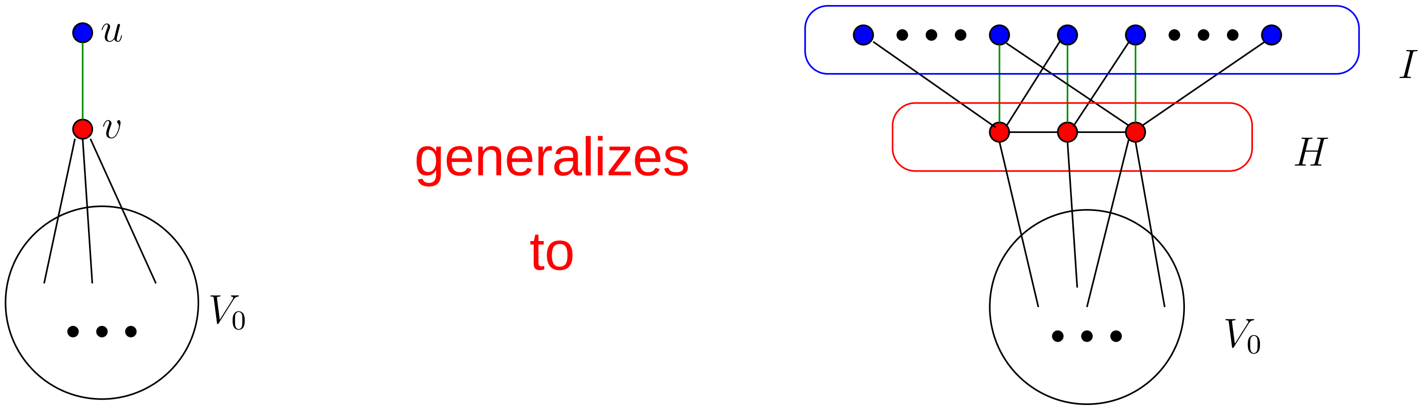
$$I_0 := \{v \in V \mid x_v < 0.5\}$$

Theorem Let $(G = (V, E), k)$ be a VERTEX COVER instance.

1. There is a minimum-size vertex cover S with $C_0 \subseteq S$ and $S \cap I_0 = \emptyset$.

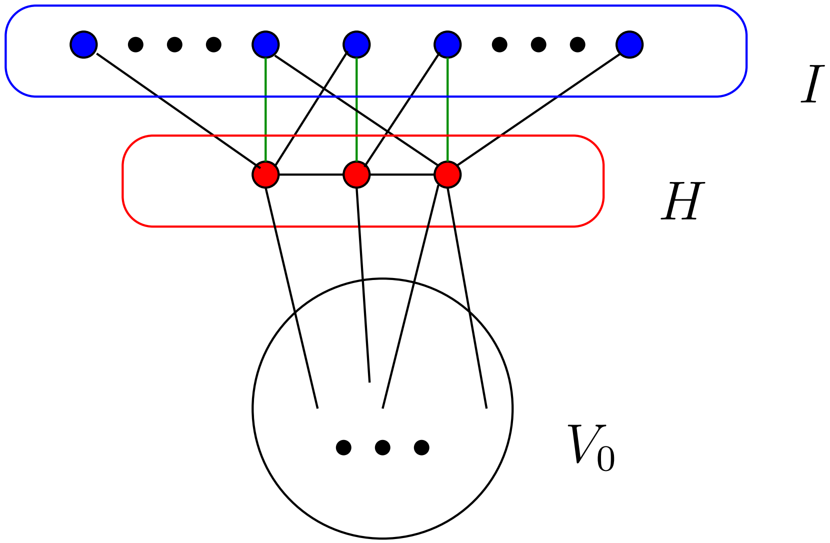
2. V_0 induces a problem kernel $(G[V_0], k - |C_0|)$ with $|V_0| \leq 2k$.

VERTEX COVER: Kernelization Based on Crown Structures



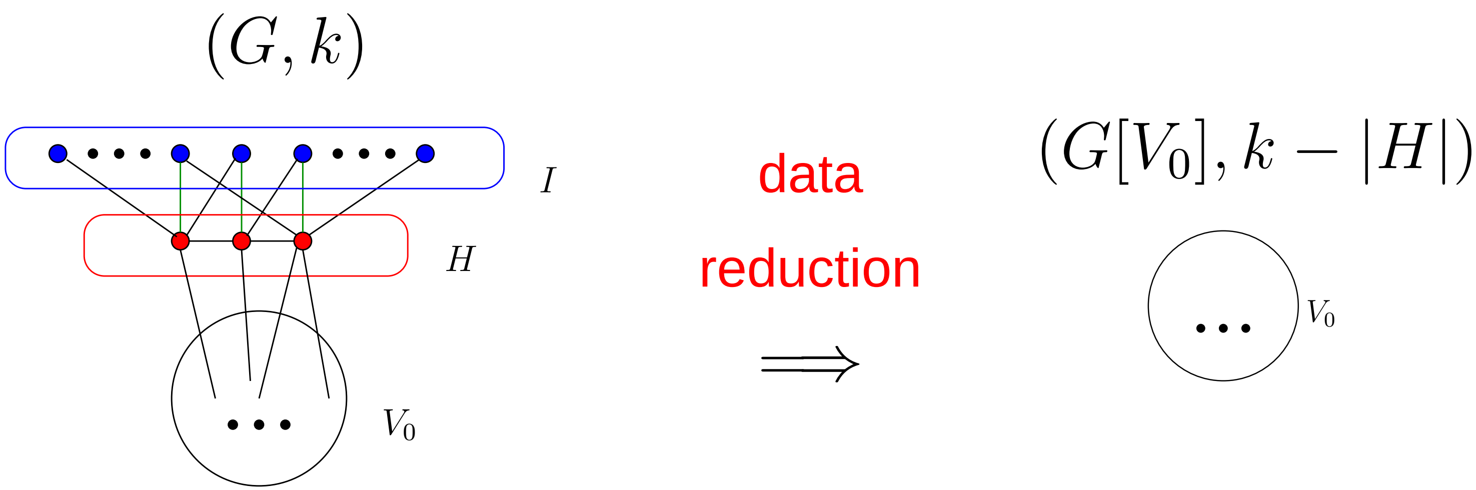
Definition A crown of a graph $G = (V, E)$ consists of $H \subseteq V$ and $I \subseteq V$ with $H \cap I = \emptyset$ such that (1) $H = N(I)$, (2) I forms an independent set, and (3) the edges connecting H and I contain a matching in which all elements of H are matched.

VERTEX COVER: Kernelization Based on Crown Structures



Lemma If G is a graph with a crown H and I, then there exists a minimum-size vertex cover of G that contains all of H and none of I.

VERTEX COVER: Kernelization Based on Crown Structures



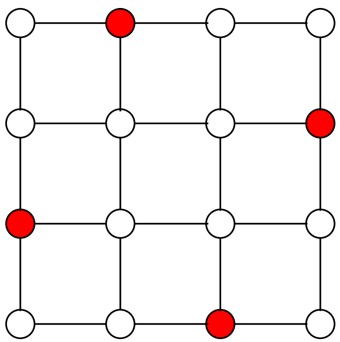
It is possible to show that, if G has a vertex cover with at most k vertices, then we can in polynomial time find a crown H and I such that the reduced instance $(G[V_0], k - |H|)$ with $V_0 := V \setminus (I \cup H)$ is a problem kernel with $|V_0| \leq 3k$.

Data reduction and problem kernels: DOMINATING SET

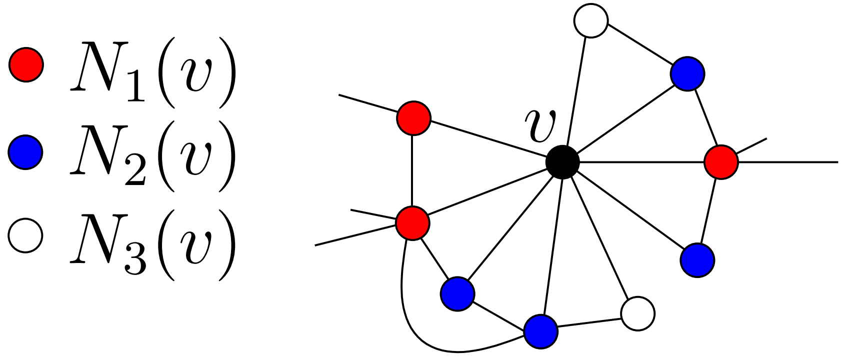
DOMINATING SET IN PLANAR GRAPHS

- Input: A planar graph $G = (V, E)$ and a nonnegative integer k.
- Task: Find a subset $S \subseteq V$ with at most k vertices such that every vertex $v \in V$ is contained in S or has at least one neighbor in S.

Example (4×4)-grid



DOMINATING SET IN PLANAR GRAPHS



Let $N(v)$ denote the open neighborhood of v and $N[v] := N(v) \cup \{v\}$. Define

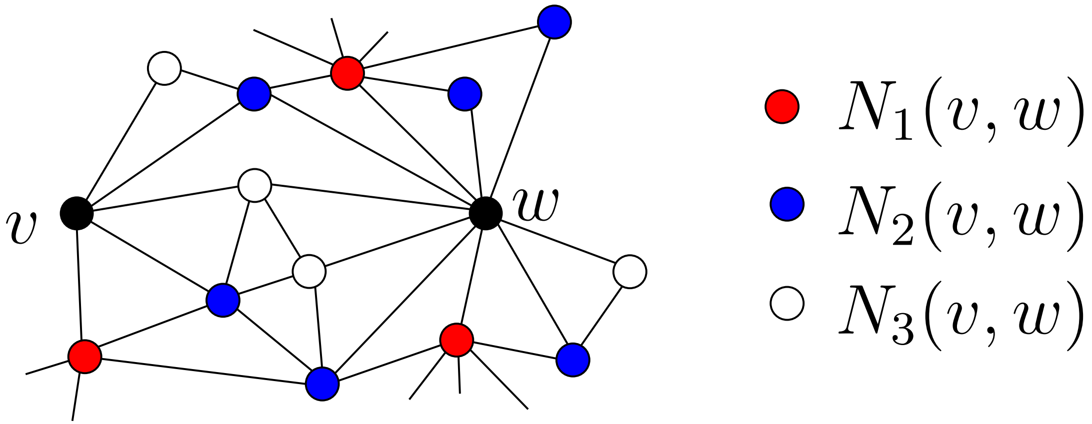
$$\begin{aligned}
N_1(v) &:= \{u \in N(v) \mid N(u) \setminus N[v] \neq \emptyset\} \text{ (exits)}, \\
N_2(v) &:= \{u \in (N(v) \setminus N_1(v)) \mid N(u) \cap N_1(v) \neq \emptyset\} \text{ (guards)}, \\
N_3(v) &:= N(v) \setminus (N_1(v) \cup N_2(v)) \text{ (prisoners)}.
\end{aligned}$$

DOMINATING SET IN PLANAR GRAPHS

Rule 1 If $N_3(v) \neq \emptyset$ for some vertex v, then remove $N_2(v) \cup N_3(v)$ from G and add a new vertex v' with the edge $\{v, v'\}$ to G.

Lemma G has a dominating set with k vertices iff G' has a dominating set with k vertices. Rule 1 can be carried out in $O(|V|)$ time for planar graphs and in $O(|V|^3)$ time for general graphs.

DOMINATING SET IN PLANAR GRAPHS



Let $N(v,w) := N(v) \cup N(w)$ and $N[v,w] := N[v] \cup N[w]$.
Define

$$
\begin{aligned}
N_1(v,w) &:= \{u \in N(v,w) \mid N(u) \setminus N[v,w] \neq \emptyset\} \text{ (exits)} , \\
N_2(v,w) &:= \{u \in (N(v,w) \setminus N_1(v,w)) \mid N(u) \cap N_1(v,w) \neq \emptyset\} \text{ (guards)} , \\
N_3(v,w) &:= N(v,w) \setminus (N_1(v,w) \cup N_2(v,w)) \text{ (prisoners)} .
\end{aligned}
$$

DOMINATING SET IN PLANAR GRAPHS

$\boxed{\text{Rule 2}}$ Consider $v, w \in V$ ($v \neq w$). Suppose that $N_3(v, w) \neq \emptyset$ and $N_3(v, w)$ cannot be dominated by a single vertex from $N_2(v, w) \cup N_3(v, w)$.

Case 1. $N_3(v, w)$ can be dominated by a single vertex from $\{v, w\}$.

Case 2. $N_3(v, w)$ cannot be dominated by a single vertex from $\{v, w\}$.

DOMINATING SET IN PLANAR GRAPHS

Case 1.1. If $N_3(v,w) \subseteq N(v)$ as well as $N_3(v,w) \subseteq N(v)$,

- remove $N_3(v,w)$ and $N_2(v,w) \cap N(v) \cap N(w)$ from G and
- add two new vertices z, z' and
  edges $\{v,z\}, \{w,z\}, \{v,z'\}, \{w,z'\}$ to G.

Dominating Set in Planar Graphs

Case 1.2. If $N_3(v,w) \subseteq N(v)$ but not $N_3(v,w) \subseteq N(w)$:

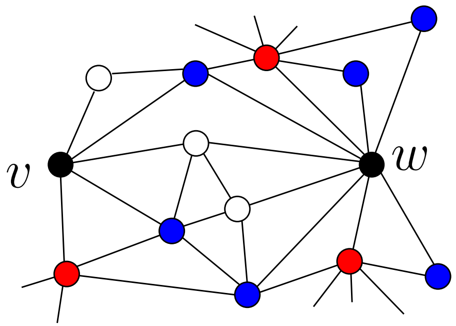
- remove $N_3(v,w)$ and $N_2(v,w) \cap N(v)$ from G and
- add a new vertex v' and the edge $\{v, v'\}$ to G.

Case 1.3. Symmetrical to Case 1.2 with roles of v and w interchanged.

DOMINATING SET IN PLANAR GRAPHS

Case 2. If $N_3(v, w)$ cannot be dominated by a single vertex from $\{v, w\}$,

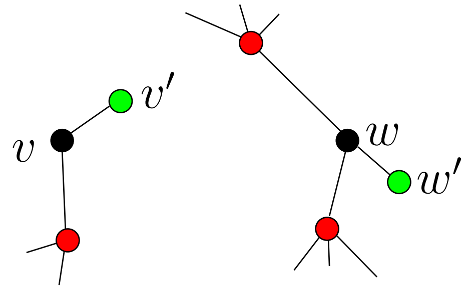
- remove $N_3(v, w)$ and $N_2(v, w)$ from G and
- add two new vertices v', w' and edges $\{v, v'\}, \{w, w'\}$ to G.



data

reduction

$\Longrightarrow$



DOMINATING SET IN PLANAR GRAPHS

Lemma Rule 2 can be carried out in $O(|V|^2)$ time for planar graphs and in $O(|V|^4)$ time for general graphs.

Lemma A graph G can be transformed into a graph G', that is reduced with respect to Rules 1 and 2, in $O(|V|^3)$ time for planar graphs and in $O(|V|^6)$ time for general graphs.

It is possible to show that DOMINATING SET IN PLANAR GRAPHS admits a linear problem kernel with $O(k)$ vertices and edges.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

❷ Algorithmic methods

- ➪ Data reduction and problem kernels
- ➪ Depth-bounded search trees
- ➪ Some advanced techniques
- ➪ Dynamic programming
- ➪ Tree decompositions of graphs

❸ Parameterized complexity theory

Depth-bounded search trees

Basic idea

In polynomial time find a "small subset" of the input instance such that at least one element of this subset is part of an optimal solution to the problem.

Example VERTEX COVER

Small subset $:= \{$ two endpoints of an edge $\}$. One of these two endpoints has to be part of the vertex cover. A search tree of size $O(2^k)$ with $k :=$ the size of the vertex cover.

As a rule, the depth of a search tree is upper-bounded by the parameter value.

Independent Set in Planar Graphs I

☞ Input: A planar graph $G = (V, E)$ and a nonnegative integer k.

☞ Task: Find a subset $I \subseteq V$ with at most k vertices that form an independent set, that is, I induces an edge-less subgraph of G.

Central observation following from Euler formula: In every planar graph there is at least one vertex of degree five or smaller.

Basic idea Pick a vertex v with minimum degree and put v or one of its neighbors into the independent set.

✗ Branching into at most six cases. ✗ Search tree of size $O(6^k)$.

Independent Set in Planar Graphs II

Proposition Independent Set in Planar Graphs can be solved in $O(6^k \cdot n)$ time where n denotes the number of vertices.

But:

- It is known that Independent Set in general graphs can be solved in $O(1.21^n)$ time.
- Because of the four-color theorem for planar graphs only the case $k > \lceil n/4 \rceil$ is really interesting ...

With $k > \lceil n/4 \rceil$ we have $6^k > 1.56^n > 1.21^n$ and, thus, the above search tree algorithm is useless ...

DOMINATING SET IN PLANAR GRAPHS I

☞ Input: A planar graph $G = (V, E)$ and a nonnegative integer k.

☞ Task: Find a subset $S \subseteq V$ with at most k vertices such that every vertex $v \in V$ is contained in S or has at least one neighbor in S.

Does a similar argument as for INDEPENDENT SET apply? No!

Problem It could be necessary to incorporate an already dominated vertex into the dominating set.

A branching argument analogous to INDEPENDENT SET works only for vertices that are not dominated.

DOMINATING SET IN PLANAR GRAPHS II

Way out of the difficulty: study a more general version of DOMINATING SET IN PLANAR GRAPHS

ANNOTATED DOMINATING SET IN PLANAR GRAPHS

- ☞ Input: A planar graph $G = (B \uplus W, E)$ with its vertices either colored black or white and a nonnegative integer k.
- ☞ Task: Find a subset $S \subseteq (B \uplus W)$ with at most k vertices such that every vertex in B is contained in S or has at least one neighbor in S.

Idea: Branching only for black vertices.

DOMINATING SET IN PLANAR GRAPHS III

Central question: Is there a black vertex with low degree?

With annotated vertices we cannot guarantee the existence of a black vertex of degree five or smaller: Consider a star with one black vertex adjacent to many white vertices.

In order to be able to devise a depth-bounded search tree:

1. Provide a set of data reduction rules,
2. show that there is always a black vertex v with its degree upper-bounded by seven,
3. and branch on v (yielding at most eight cases to branch into).

DOMINATING SET IN PLANAR GRAPHS IV

$\boxed{\text{Data reduction rules}}$ for simplifying instances of ANNOTATED DOMINATING SET IN PLANAR GRAPHS:

D1 Delete edges between white vertices.

D2 Delete degree-one white vertices.

D3 If there is a degree-one black vertex w with neighbor u (either black or white), then delete w, place u into the dominating set, and decrement parameter k by one.

D4 If there is a white vertex u of degree two with two black neighbors u_1 and u_2 connected by an edge $\{u_1, u_2\}$, then delete u.

DOMINATING SET IN PLANAR GRAPHS V

Data reduction rules continued:

D5 If there is a white vertex u of degree two with black neighbors u_1 and u_2 and if there is a vertex u_3 with edges $\{u_1, u_3\}$ and $\{u_2, u_3\}$, then delete u.

D6 If there is a white vertex u of degree three with black neighbors u_1, u_2, and u_3, and additionally existing edges $\{u_1, u_2\}$ and $\{u_2, u_3\}$, then delete u.

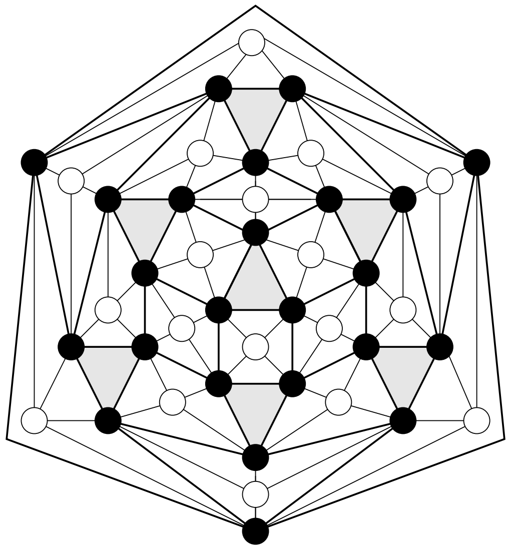
Lemma The data reduction rules are correct.

Proof: Trivial !

DOMINATING SET IN PLANAR GRAPHS VI

A graph where none of the above data reduction rules applies any more is called a reduced graph.

Example The following graph is reduced.



Dominating Set in Planar Graphs VII

Lemma Applying D1–D6, a given black-and-white graph $G = (B \uplus W, E)$ can be transformed into a reduced black-and-white graph $G' = (B' \uplus W', E')$ in $O(n^2)$ time, where $n := |B \uplus W|$.

Proof: Omitted.

The following lemma is decisive for the search tree algorithm:

Lemma If $G = (B \uplus W, E)$ is a planar black-and-white graph that is reduced, then there exists a black vertex $u \in B$ with degree at most seven.

The lengthy proof of this lemma relies on “Euler-like considerations”.

Dominating Set in Planar Graphs VIII

Theorem (Annotated) Dominating Set in Planar Graphs can be solved in $O(8^k \cdot n^2)$ time.

Proof:

Based on the preceding lemmas we can compute in $O(n^2)$ time a reduced planar black-and-white graph where there exists a black vertex with degree at most seven.

Branch with respect to a black vertex with minimum degree into at most eight cases, each case representing a possible way to dominate this black vertex.

Leads to a size-$O(8^k)$ search tree.

DOMINATING SET IN PLANAR GRAPHS IX

Theorem (ANNOTATED) DOMINATING SET IN PLANAR GRAPHS can be solved in $O(8^k \cdot k^2 + n^3)$ time.

Proof: Apply the two data reduction rules for DOMINATING SET IN PLANAR GRAPHS introduced in Chapter “Data reduction and problem kernels”. Exhaustive application of the two rules needs $O(n^3)$ time.

Results in an $O(k)$-vertex planar graph (the problem kernel).

Apply the search tree algorithm to this planar graph.

Closest String I

- Input: A set of k strings $s_1, \ldots, s_k$ over alphabet Σ of length L each and a nonnegative integer d.
- Task: Find a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$.

Lemma If there are $i, j \in \{1, \ldots, k\}$ with $d_H(s_i, s_j) > 2d$, then there is no string s with $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$.

Proof: The Hamming distance satisfies the triangle inequality:

$d_H(q, r) \leq d_H(q, t) + d_H(t, r)$, for arbitrary strings q, r, and t.

Thus, from $d_H(s_i, s_j) > 2d$ it follows that $d_H(s, s_i) > d$ or $d_H(s, s_j) > d$ for any string s. No solution!

Closest String II

Application Primer design.

```
...GGTGAG ATCTATAGAAGT TGAATGC...
...GGTGGA ATCTACAGTAAC GGATTGT...
...GGCGAG ATCTACAGAAGT GGAATGC...
...GGCGAG ATCTATAGAGAT GGAATGC...
...GGCAAG ATCTATAGAAGT GGAATGC...
```

"closest string": ATCTACAGAAAT

"primer candidate": TAGATGTCTTTA

Closest String III

Search tree algorithm for Closest String uses a recursive procedure *CSd* $(s, \Delta d)$:

Global variables: Set of strings $S = \{s_1, \ldots, s_k\}$, nonnegative integer d.

Input: Candidate string s and integer Δd.

Output: A string $\hat{s}$ with $\max_{i=1,\ldots,k} d_H(\hat{s}, s_i) \leq d$ and $d_H(\hat{s}, s) \leq \Delta d$, if it exists, and "not found", otherwise.

Method:

(D0) if $\Delta d < 0$ then return "not found";

(D1) if $d_H(s, s_i) > d + \Delta d$ for some $i \in \{1, \ldots, k\}$
then return "not found";

Closest String IV

(D2) if $d_H(s, s_i) \leq d$ for all $i \in \{1, \ldots, k\}$ then return s;

(D3) choose any $i \in \{1, \ldots, k\}$ such that $d_H(s, s_i) > d$:

$P := \{p \mid s[p] \neq s_i[p]\}$;

choose any $P' \subseteq P$ with $|P'| = d + 1$;

for all $p \in P'$ do

$s' := s$; $\quad s'[p] := s_i[p]$;

$s_{ret} :=$ CSd$(s', \Delta d - 1)$;

if $s_{ret} \neq$ "not found" then return s_{ret};

(D4) return "not found";

Closest String V

Theorem Closest String can be solved in $O(k \cdot L + k \cdot d^{d+1})$ time.

Proof: We show that the call *CSd*(s_1, d) solves Closest String in the claimed running time.

Correctness: Only show the correctness of the first recursive step where s_1 is the candidate string; the correctness of the algorithm follows with an inductive argument.

To show: At least one of the $(d+1)$ subcases of the branching in (D3) leads to a solution of the given instance if one exists.

Closest String VI

Proof: [Correctness]

Let s_i, $i \in \{2, \ldots, k\}$, denote an input string with $d_H(s_1, s_i) > d$.

Consider $P := \{p \mid s_1[p] \neq s_i[p]\}$.

Assume that $\hat{s}$ is a desired solution of the given instance.

Partition P into $P_1 := \{p \mid s_1[p] \neq s_i[p] \land s_i[p] = \hat{s}[p]\}$ and

$$P_2 := \{p \mid s_1[p] \neq s_i[p] \land s_i[p] \neq \hat{s}[p]\}.$$

Since $d_H(\hat{s}, s_i) \leq d$, $|P_2| \leq d$. From $|P| \leq 2d$, at least one of the $d+1$ chosen positions of the branching in (D3) is from P_1 and, thus, one of the branching subcases leads to a solution.

Closest String VII

Proof:

Running time: The depth of the search tree is upper-bounded by d and the number of branching cases is at most $d+1$.

$\rightsquigarrow$ The size of the search tree: $O((d+1)^d) = O(d^d)$.

The problem kernel of size $k \cdot d$ can be computed in $O(k \cdot L)$ time.

At each search tree node, $O(k \cdot d)$ time.

Altogether, we have the running time of $O(k \cdot L + k \cdot d^{d+1})$.

Analysis of search tree sizes I

Until now:

Extremely regular branching strategies: Determination of the upper bounds on the search tree sizes requires little mathematical effort.

Now:

Complicated branchings strategies with numerous case distinctions: Estimation of the size of the corresponding search trees requires rigorous mathematical analysis.

Mathematical tool: Recurrence relations.

Analysis of search tree sizes II

Idea Transform the recursive structure of search tree algorithms into recurrence relations.

Homogeneous, linear recurrence relations with constant coefficients.

Example Trivial $O(2^k)$ search tree algorithm for VERTEX COVER

Recurrence relation for the size of the search tree T_i:

$$T_i = 1 + T_{i-1} + T_{i-1}, \quad T_0 = 1.$$

It suffices to estimate the number of leaves of the search tree:

$B_i = B_{i-1} + B_{i-1}, \; B_0 = 1.$ Solution: $B_i = 2^i$ (Trivial).

Analysis of search tree sizes III

In general

$$B_i = B_{i-d_1} + B_{i-d_2} + \cdots + B_{i-d_r}$$

where we set

$$B_0 = B_1 = \cdots = B_{\max\{d_1, d_2, \ldots, d_r\}-1} = 1.$$

This means that the search tree algorithm solving a problem of size i calls itself recursively for problems of sizes $i - d_1, \ldots, i - d_r$.

Analysis of search tree sizes IV

Example Improved search tree algorithm for VERTEX COVER

Three cases:

(1) Degree-one vertex: take its neighbor into the vertex cover;	No branching.
(2) Degree-two vertex u: take either u's two neighbors v and w or all neighbors of v and w into the vertex cover;	$B_i = B_{i-2} + B_{i-2}$ $\leadsto$ Solution $O(1.42^k)$
(3) Degree-three vertex u: take u or all its neighbors into the vertex cover.	$B_i = B_{i-1} + B_{i-3}$ $\leadsto$ Solution $O(1.47^k)$

Analysis of search tree sizes V

- Branching vector $(d_1, d_2, \ldots, d_r)$ characterizes the above recurrence relation uniquely.

- The roots of the "characteristic polynomial" $z^d = z^{d-d_1} + z^{d-d_2} + \cdots + z^{d-d_r}$, where $d := \max\{d_1, d_2, \ldots, d_r\}$, determine the solution of the recurrence relation.

In our context, the characteristic polynomial has always a single root α which has maximum absolute value.

$|\alpha|$ is called the branching number with respect to the branching vector $(d_1, \ldots, d_r)$.

Analysis of search tree sizes VI

It is possible to show that $B_i = O(|\alpha|^i)$.

Summary: We can solve the recurrence relation by computing the root α of the characteristic polynomial, e.g., using the Newton method.

Example Improved search tree algorithm for VERTEX COVER (continued)

We obtain the branching vectors $(2, 2)$ and $(1, 3)$ for Cases (2) and (3), respectively, and the corresponding branching numbers are bounded by 1.42 and 1.47, respectively.

$\rightsquigarrow$ The search tree size is $O(1.47^k)$ (worst-case!).

Analysis of search tree sizes VII

Remarks:

- ✗ Several cases of recursive branching:

  Compute for each case the corresponding branching vector; the maximum branching number provides the worst-case bound for the search tree size.

- ✗ One thing to learn from the above point: In practical applications, the sizes of the search trees may be significantly smaller than the theoretical worst-case bounds.

- ✗ Further heuristic improvement on the bounds of search tree sizes can be achieved by incorporating techniques such as Branch&Bound.

Analysis of search tree sizes VIII

Example Branching vector and branching number

Branching vector	Branching number	Branching vector	Branching number
(1,1)	2.0	(1,1,1)	3.0
(1,2)	1.6180	(1,1,2)	2.4142
(1,3)	1.4656	(1,1,3)	2.2056
(1,4)	1.3803	(1,1,4)	2.1069
(2,1)	1.6180	(1,2,1)	2.4142
(2,2)	1.4142	(1,2,2)	2.0
(2,3)	1.3247	(1,2,3)	1.8929
(2,4)	1.2720	(1,2,4)	1.7549

Cluster Editing I

☞ Input: A graph G and a nonnegative integer k.

☞ Task: Find out whether we can transform G, by deleting or adding at most k edges, into a graph that consists of a disjoint union of cliques.

Forbidden subgraph characterization:

Lemma A graph $G = (V, E)$ consists of disjoint cliques iff there are no three vertices $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$.

Such three vertices are called "conflict triple".

CLUSTER EDITING II

Simple branching strategy:

(1) If G is already a union of disjoint cliques, then return the solution.

(2) Otherwise, if $k \leq 0$, then return that there is no solution in this branch.

(3) Otherwise, identify a conflict triple u, v, w. Recursively call the branching procedure on the following three instances with $G' = (V, E')$ and k':

(B1) $E' := E \setminus \{\{u, v\}\}$ and $k'; = k - 1$;

(B2) $E' := E \setminus \{\{u, w\}\}$ and $k'; = k - 1$;

(B3) $E' := E \cup \{\{v, w\}\}$ and $k'; = k - 1$;

Proposition There is a size-$O(3^k)$ search tree for CLUSTER EDITING.

Cluster Editing III

In order to achieve an improved branching strategy, distinguish three situations when considering a conflict triple u, v, w:

(C1) v and w have no common neighbor besides u.

(C2) v and w have a common neighbor x with $x \neq u$ and $\{u, x\} \in E$.

(C3) v and w have a common neighbor x with $x \neq u$ and $\{u, x\} \notin E$.

CLUSTER EDITING IV

Recall: In Chapter "Data reduction and problem kernels" we introduced the table T which has an entry for every pair of vertices $u, v \in V$. This entry is either empty or takes one of the following two values:

- "permanent": $\{u, v\} \in E$ and it is not allowed to delete $\{u, v\}$;
- "forbidden": $\{u, v\} \notin E$ and it is not allowed to add $\{u, v\}$;

Moreover, a data reduction rule can be applied to table T:

(1) If $T[u, v] =$ permanent and $T[u, w] =$ permanent, then $T[v, w] :=$ permanent;

(2) If $T[u, v] =$ permanent and $T[u, w] =$ forbidden, then $T[v, w] :=$ forbidden.

Table T and this data reduction rule are used to achieve the improved branching strategy.

Cluster Editing V

Improved branching strategy for case (C1): A branching into two cases (B1) and (B2) suffices.

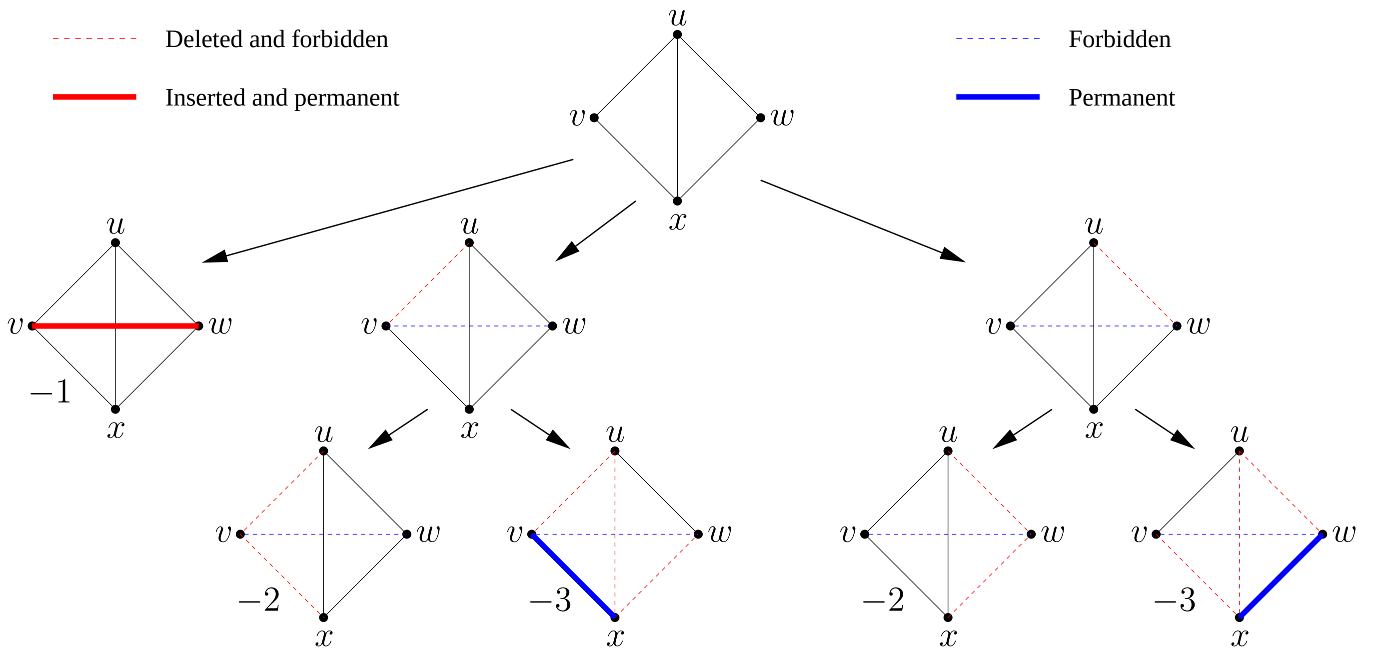
Lemma Given a graph $G = (V, E)$ and a conflict triple u, v, w with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$, if v and w have no common neighbor besides u, then branching case (B3) cannot yield a better solution than cases (B1) and (B2), and it can therefore be omitted.

Proof: Blackboard.

$\leadsto$ Branching vector $(1, 1)$, branching number 2.0.

Cluster Editing VI

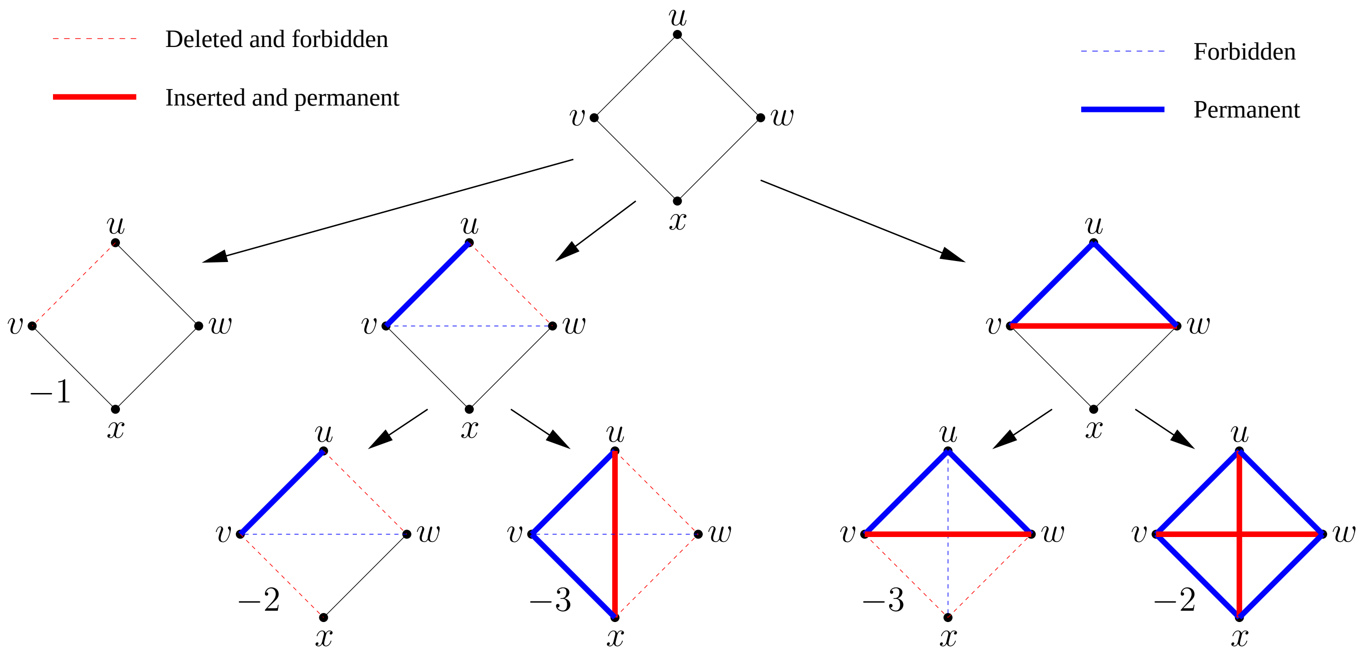
Improved branching strategy for case (C2):



$\rightsquigarrow$ Branching vector $(1, 2, 3, 2, 3)$, branching number 2.27.

Cluster Editing VII

Improved branching strategy for case (C3):



$\rightsquigarrow$ Branching vector $(1, 2, 3, 3, 2)$, branching number 2.27.

Cluster Editing VIII

$\rightsquigarrow$ The worst-case branching number is 2.27 (cases (C2) and (C3)).

Theorem There is a size-$O(2.27^k)$ search tree for Cluster Editing.

Using the kernelization process for Cluster Editing as a preprocessing, we achieve an algorithm solving Cluster Editing in $O(2.27^k \cdot k^6 + n^3)$ time.

Further improved search tree for VERTEX COVER I

Basic idea: A complex case distinction based on vertex degrees.

More specifically, consider vertex degrees in the following order:

first degree one, then degrees five and higher, then degrees two and three, and finally degree four.

Thus, when dealing with vertices of a specified degree, assume that there exists no vertex with the degrees which have been considered before.

W.l.o.g. assume the input graph $G = (V, E)$ connected.

Further improved search tree for Vertex Cover II

Case distinction for Vertex Cover:

Case 1. $\exists$ degree-one vertex v with u as its neighbor.

$\leadsto$ Take u into the vertex cover.

Branching vector: no branching.

Case 2. $\exists$ vertex v with degree five or higher.

$\leadsto$ Take v or $N(v)$ into the vertex cover.

Branching vector: $(1, 5)$.

Further improved search tree for VERTEX COVER III

Case 3. $\exists$ degree-two vertex v with $N(v) = \{a, b\}$.

Case 3.1 $\exists$ an edge between a and b.

$\rightsquigarrow$ Take $N(v)$ into the vertex cover.

Correctness: Vertices a, b, v form a triangle—thus, two of them have to be in the vertex cover—and a, b cover a superset of the edges covered by any other “two out of three” combination.

Branching vector: no branching.

Further improved search tree for VERTEX COVER IV

Case 3.2. $\nexists$ edge between a and b and $N(a) = N(b) = \{v, a_1\}$.

$\leadsto$ Take v, a_1 into the vertex cover.

Correctness: Vertices v, a_1 cover a superset of the edges covered by any other pair of vertices from $\{v, a, b, a_1\}$.

Branching vector: no branching.

Further improved search tree for Vertex Cover V

Case 3.3. $\nexists$ edge between a and b and $|N(a) \cup N(b)| \geq 3$.

$\leadsto$ Take either $N(v)$ or $N(a) \cup N(b)$ into the vertex cover.

Correctness: The first branching deals with the case that v is not part of an optimal vertex cover and, then, all its neighbors have to be. If v is in an optimal vertex cover, then it is not necessary to search for an optimal vertex cover containing x and one of a and b. The reason is that choosing a and b then also must give an optimal vertex cover. Then, the vertices in $N(a) \cup N(b)$ have to be in the optimal vertex cover.

Branching vector: $(2, 3)$ (note $|N(a) \cup N(b)| \geq 3$).

Further improved search tree for VERTEX COVER VI

Case 4. $\exists$ degree-three vertex v with $N(v) = \{a, b, c\}$.

Case 4.1 $\exists$ an edge between two vertices in $N(v)$, say a and b.

$\rightsquigarrow$ Take either $N(v)$ or $N(c)$ into the vertex cover.

Correctness: If v is not in the vertex cover, then $N(v)$ is. If v is a part of the vertex cover, then is a or b in order to cover edge $\{a, b\}$. If c was also in the cover, then $N(v)$ would cover a superset of the edges covered by, for instance, $\{v, a, c\}$. Hence, we choose $N(c)$ which includes v.

Branching vector: $(3, 3)$.

Further improved search tree for VERTEX COVER VII

Case 4.2. $\exists$ a common neighbor d of two neighbors of v with $d \neq v$, say d is a common neighbor of a and b.

$\rightsquigarrow$ Take either $N(v)$ or $\{d, v\}$ into the vertex cover.

Correctness: If v is not in the vertex cover, then all vertices in $N(v)$ have to be. If v is a part of the vertex cover, then not choosing d would mean that we would have to take a and b. This, however, cannot give a smaller vertex cover than choosing $N(v)$.

Branching vector: $(3, 2)$.

Further improved search tree for Vertex Cover VIII

Case 4.3. $\nexists$ edge between a, b, c and one vertex in $N(v)$ has degree at least four, say $N(a) = \{v, a_1, a_2, a_3\}$.

$\leadsto$ Take either $N(v)$ or $N(a)$ or $\{a\} \cup N(b) \cup N(c)$ into the vertex cover.

Correctness: If both of v and a are part of the vertex cover, then it is of no interest to choose additionally b or c. (For example $\{v, a, b\}$ is never better than $N(v)$.)

Branching vector: $(3, 4, 6)$ (note that $|\{a\} \cup N(b) \cup N(c)| \geq 6$, since, due to Case 4.2, $N(b) \cap N(c) = \{v\}$ and, hence, $|N(b) \cup N(c)| \geq 5$. In addition, $a \notin N(b)$ and $a \notin N(c)$ due to Case 4.1).

Further improved search tree for VERTEX COVER IX

Case 4.4. Otherwise, i.e., there is no edge between a, b, c and all of a, b, c have degree three.

$\leadsto$ Take either $N(v)$ or $N(a) = \{v, a_1, a_2\}$ or $N(b) \cup N(c) \cup N(a_1) \cup N(a_2)$ into the vertex cover.

Correctness: If both v and a are in the vertex cover, then it makes no sense to take additionally b or c. With the same argument, taking a_1 or a_2 would result in not taking a.

Branching vector: $(3, 3, 6)$ (note that $N(b) \cap N(c) = \{v\}$ and then $|N(b) \cup N(c)| = 5$; moreover, $a \in N(a_1)$ and $a \notin (N(b) \cup N(c))$).

Further improved search tree for VERTEX COVER X

Case 5. The graph is four-regular, i.e., each vertex has degree four.

$\rightsquigarrow$ Choose an arbitrary vertex v; take either v or $N(v)$ into the vertex cover.

Branching vector: $(1, 4)$.

This case can occur only once in an application of the search tree algorithm: Taking v or $N(v)$ into the vertex cover results in at least one vertex with degree at most three. Thus, this case is neglectable for the analysis of the search tree size.

Further improved search tree for VERTEX COVER XI

Theorem There is a size-$O(1.342^k)$ search tree for VERTEX COVER.

Proof

Case	Branching vector	Branching number	Case	Branching vector	Branching number
1	-	-	4.1	(3,3)	1.260
2	(1,5)	1.325	4.2	(3,2)	1.325
3.1	-	-	4.3	(3,4,6)	1.305
3.2	-	-	4.4	(3,3,6)	1.342
3.3	(2,3)	1.325	5	(1,4)	1.381

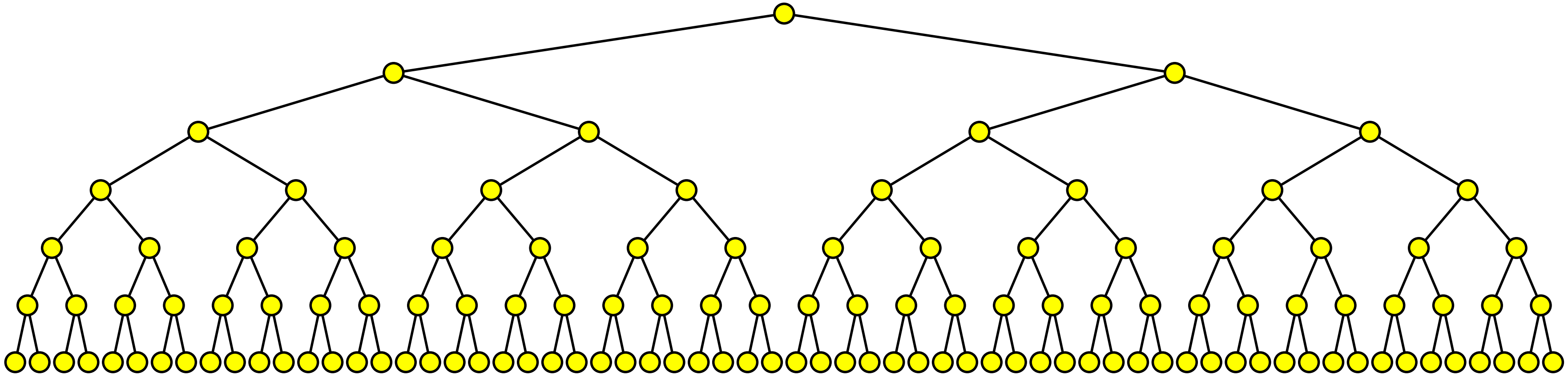
Further improved search tree for VERTEX COVER XII

Remarks:

- ✗ The currently best fixed-parameter algorithms for VERTEX COVER—which are based on depth-bounded search trees—have a search tree size of $O(1.28^k)$.
- ✗ The best non-parameterized, exact algorithm for VERTEX COVER has a running time of $O(1.22^n)$.

Interleaving Search Trees and Kernelization I

Initial situation:

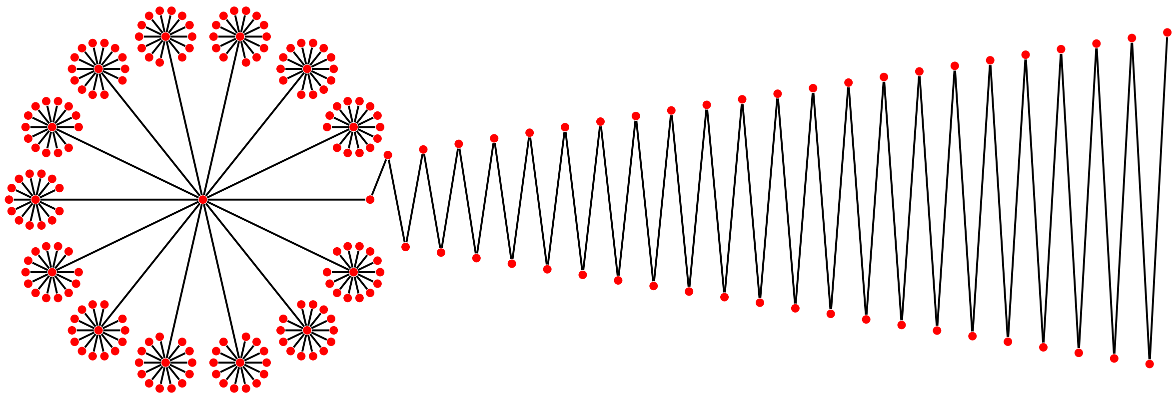


While the parameter value is decreased along the paths from the root to the leaves, the instance size could remain almost the same.

If we, at each search tree node, need $P(n)$ time for computing the branching subcases, then the overall running time is $O(s(k, n) \cdot P(n))$, where $s(k, n)$ denotes the search tree size.

Interleaving Search Trees and Kernelization II

Example VERTEX COVER



Consider the trivial size-2^k search tree and Buss' kernelization process with $k = 15$. Buss' data reduction rule is not applicable to the above graph. Assume that the algorithm chooses edges from right to left. While examining this graph, the algorithm removes vertices and edges but the "head" (the left part) remains unchanged. Consequently, instances have size $\Theta(k^2)$ during *each* branching step.

Interleaving Search Trees and Kernelization III

Basic idea: When making a branching step which, for a given instance (I, k), produces r new instances, $(I_1, k - d_1)$, $(I_2, k - d_2)$, . . ., $(I_r, k - d_r)$, use the following extended algorithm. Herein, let $q(k)$ denote the problem kernel size.

(1) if $|I| > c \cdot q(k)$ then

Apply the data reduction process to (I, k) and set

$(I, k) := (I', k')$ where (I', k') forms a problem kernel.

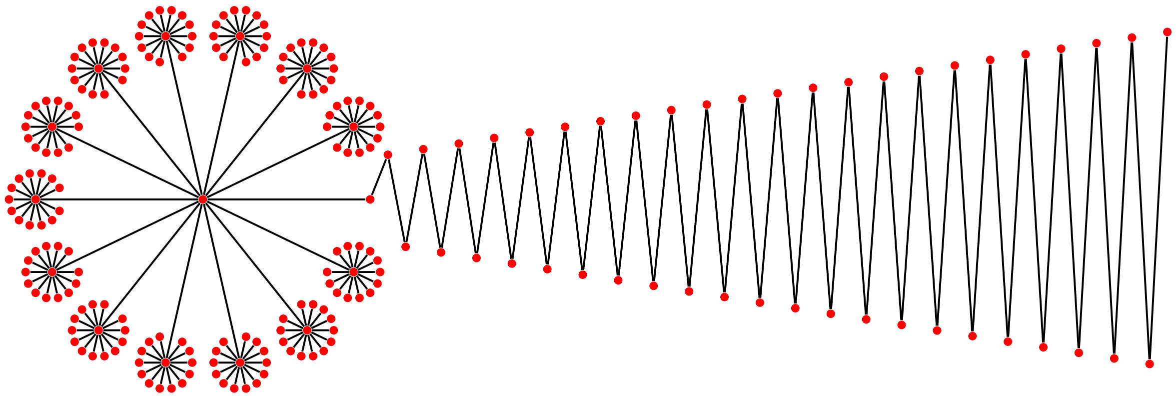
(2) Replace (I, k) with $(I_1, k - d_1), (I_2, k - d_2), \ldots, (I_r, k - d_r)$.

Here, $c \geq 1$ is a constant that can be chosen with the aim of further optimizing the running time.

Interleaving Search Trees and Kernelization IV

The key point: We repeatedly apply the kernelization process inside the search tree.

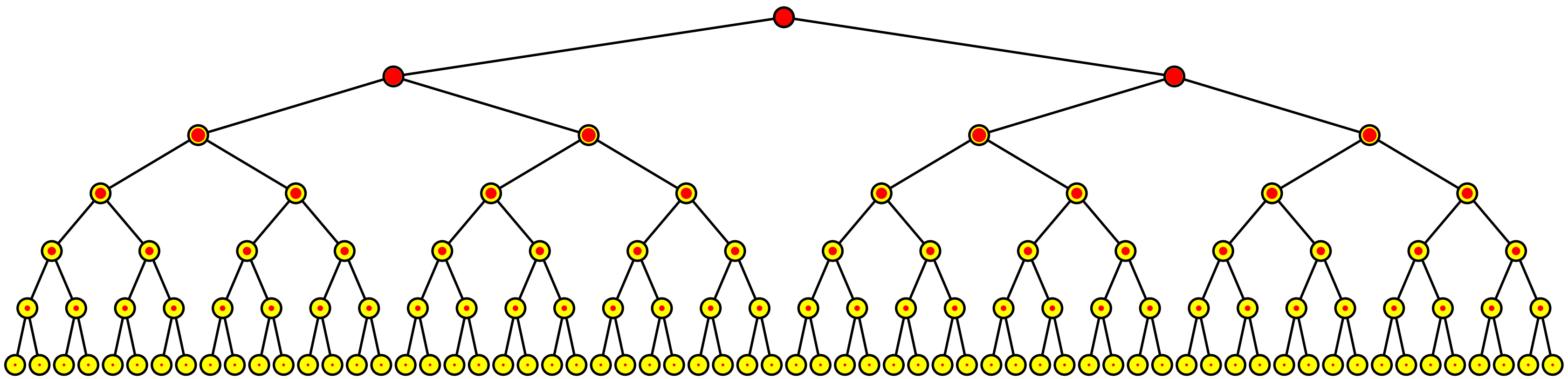
Example VERTEX COVER (continued)



In the above graph, after the *second* edge is removed and parameter k is decreased by two, the whole head will be removed from the graph.

Interleaving Search Trees and Kernelization V

Illustration of the parameter decrease (without interleaving)

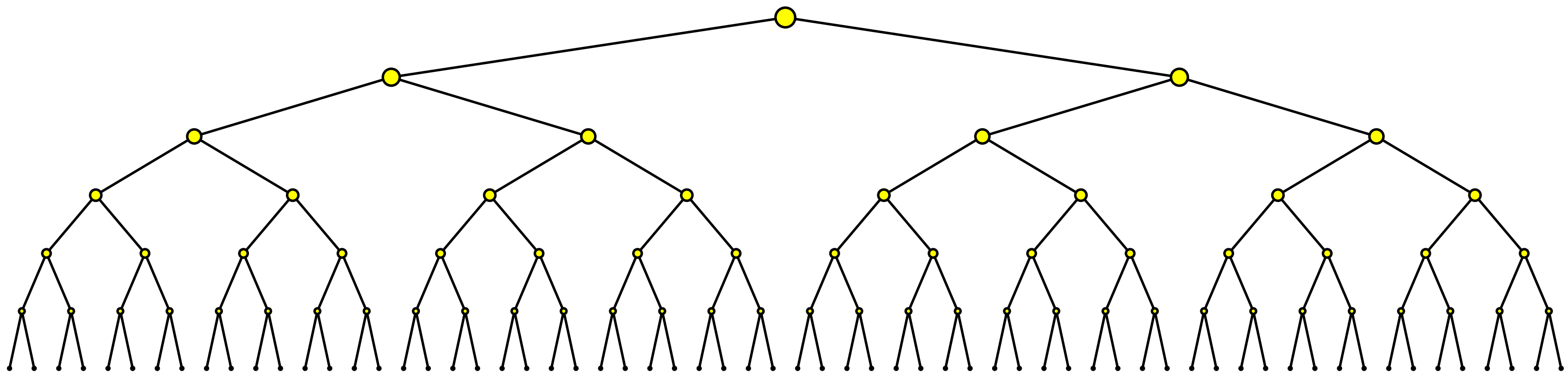


The red part of a node figuratively shows the parameter value. The recursion stops if the parameter is less than a constant (for instance, 1).

Almost all nodes are leaves or near leaves and have a small parameter value.

Interleaving Search Trees and Kernelization VI

Illustration of the parameter decrease (with interleaving)



Compared with the case without interleaving, not only the parameter values are smaller, but also the size of the instances (illustrated by the sizes of the nodes).

Interleaving Search Trees and Kernelization VI

The following can be shown.

Let (I,k) denote the input instance of a parameterized problem. Let there be an algorithm solving this problem in $O(P(|I|) + R(q(k)) \cdot \alpha^k)$ time: First, it reduces (I,k) in $P(|I|)$ time to a problem kernel (I',k') with $|I'| = q(k)$ and then it applies a search tree of size $O(\alpha^k)$ to (I',k') where at each search tree node it needs $O(R(q(k)))$ time.

(Note: This is clearly a two-phases approach: kernelization (1st phase) $+$ search tree (2nd phase).)

Then, by interleaving the kernelization and the search tree, the running time can be improved to $O(P(|I|) + \alpha^k)$.

Automated Search Tree Generation and Analysis I

Initial situation:

Both search tree algorithms for VERTEX COVER and CLUSTER EDITING are based on fairly extensive case distinctions.

Idea Automated case distinctions.

Human part:

$\leadsto$ develop clever "problem-specific rules".

Machine part:

$\leadsto$ analyze numerous cases using the problem-specific rules.

Automated Search Tree Generation and Analysis II

Rough framework:

Step 1. For a constant c, enumerate all “relevant” subgraphs of size c such that every input instance has c vertices inducing at least one of the enumerated subgraphs.

Step 2. For every local substructure enumerated in Step 1, check all possible branching rules and select the one corresponding to the *best*, that is, smallest, branching number. The set of all these best branching rules then defines our search tree algorithm.

Step 3. Determine the worst-case branching rule stored in Step 2; this branching rule yields the worst-case bound on the search tree size of the generated algorithm.

Automated Search Tree Generation and Analysis III

Successful example: Cluster Editing

"Human search tree" of size $O(2.27^k)$ improved to computer-generated search tree strategy with search tree size $O(1.92^k)$.

Literature: Gramm et al., Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39:321–347, 2004.

Summary and Concluding Remarks

Search tree algorithms

- ✗ ... build one of the most important techniques to cope with the really hard kernel of a problem.
- ✗ ... can be easily parallelized.
- ✗ ... can often be further accelerated by incorporating heuristic techniques such as Branch&Bound.
- ✗ ... may be combined with approximation algorithms.
- ✗ ... in applications frequently have few cases of their case distinctions that occur very often and the remaining cases usually occur very seldom.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

❷ Algorithmic methods

- Data reduction and problem kernels
- Depth-bounded search trees
- Some advanced techniques
- Dynamic programming
- Tree decompositions of graphs

❸ Parameterized complexity theory

Some advanced techniques

- ✗ Color-coding
- ✗ Integer linear programming
- ✗ Iterative compression

Color-coding I

Many graph problems are special versions of the SUBGRAPH ISOMORPHISM problem:

- Input: Two graphs $G = (V, E)$ and $G' = (V', E')$.
- Task: Determine whether there is a subgraph of G that is isomorphic to G'.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are called isomorphic, if there is a bijective function $f : V \to V'$, such that $\{u, v\} \in E$ iff $\{f(u), f(v)\} \in E'$.

Examples:

Independent set: $G' =$ edgeless graph with k vertices.

Clique: $G' =$ complete graph with k vertices.

Color-coding II

Method used to derive (randomized) fixed-parameter algorithms for several subgraph isomorphism problems.

An example application to the NP-complete LONGEST PATH problem:

- ☞ Input: A graph $G = (V, E)$ and a nonnegative integer k.
- ☞ Task: Find a *simple* path of length k in G, i.e., a path consisting of k vertices such that no vertex may appear on the path more than once.

Color-coding III

Basic idea

Randomly color the whole graph with k colors and "hope" that all vertices of one path will obtain different colors.

A colorful path, a path of vertices with pairwise different colors, can be found by using *dynamic programming*.

A colorful path is clearly simple.

Color-coding IV

Randomized approach to solve LONGEST PATH:

Color the graph vertices uniformly at random with k colors.

A path is called colorful if all vertices of the path obtain pairwisely different colors.

Remark: Clearly, each colorful path is simple. Reversely, each simple path is colorful with probability $k!/k^k > e^{-k}$.

$\boxed{\text{Lemma}}$ Let $G = (V, E)$ and let $C : V \to \{1, \ldots, k\}$ be a coloring. Then a colorful path of k vertices can be found (if it exists) in $2^{O(k)} \cdot |E|$ time.

Color-coding V

Proof of Lemma:

We describe an algorithm that finds all colorful paths of k vertices starting at some fixed vertex s. This is not really a restriction because to solve the general problem, we may just add some extra vertex s' to V, color it with the new color 0, and connect it with each of the vertices in V by an edge.

To find the described paths, we use dynamic programming.

Assumption: $\forall v \in V$ already *all* possible color sets of colorful paths between s and v consisting of i vertices have been found.

Note: We store color sets instead of paths! For each vertex v, there are at most $\binom{k}{i}$ such color sets.

Color-coding VI

Proof of Lemma: (continued)

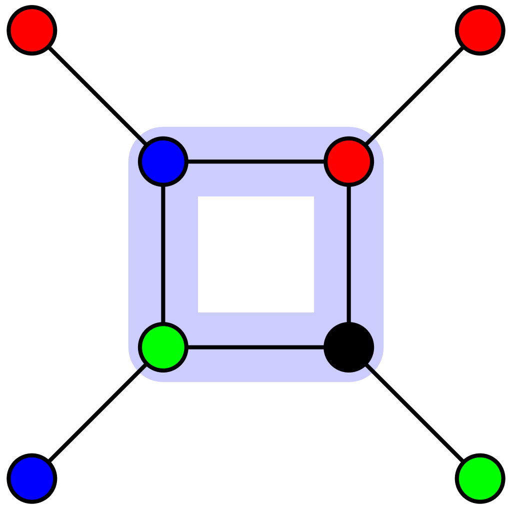
Let F be such a color set belonging to v. Consider every F and every edge $\{u, v\}$: If $C(u) \notin F$ then build the new color set $F' := F \cup \{C(u)\}$. Color set F' becomes part of the set of the color sets belonging to u. In this way, we obtain all color sets belonging to paths of length $i + 1$ and so on.

Graph G obtains a colorful path with respect to coloring C *iff* there exists a vertex $v \in V$ that has at least one color set that corresponds to a path of k vertices.

Time complexity: Algorithm performs $O(\sum_{i=1}^{k} i \cdot \binom{k}{i} \cdot |E|)$ steps. Herein, i refers to the test whether or not $C(u)$ is already contained in F. The factor $\binom{k}{i}$ refers to the number of possible sets F and $|E|$ refers to the time to check all edges $\{u, v\} \in E$. The whole expression is upper-bounded by $O(k \cdot 2^k \cdot |E|)$.

Color-coding VII

Example



With respect to a random coloring with 4 colors, the four highlighted vertices that build a path of length 4 (actually, a cycle) can be found with probability $4!/4^4 = 3/32$.

Color-coding VIII

Theorem LONGEST PATH can be solved in expected time $2^{O(k)} \cdot |E|$.

Proof: According to the above remark a simple path of k vertices is colorful with probability at least e^{-k}.

According to the above lemma, such a colorful path can be found in $2^{O(k)} \cdot |E|$ time; more precisely, all colorful paths of k vertices can be found.

Color-coding IX

Proof of Theorem (continued)

Algorithm:

We repeat the following $e^k = 2^{O(k)}$ times:

1. Randomly choose a coloring $C : V \to \{1, \ldots, k\}$.
2. Check using the above lemma whether or not there is a colorful path; if so then this is a simple path of k vertices.

After trying $2^{O(k)}$ random colorings, the expected value concerning the number of colorful paths found (if any are existing) is at least one.

Thus, LONGEST PATH can be solved in expected time $e^k \cdot 2^{O(k)} \cdot |E| = 2^{O(k)} \cdot |E|$.

Color-coding X

Using *hashing* (more precisely, so-called k-perfect families of hash functions), the above randomized algorithm can be de-randomized at the cost of somewhat increased running time.

The following can be shown.

> LONGEST PATH can be solved deterministically in $2^{O(k)} \cdot |E| \cdot \log |V|$ time.
>
> In particular, LONGEST PATH is fixed-parameter tractable with respect to parameter k.

Integer Linear Programming (ILP) I

Recall: ILP for VERTEX COVER

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{v \in V} x_v \\
\text{subject to} \quad & x_u + x_v \geq 1 \quad \forall e = \{u, v\} \in E \\
& x_v \in \{0, 1\} \quad \forall v \in V
\end{aligned}
$$

The following theorem is due to Hendrik W. Lenstra (1983).

Theorem Integer linear programs can be solved with $O(p^{9p/2}L)$ arithmetic operations in integers of $O(p^{2p}L)$ bits in size, where p is the number of ILP variables and L is the number of bits in the input.

Integer Linear Programming (ILP) II

An example application of Lenstra's theorem to CLOSEST STRING:

- Input: A set of k strings $s_1, \ldots, s_k$ over alphabet Σ of length L each and a nonnegative integer d.
- Task: Find a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$.

Goal: Give an ILP formulation for CLOSEST STRING such that the number of variables solely depends on the parameter k, the number of input strings.

Integer Linear Programming (ILP) III

Thinking of the input strings as a $k \times L$ character matrix, the key for the ILP formulation lies in the notion of *column types*.

Fact: The columns of the matrix are independent from each other in the sense that the distance from the closest string is measured columnwise.

For instance, consider two columns $(a, a, b)^t$ and $(b, b, a)^t$ when $k = 3$. These two columns are *isomorphic* because they express the same structure except the symbols play different roles.

Isomorphic columns form *column types*, that is, a column type is a set of columns isomorphic to each other.

Integer Linear Programming (ILP) IV

A CLOSEST STRING instance is called *normalized* if, in each column of its corresponding character matrix, the most often occurring letter is denoted by a, the second often occurring by b, and so on.

Lemma A CLOSEST STRING instance with arbitrary alphabet Σ, $|\Sigma| > k$, is isomorphic to a CLOSEST STRING instance with alphabet Σ', $|\Sigma'| = k$.

Example For $k = 3$, there are 5 possible column types for a normalized CLOSEST STRING instance:

$$(a, a, a)^t, (a, a, b)^t, (a, b, a)^t, (b, a, a)^t, (a, b, c)^t.$$

Integer Linear Programming (ILP) V

Generally, the number of column types for k strings is given by the so-called Bell number $B(k) \leq k!$.

Using the column types, CLOSEST STRING can be formulated as an ILP having only $B(k) \cdot k$ variables:

Variables: $x_{t,\varphi}$ where t denotes a column type which is represented by a number between 1 and $B(k)$ and $\varphi \in \Sigma$ where $|\Sigma| = k$.

Meaning: $x_{t,\varphi}$ denotes the number of columns of column type t whose corresponding character in the desired solution string is set to φ.

Integer Linear Programming (ILP) VI

The goal function of the ILP is

$$\text{Minimize} \quad \max_{1\leq i\leq k} \Big(\sum_{1\leq t\leq B(k)} \; \sum_{\varphi\in(\Sigma\setminus\{\varphi_{t,i}\})} x_{t,\varphi} \Big) \,,$$

where $\varphi_{t,i}$ denotes the symbol at the ith entry of column type t,

subject to

1. $x_{t,\varphi} \geq 0$ for all variables and
2. $\sum_{\varphi\in\Sigma} x_{t,\varphi} = \#_t$ for all column types t, where $\#_t$ denotes the number of columns of type t in the input instance (taking into account isomorphism as described before).

Integer Linear Programming (ILP) VII

Since the number p of the ILP variables is bounded by $B(k) \cdot k$, we arrive at the following theorem.

Theorem CLOSEST STRING is fixed-parameter tractable with respect to parameter k.

Remarks:

- ✗ The ILP approach helps classifying whether a problem is fixed-parameter tractable. The combinatorial explosion is huge.
- ✗ There exists an alternative ILP formulation for CLOSEST STRING where the variables have only binary values but the number of variables is $|\Sigma| \cdot L$ (for alphabet Σ and string length L).

Iterative Compression I

Basic idea

Use a *compression routine* iteratively.

Compression routine: Given a size-$(k+1)$ solution, either compute a size-k solution or prove that there is no size-k solution.

Algorithm for graph problems:

1. Start with empty graph G' and empty solution set X
2. For each vertex v in G:

   Add v to both G' and X

   Compress X using the compression routine.

Iterative Compression II

Example application to VERTEX COVER:

- ☞ Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- ☞ Task: Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

Let $V = \{v_1, v_2, \ldots, v_n\}$.

Iterative Compression III

Iteration for VERTEX COVER:

1. Start with $G_1 := G[\{v_1\}]$ and $C_1 := \emptyset$.
2. Given a cover C_i for $G_i := G[\{v_1, \ldots, v_i\}]$, compute a cover C_{i+1} for $G_{i+1} := G[\{v_1, \ldots, v_{i+1}\}]$.

Simple observation:

Clearly, $C_i \cup \{v_{i+1}\}$ is a vertex cover for G_{i+1}.

Question: Can we get a cover C_{i+1} with $|C_{i+1}| < |C_i \cup \{v_{i+1}\}|$?

Iterative Compression IV

Compression of $C'_{i+1} := C_i \cup \{v_{i+1}\}$ into C_{i+1}:

Try all $2^{|C'_{i+1}|}$ partitions of C'_{i+1} into two sets A and B.

- Assume that $A \subseteq C_{i+1}$ but $B \cap C_{i+1} = \emptyset$.
- B has to be an independent set!
- $A \cup N(B)$ is a vertex cover.
- Thus, C'_{i+1} can be compressed iff there is a partition of C'_{i+1} with $|A \cup N(B)| < |C'_{i+1}|$.

Result: Iterating n times (from G_1 to $G_n = G$), we can decide whether G has a size-k vertex cover in $O(2^k \cdot mn)$ time, where $m := |E|$.

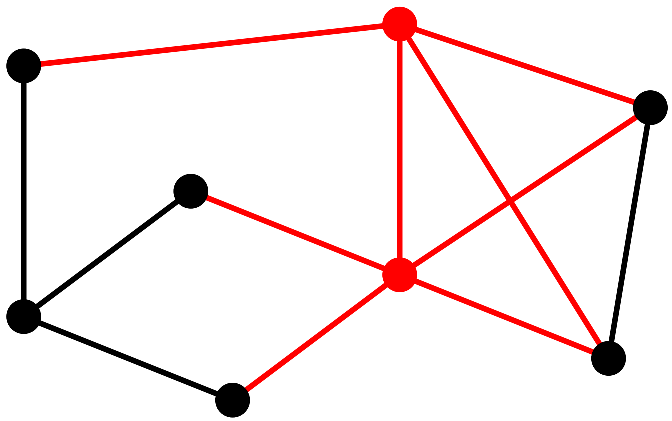
Iterative Compression V

Example application to FEEDBACK VERTEX SET (FVS):

- Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- Task: Find a subset of vertices $F \subseteq V$ with at most k vertices such that the deletion of F makes G cycle-free.

Set F is called *feedback vertex set*.

Example



Iterative Compression VI

Goal: An $O(c^k \cdot n^{O(1)})$ time algorithm for some constant c.

Iteration for FVS: (similar to the iteration for Vertex Cover)

1. Start with $G_1 := G[\{v_1\}]$ and $F_1 := \emptyset$.
2. Given a feedback vertex set F_i for $G_i := G[\{v_1, \ldots, v_i\}]$, compute a feedback vertex set F_{i+1} for $G_{i+1} := G[\{v_1, \ldots, v_{i+1}\}]$.

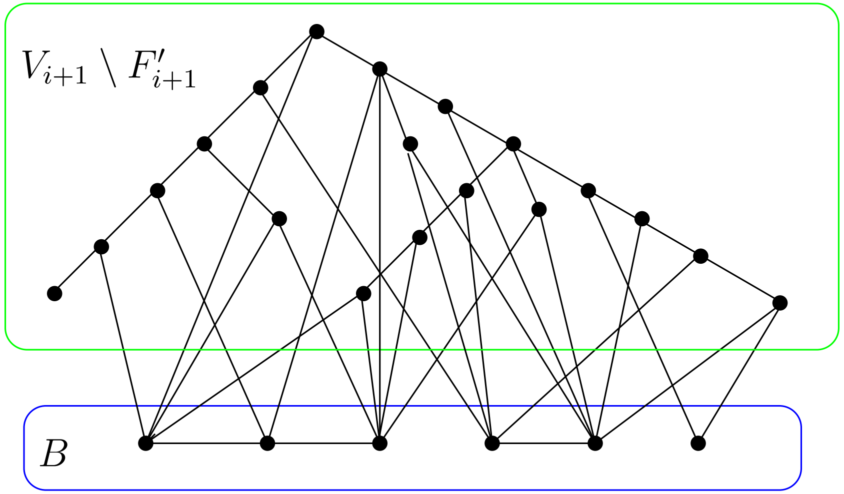
Clearly, $F_i \cup \{v_{i+1}\}$ is a feedback vertex set for G_{i+1}.

Iterative Compression VII

Compression of $F'_{i+1} := F_i \cup \{v_{i+1}\}$ into F_{i+1}:

Try all $2^{|F'_{i+1}|}$ partitions of F'_{i+1} into two sets A and B.

- Assume that $A \subseteq F_{i+1}$ but $B \cap F_{i+1} = \emptyset$.
- B has to induce a cycle-free graph!
- Remove vertices in A from G_{i+1}.

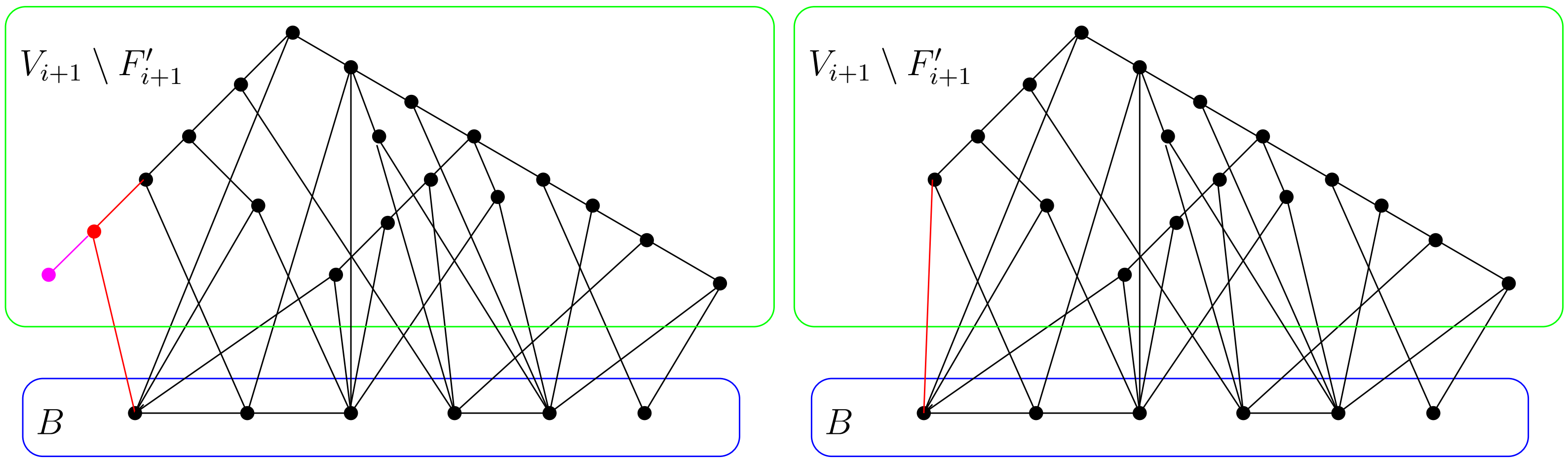


Iterative Compression VIII

Compression of $F'_{i+1} := F_i \cup \{v_{i+1}\}$ into F_{i+1}: (continued)

Idea: Shrink the set of candidates for F_{i+1} by using data reduction:

➠ Eliminate degree-one and degree-two vertices in $V_{i+1} \setminus F'_{i+1}$.



Iterative Compression IX

Compression of $F'_{i+1} := F_i \cup \{v_{i+1}\}$ into F_{i+1}: (continued)

Idea: If, in the reduced $G'_{i+1} = (V'_{i+1}, E'_{i+1})$, the set $V'_{i+1} \setminus F'_{i+1}$ is too large compared to F'_{i+1}, then there is no solution F_{i+1}.

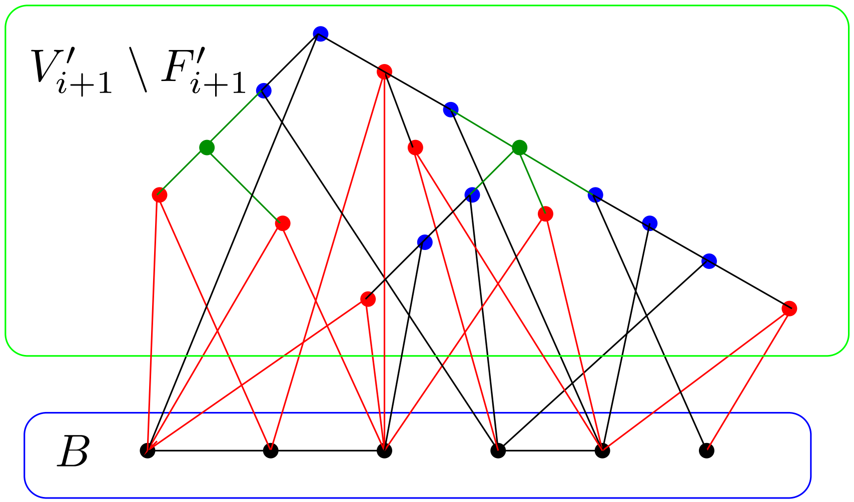
$\rightsquigarrow$ Use brute force to choose the candidates for F_{i+1}.

$\boxed{\text{Lemma}}$ If $|V'_{i+1} \setminus F'_{i+1}| > 14 \cdot |F'_{i+1}|$, then there is no solution F_{i+1}.

Proof: Partition $V'_{i+1} \setminus F'_{i+1}$ into three sets and separately upper-bound their sizes.

Iterative Compression X

Proof of Lemma: (continued)

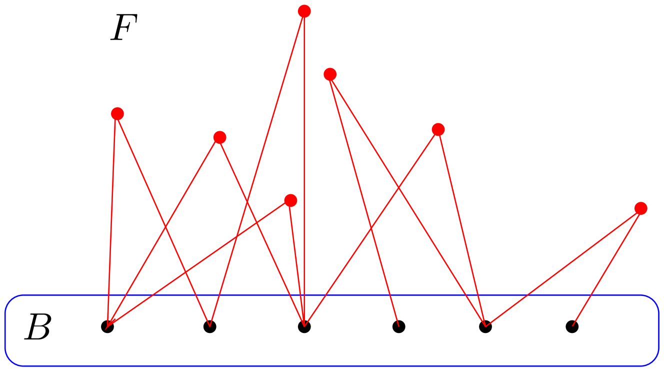


- at least two neighbors in B
- at least three neighbors in $V'_{i+1} \setminus F'_{i+1}$
- the rest

Iterative Compression XI

Proof of Lemma: (continued)

F: at least two neighbors in B.



Observation: If $|F| \geq |B|$, then there is a cycle.

Hence:

1. If $|F| \geq 2 \cdot |B|$, then with $|B| - 1$ deletions we cannot get rid of all cycles.
2. If there is a solution for FVS, then $|F| \leq 2k$.

Using a similar argument, we can derive the bounds for the other two sets.

Iterative Compression XII

Compression of $F'_{i+1} := F_i \cup \{v_{i+1}\}$ into F_{i+1}: (continued)

➠ Try all subsets of $V'_{i+1} \setminus F'_{i+1}$ that are of size at most $|B|$.

With the above lemma, the running time by brute force for the compression:

$O(37.7^k \cdot m)$ where $c := |E|$.

Theorem FEEDBACK VERTEX SET can be solved in $O(c^k \cdot mn)$ time for a constant c.

Remarks:

✗ FVS can be solved in $O(c^k \cdot m)$ time for a constant c.

✗ All minimal feedback vertex sets of size at most k can be enumerated in $O(c^k \cdot m)$ time for a constant c.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

❷ Algorithmic methods

- ➪ Data reduction and problem kernels
- ➪ Depth-bounded search trees
- ➪ Some advanced techniques
- ➪ Dynamic programming
- ➪ Tree decompositions of graphs

❸ Parameterized complexity theory

Dynamic Programming

A general technique applied to problems whose solution can be computed from solutions to subproblems.

Idea:

Avoiding time-consuming recomputations of solutions of subproblems by storing intermediate results and doing table look-ups.

LONGEST COMMON SUBSEQUENCE I

☞ Input: Two strings x and y.

☞ Task: Find a maximum length subsequence z of x and y.

Note: The longest common substring is contiguous, while the longest common subsequence needs not be.

Example Various applications in molecular biology, file comparison, screen display, ...

$x :=$ BDCABA and $y :=$ ABCBDAB $\rightsquigarrow z :=$ BCBA.

Longest Common Subsequence II

Use $x[1,i]$ to denote the ith prefix of x.

Define a table T of size $|x| \cdot |y|$, where $T[i,j]$ stores the length of the longest common subsequence of $x[1,i]$ and $y[1,j]$.

Compute T as follows:

$$
\begin{aligned}
T[i,j] &= 0, && \text{if } i \cdot j = 0 \\
T[i,j] &= T[i-1,j-1]+1, && \text{if } i \cdot j > 0 \text{ and } x_i = y_j \\
T[i,j] &= \max\{T[i,j-1], T[i-1,j]\}, && \text{if } i \cdot j > 0 \text{ and } x_i \neq y_j
\end{aligned}
$$

The length of z is then in $T[|x|,|y|]$; construct z by *traceback*.

TRAVELING SALESPERSON I

- ☞ Input: A set $\{1, 2, \ldots, n\}$ of "cities" with pairwise nonnegative distances $d(i, j)$, $1 \leq i, j \leq n$.
- ☞ Task: Find an order of the cities such that following this order each city is visited exactly once and the total distance traveled is minimized.

Trivial: Enumerate all possible $O(n!)$ tours.

Using dynamic programming leads to an $O(2^n \cdot n^2)$-time algorithm.

TRAVELING SALESPERSON II

Assume that the tour we search for may start in city 1.

Opt(S, i), for every non-empty subset $S \subseteq \{2, \ldots, n\}$ and every city $i \in S$, denotes the length of the shortest path that starts in city 1, then visits all cities in $S \setminus \{i\}$ in arbitrary order, and finally stops in city i.

Obviously, Opt$(\{i\}, i) = d(1, i)$.

Compute Opt(S, i) recursively:

$$\text{Opt}(S, i) = \min_{j \in S \setminus \{i\}} \{\text{Opt}(S \setminus \{i\}, j) + d(j, i)\}.$$

The optimal solution is given by

$$\min_{2 \leq j \leq n} \{\text{Opt}(\{2, \ldots, n\}, j) + d(1, j)\}.$$

Applicability of Dynamic Programming

Criteria for the applicability of dynamic programming?

- Optimal substructure: An optimal solution contains within it optimal solutions to subproblems.
- Overlapping subproblems: The same problem occurs as a subproblem of different problems.
- Independence: The solution of one subproblem does not affect the solution of an other subproblem of the same problem.

Scheme of Dynamic Programming

Dynamic programming's four steps:

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute in a bottom-up way the value of an optimal solution.

4. Construct an optimal solution from computed information.

In the following, by dynamic programming, we derive fixed-parameter algorithms.

Binary Knapsack I

- Input: A set of n items, each with a positive integer value v_i and a positive integer weight w_i, $1 \leq i \leq n$, and a positive integer bound W.
- Task: Find a subset of items such that their total value is maximized under the condition that their total weight does not exceed W.

Example

Input

Output

Binary Knapsack II

Dynamic programming for BINARY KNAPSACK:

Assume $w_i \leq W$ for all $1 \leq i \leq n$.

Define a table R of size $W \times n$, where $R[X, S]$, for an integer X with $1 \leq X \leq W$ and $S \subseteq \{1, \ldots, n\}$, stores the maximum possible value that can be achieved by a subset of S whose total weight is exactly X.

Step by step, consider $S = \emptyset$, $S = \{1\}$, $S = \{1, 2\}$, ..., $S = \{1, \ldots, n\}$.

Clearly, $R[X, \emptyset] = 0$ for all X.

Binary Knapsack III

The value of $R[X, S \cup \{i+1\}]$ is determined by

$$R[X, S \cup \{i+1\}] = \max\{R[X, S], R[X - w_{i+1}, S] + v_{i+1}\}.$$

The overall solution is in $R[W, \{1, \ldots, n\}]$.

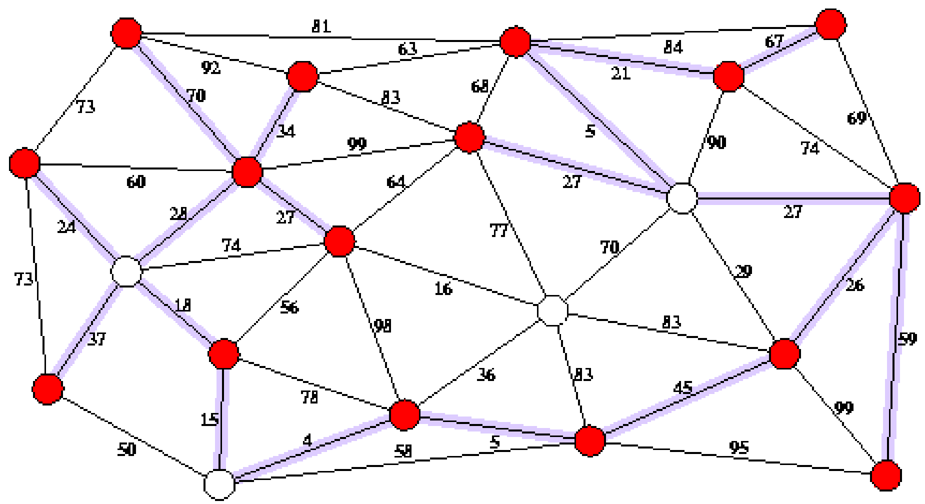
$\boxed{\text{Theorem}}$ Binary Knapsack can be solved in $O(W \cdot n)$ time.

Remark: This is not a polynomial-time algorithm, but it is a fixed-parameter algorithm with respect to the parameter "length of the binary encoding of W".

Steiner Problem in Graphs I

- Input: An edge-weighted graph $G = (V, E)$ and a set of *terminal vertices* $S \subseteq V$ with $|S| =: k$.
- Task: Find a subgraph G' of G that connects all vertices in S and whose total edge weight is minimum.

Note: G' can contain non-terminal vertices.



STEINER PROBLEM IN GRAPHS II

The original STEINER PROBLEM is from geometry: Find a set of lines of minimum total length which connect a given set S of points in the Euclidean plane.

Opposite to the geometric problem, now the triangle inequality may be invalid.

Easy to observe: The connecting subgraph G' must be a tree (*Steiner tree*).

Two simple special cases of STEINER PROBLEM IN GRAPHS

- $S = V$: The problem reduces to computing a minimum weight spanning tree of G and can be solved in polynomial time.
- $|S| = 2$: The problem reduces to computing a shortest path between two vertices and is polynomial-time solvable.

In the following, we consider the parameter $k := |S|$.

STEINER PROBLEM IN GRAPHS III

Idea:

The STEINER PROBLEM IN GRAPHS carries a *decomposition property*.

Compute the weight of a minimum Steiner tree for a given terminal set by considering the weights of the minimum Steiner trees of all proper subsets of this set.

Start with two-element subsets, where the Steiner tree is the shortest path.

Some notation (let $X \subseteq S$ and $v \in V \setminus X$):

$s(X \cup \{v\})$ denotes the weight of a minimum Steiner tree connecting all vertices from $X \cup \{v\}$ in G.

$p(u, v)$ denotes the total weight of the shortest path between vertices u and v.

Steiner Problem in Graphs IV

Lemma Let $X \neq \emptyset$, $X \subseteq S$, and $v \in V \setminus X$. Then,

$$s(X \cup \{v\}) = \min \left\{ \begin{array}{l} \min_{u \in X} \{s(X) + p(u,v)\}, \\ \min_{u \in V \setminus X} \{s_u(X \cup \{u\}) + p(u,v)\} \end{array} \right\},$$

where

$$s_u(X \cup \{u\}) := \min_{X' \neq \emptyset, X' \subseteq X} \{s(X' \cup \{u\}) + s((X \setminus X') \cup \{u\})\}.$$

STEINER PROBLEM IN GRAPHS V

Proof:

Assume T is a minimum Steiner tree for $X \cup \{v\}$. If v is a leaf in T, then define P_v as the longest path starting in v and in which all interior points have degree two in T. Distinguishing three cases and minimizing over all of them then gives the lemma:

- The case that v is no leaf in T is covered by setting $u = v$.
- The case that v is a leaf in T and P_v ends at a vertex $u \in X$ implies that

$$s(X \cup \{v\}) = s(X) + p(u, v).$$

- The case that v is a leaf and P_v ends at a vertex $u \in V \setminus X$ implies that T consists of a minimum Steiner tree for $X \cup \{u\}$ in which u has degree at least two (see the first case) and a shortest path from u to v.

Steiner Problem in Graphs VI

Theorem The Steiner Problem in Graphs can be solved in $O(3^k \cdot n + 2^k \cdot n^2 + n^2 \cdot \log n + n \cdot m)$ time, where $n := |V|$ and $m := |E|$.

Proof:

Initialization: $s(\{u, v\}) := p(u, v)$ for all $u, v \in S$.

Time: $O(n^2 \cdot \log n + n \cdot m)$ (using n times Dijkstra's shortest path algorithm which runs in $O(n \cdot \log n + m)$ time.).

Steiner Problem in Graphs VII

Proof of Theorem: (continued)

To compute $s_u(X \cup \{u\})$ in the above lemma for all X with $X \neq \emptyset$ and $X \subseteq S$ and all $u \in V \setminus X$, we need at most $n \cdot 3^k$ recursive calls: We have to consider all $X' \neq \emptyset$ and $X' \subseteq X$. The number of combinations can be upper-bounded by

$$\sum_{i=1}^{k} \binom{k}{i} \cdot \sum_{j=1}^{i-1} \binom{i}{j} \cdot n \leq n \cdot \sum_{i=1}^{k} 2^{i-1} \leq n \cdot 3^k.$$

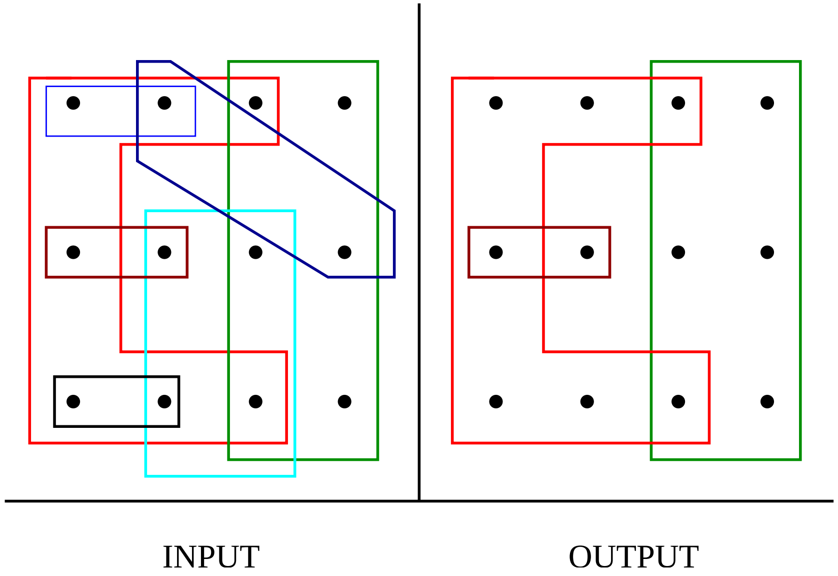
Each combination leads to two table look-ups. Hence, the running time for computing $s_u(X \cup \{u\})$ is $O(3^k \cdot n)$.

$s(X \cup \{v\})$ in the above lemma for all $X \neq \emptyset$, $X \subseteq S$, and $v \in V \setminus X$ can be computed in $O(3^k \cdot n + 2^k \cdot n^2)$ time.

TREE-LIKE WEIGHTED SET COVER I

SET COVER:

- Input: A base set $S = \{s_1, s_2, \ldots, s_n\}$ and a collection C of subsets of S, $C = \{c_1, c_2, \ldots, c_m\}$, $c_i \subseteq S$ for $1 \leq i \leq m$, and $\bigcup_{1 \leq i \leq m} c_i = S$.
- Task: Find a subset C' of C of minimum cardinality with $\bigcup_{c \in C'} c = S$.



WEIGHTED SET COVER: $w(c_i) \geq 0$ for $1 \leq i \leq m \rightsquigarrow$ minimize overall weight.

TREE-LIKE WEIGHTED SET COVER II

SET COVER is

- ✗ NP-complete.
- ✗ solvable in polynomial time if $|C_i| \leq 2$ for $1 \leq i \leq m$.
- ✗ polynomial-time $\Theta(\log n)$-approximable.
- ✗ likely to be fixed-parameter intractable with respect to the parameter $|C'|$.

TREE-LIKE WEIGHTED SET COVER III

Definition [Tree-like subset collection]

A subset collection C is called a *tree-like* subset collection of S if the subsets in C can be organized in a tree T such that

- *every* subset one-to-one corresponds to a node of T and,
- for each element $s \in S$, all nodes in T corresponding to the subsets in C containing s induce a *subtree* of T.

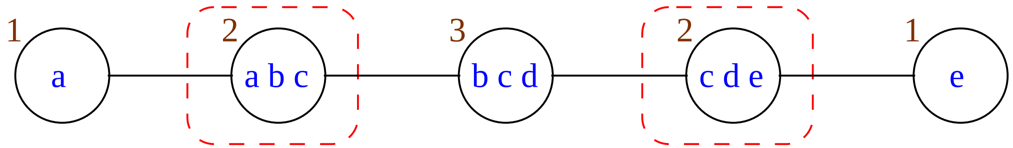
Tree T is called the underlying *subset tree*.

Note: It can be tested in linear time whether or not a subset collection is tree-like.

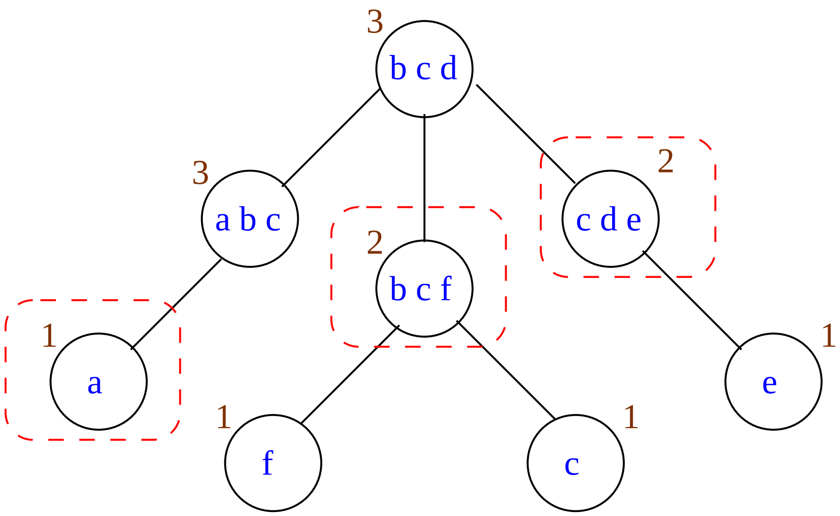
TREE-LIKE WEIGHTED SET COVER (TWSC) is the same as WEIGHTED SET COVER restricted to a tree-like subset collection.

TREE-LIKE WEIGHTED SET COVER IV

Example: Path-like subset collection



Example: Tree-like subset collection



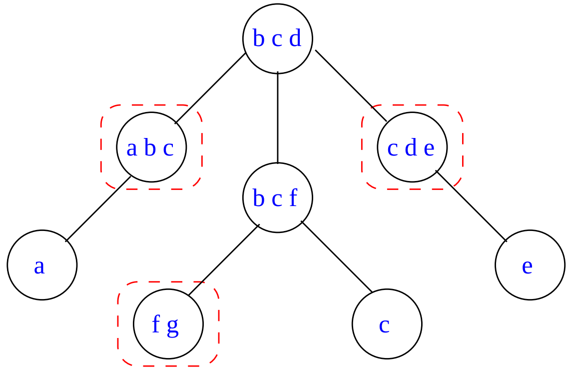
Motivation: Memory saving in tree decomposition based dynamic programming, locating gene duplications, ...

Tree-like Weighted Set Cover V

Tree-like Unweighted Set Cover can be solved in polynomial time by a simple bottom-up algorithm.

Decisive observation:

Lemma Given a tree-like subset collection C of S together with its underlying subset tree T, then each leaf of T is either a subset of its parent node or there exists an element of S which appears only in this leaf.



TREE-LIKE WEIGHTED SET COVER VI

In contrast to the unweighted case, TREE-LIKE WEIGHTED SET COVER is

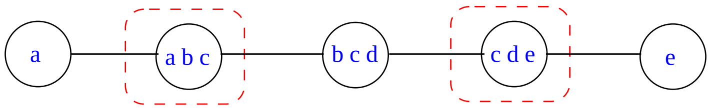
- NP-complete even if each element appears in at most three subsets in C.
- hard to approximate (best approximation factor: $\Theta(\log n)$).
- likely to be fixed-parameter *intractable* with respect to the parameter *overall weight of the set cover*.

We consider *the maximum subset size* as parameter, that is, $k := \max_{c \in C} |c|$.

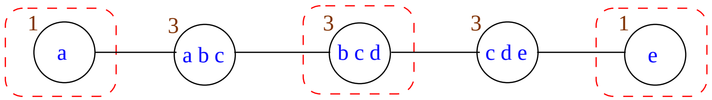
TREE-LIKE WEIGHTED SET COVER VII

Warmup Dynamic programming for PATH-LIKE WEIGHTED SET COVER.

Unweighted case:



Weighted case:



Define for each subset $c_i \in C$:

$$A(c_i) := c_1 \cup c_2 \cup \cdots \cup c_i,$$

$$B(c_i) := \text{minimum weight to cover } A(c_i) \text{ by using only subsets from } c_1, \ldots, c_i.$$

Clearly, $A(c_m) = S$ and $B(c_m)$ stores the overall solution.

TREE-LIKE WEIGHTED SET COVER VIII

Assume that each element appears in at least two subsets.

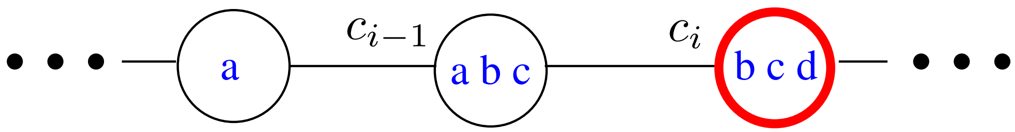
Process the subsets from left to right. Introduce $c_0 := \emptyset$ with $w(c_0) := 0$, $A(c_0) := \emptyset$, and $B(c_0) := 0$.

Initialization: Trivial. $A(c_1) := \{c_1\}$ and $B(c_1) := w(c_1)$.

Main part: Assume that $A(c_j)$ and $B(c_j)$ are computed for all $1 \leq j \leq i - 1$.
Clearly, $A(c_i) := A(c_{i-1}) \cup \{c_i\}$. To compute $B(c_i)$, distinguish two cases.

TREE-LIKE WEIGHTED SET COVER IX

Case 1. $c_i \nsubseteq c_{i-1}$.

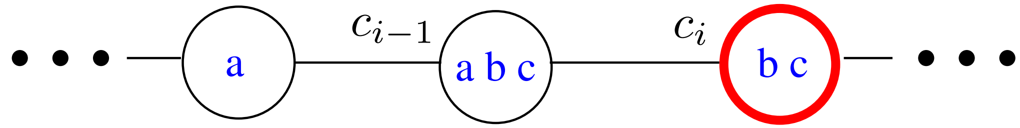


Then $\exists s \in c_i$ with $s \notin A(c_{i-1})$. To cover $A(c_i)$, we have to take c_i. Hence,

$$B(c_i) := w(c_i) + \min_{0 \leq l \leq i-1} \{B(c_l) \mid (A(c_i) \setminus \{c_i\}) \subseteq A(c_l)\}.$$

Tree-like Weighted Set Cover X

Case 2. $c_i \subseteq c_{i-1}$.



Then $A(c_i) = A(c_{i-1})$. We compare two alternatives to cover $A(c_i)$, to take c_i or not. Thus,

$$B(c_i) := \min \left\{ \begin{array}{l} B(c_{i-1}), \\ w(c_i) + \min_{0 \le l \le i-1} \{ B(c_l) \mid (A(c_i) \setminus \{c_i\}) \subseteq A(c_l) \} \end{array} \right\}.$$

By *traceback*, one can easily construct a minimum set cover.

Theorem Path-like Weighted Set Cover can be solved in $O(m^2 \cdot n)$ time.

Tree-like Weighted Set Cover XI

Extend the dynamic programming to the tree-like case:

Root the subset tree T at an arbitrary node. Process the nodes bottom-up.

Modify the definition of $A(c_i)$:

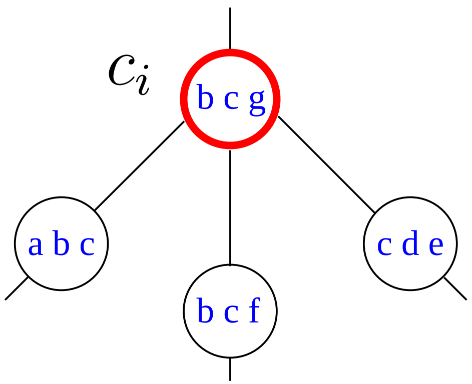
$$A(c_i) := \bigcup_{c \in T[c_i]} c,$$

where $T[c_i]$ denotes the node set of the subtree of T rooted at c_i.

Tree-like Weighted Set Cover XII

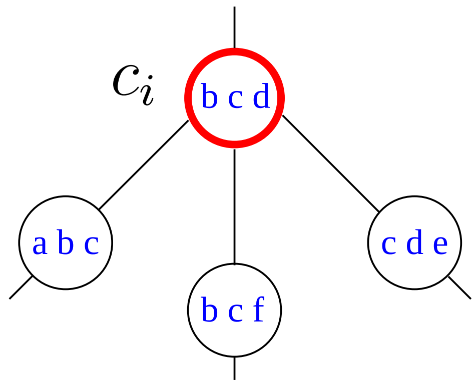
Key point: nodes with degree ≥ 3.

Case 1:



Take c_i.

Case 2:



Many alternatives.

Tree-like Weighted Set Cover XIII

Idea

For each subset c_i, define a size-$(3 \cdot 2^{|c_i|})$ table D_{c_i} instead of $B(c_i)$. Table D_{c_i} stores, for all possible subsets x of c_i, the minimum weight to cover $A(c_i) \setminus x$ with only subsets in $T[c_i]$.

With some effort, the following can be shown:

Using tables D_{c_i} to store the intermediate results and a similar dynamic programming approach as in the unweighted case, one can solve Tree-like Weighted Set Cover in $O(3^k \cdot m \cdot n)$ time.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

❷ Algorithmic methods

- ➯ Data reduction and problem kernels
- ➯ Depth-bounded search trees
- ➯ Some advanced techniques
- ➯ Dynamic programming
- ➯ Tree decompositions of graphs

❸ Parameterized complexity theory

Tree Decompositions of Graphs I

Intuition / Motivation

Many hard graph problems turn easy when restricted to trees, for instance, VERTEX COVER and DOMINATING SET.

Idea: When restricted to a "tree-like" graph, a problem should be easier than in general graphs.

The concept of tree decompositions of graphs provides a measure of the tree-likeness of graphs.

Using tree decompositions of graphs, we can generalize dynamic programming algorithms solving graph problems in trees to algorithms solving graph problems in tree-like graphs, i.e., graphs with "bounded treewidth".

Tree Decompositions of Graphs II

Definition Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair $< \{X_i \mid i \in I\}, T >$ where each X_i is a subset of V, called a *bag*, and T is a tree with the elements of I as nodes. The following three properties must hold:

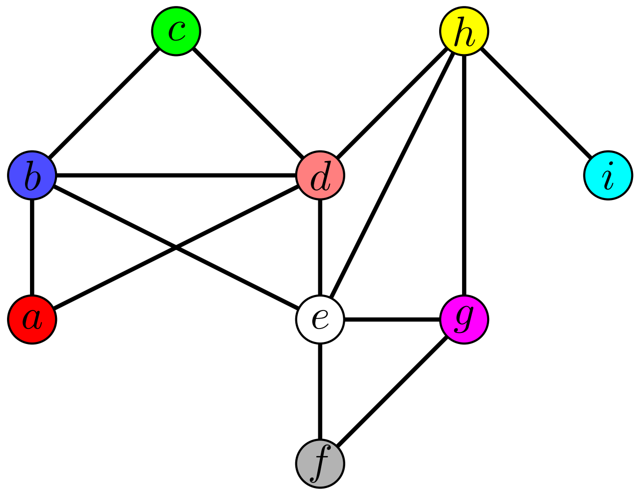
1. $\bigcup_{i \in I} X_i = V$;
2. for every edge $\{u, v\} \in E$, there is an $i \in I$ such that $\{u, v\} \subseteq X_i$;
3. for all $i, j, l \in I$, if j lies on the path between i and l in T, then $X_i \cap X_l \subseteq X_j$.

The *width* of $< \{X_i \mid i \in I\}, T >$ equals $\max\{|X_i| \mid i \in I\} - 1$. The *treewidth* of G is the minimum k such that G has a tree decomposition of width k.

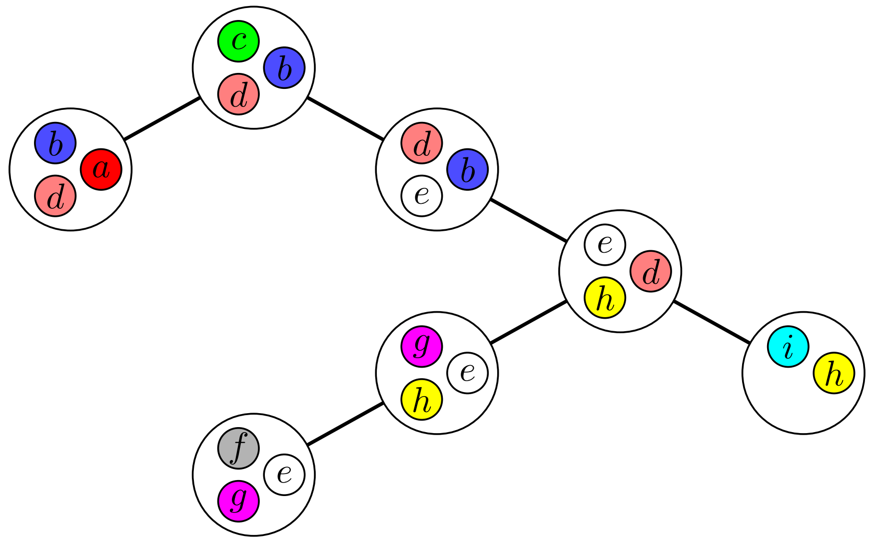
Tree Decompositions of Graphs III

Example

Graph G:



A tree decomposition of G:



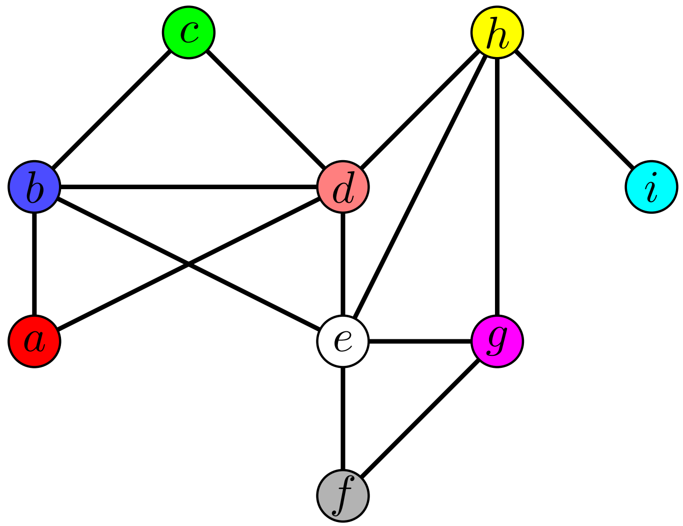
Tree Decompositions of Graphs IV

Remarks:

- ✗ A tree has treewidth 1. A clique of n vertices has treewidth $n-1$.
- ✗ The smaller is the treewidth of a graph, the more tree-like is the graph.
- ✗ There are several equivalent notions for tree decompositions; among others, graphs of treewidth at most k are known as *partial k-trees*.
- ✗ In general, it is NP-hard to compute an optimal tree decomposition of a graph.

Tree Decompositions of Graphs V

A very useful and intuitively appealing characterization of tree decompositions in terms of a game: *robber-cop game*.



The treewidth of a graph is the minimum number of cops needed to catch a robber minus one.

Tree Decompositions of Graphs VI

Typically, tree decomposition based algorithms proceed according to the following scheme in two stages:

1. Find a tree decomposition of bounded width for the input graph;
2. Solve the problem by *dynamic programming on the tree decomposition*.

In the following, we describe how the second stage works.

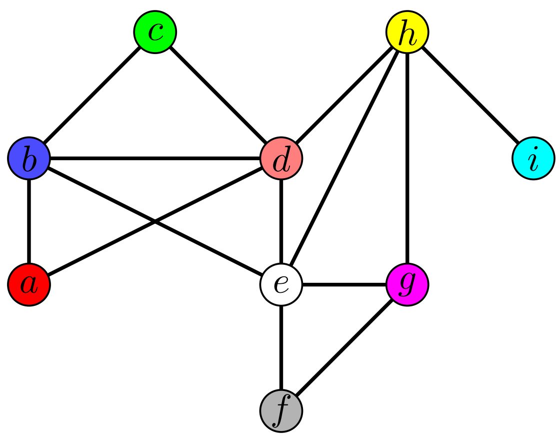
The running time and the memory consumption of the dynamic programming on a given tree decomposition grow exponentially in the treewidth.

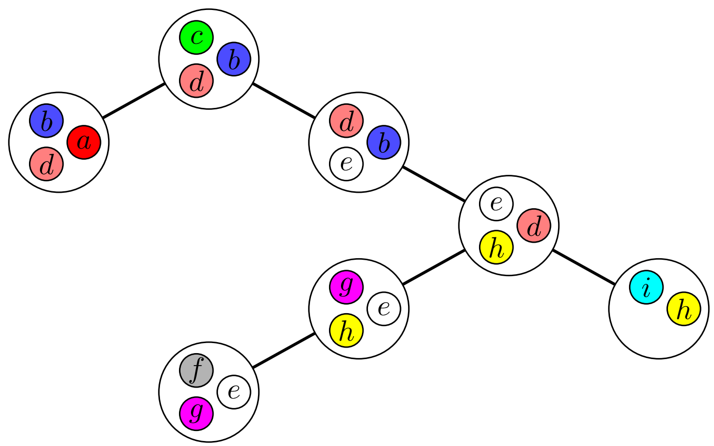
$\leadsto$ only efficient for small treewidths.

VERTEX COVER I

Example application to VERTEX COVER:

- Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- Task: Find a subset of vertices $V' \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in V'.





VERTEX COVER II

Theorem For a graph $G = (V, E)$ with a given tree decomposition $< \{X_i \mid i \in I\}, T >$, an optimal vertex cover can be computed in $O(2^\omega \cdot \omega \cdot |I|)$ time. Here, ω denotes the width of the tree decomposition.

Proof: For each bag X_i for $i \in I$, check all of the at most $2^{|X_i|}$ possibilities to obtain a vertex cover for the subgraph $G[X_i]$ of G induced by the vertices from X_i. (This information is stored in a table A_i.)

Adjacent tables will be updated in a bottom-up process (from the leaves to the root).

During this update process, the “local” solutions for each subgraph are combined into a “globally optimal” solution for the overall graph G.

VERTEX COVER III

Proof of Theorem: (continued)

Step 0: $\forall X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$, $|X_i| = n_i$, create

$$A_i \;=\; \left.\begin{array}{ccccc||c} x_{i_1} & x_{i_2} & \cdots & x_{i_{n_i-1}} & x_{i_{n_i}} & m \\ \hline 0 & 0 & \cdots & 0 & 0 & \\ 0 & 0 & \cdots & 0 & 1 & \\ & & \vdots & & & \\ & & \vdots & & & \\ 1 & 1 & \ldots & 1 & 0 & \\ 1 & 1 & \ldots & 1 & 1 & \end{array}\right\} 2^{n_i}$$

VERTEX COVER IV

Proof of Theorem: (continued)

Each row represents a so-called "coloring" of $G[X_i]$, i.e.,

$$C_i \, : \, X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\} \rightarrow \{0, 1\}.$$

Interpretation:

- $C_i(x) = 0$: Vertex x not in vertex cover.
- $C_i(x) = 1$: Vertex x in vertex cover.

VERTEX COVER V

Proof of Theorem: (continued)

The last column stores for each coloring C_i the number of vertices which a minimal vertex cover containing those vertices from X_i selected by C_i would need, that is,

$$\begin{aligned} m(C_i) \quad = \quad & \min\left\{\, |V'| : V' \subseteq V \text{ is a vertex cover for } G\text{, such that} \right. \\ & \forall v \in (C_i)^{-1}(1) : \; v \in V' \quad \text{and} \\ & \left. \forall v \in (C_i)^{-1}(0) : \; v \notin V' \right\}. \end{aligned}$$

VERTEX COVER VI

Proof of Theorem: (continued)

Not every possible coloring may lead to a vertex cover. Such a coloring is called *invalid*. To check whether a coloring is *valid*:

bool is_valid (coloring $C_i : X_i \rightarrow \{0, 1\}$)

result = **true**;

for $(e = \{u, v\} \in E_{G[X_i]})$

  if $(C_i(u) = 0 \wedge C_i(v) = 0)$ **then** result = **false**;

Vertex Cover VII

Proof of Theorem: (continued)

Step 1: (Table initialization)

For all bags X_i and each coloring $C_i : X_i \to \{0, 1\}$, set

$$m(C_i) := \begin{cases} \left| (C_i)^{-1}(1) \right|, & \text{if (is\_valid } (C_i)) \\ +\infty, & \text{otherwise} \end{cases}$$

Vertex Cover VIII

Proof of Theorem: (continued)

Step 2: (Dynamic programming)

We now go through the tree T from the leaves to the root and compare the corresponding tables against each other.

Let $i \in I$ be the parent node of $j \in I$. We show how the table for X_i can be updated by the table for X_j.

Assume that

$$\begin{aligned} X_i &= \{z_1, \ldots, z_s, u_1, \ldots, u_{t_i}\} \\ X_j &= \{z_1, \ldots, z_s, v_1, \ldots, v_{t_j}\}, \end{aligned}$$

where $X_i \bigcap X_j = \{z_1, \ldots, z_s\}$.

Vertex Cover IX

Proof of Theorem: (continued)

A coloring $\tilde{C} : \tilde{W} \to \{0,1\}$ is an *extension* of a coloring $C : W \to \{0,1\}$ if $W \subseteq \tilde{W} \subseteq V$ and $\tilde{C}$ restricted to ground set W yields C, that is, $\tilde{C}|_W = C$.

For each possible coloring

$$C : \{z_1, \ldots, z_s\} \to \{0,1\}$$

and each extension $C_i : X_i \to \{0,1\}$ of C, set

$$\begin{aligned} m_i(C_i) &:= m_i(C_i) \\ &+ \min\left\{ m_j(C_j) \mid C_j : X_j \to \{0,1\} \text{ is an extension of } C \right\} \\ &- \left|C^{-1}(1)\right| \end{aligned}$$

Vertex Cover X

Proof of Theorem: (continued)

Step 3: (Construction of a minimum vertex cover V')

The size of V' is derived from the minimum entry of the last column of the root table A_r. The coloring of the corresponding row shows which of the vertices of X_r are contained in V'. By *traceback*, one can easily compute all vertices of V'.

Vertex Cover XI

Proof of Theorem: (continued)

Correctness of the algorithm:

1. The first condition in the definition of tree decompositions, $V = \bigcup_{i \in I} X_i$, makes sure that every graph vertex is taken into account during the computation.

2. The second condition in the definition of tree decompositions, that is, $\forall e \in E,\ \exists i \in I : e \subseteq X_i$, makes sure that after the treatment of invalid colorings in Step 1 (table initialization) only actual vertex covers are dealt with.

3. The third condition in the definition of tree decompositions guarantees the consistency of the dynamic programming.

VERTEX COVER XII

Proof of Theorem: (continued)

Running time of the algorithm:

The comparison of a table A_j against a table A_i can be done in time proportional to the maximum table size, that is,

$$2^{\omega} \cdot \omega.$$

For each edge in the decomposition tree T, such a comparison has to be done, that is, the overall running time is

$$O(2^{\omega} \cdot \omega \cdot |I|).$$

DOMINATING SET I

- ☞ Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- ☞ Task: Find a subset $S \subseteq V$ with at most k vertices such that every vertex $v \in V$ is contained in S or has at least one neighbor in S.

More elusive than VERTEX COVER $\leadsto$ larger overhead needed to be solved via dynamic programming on tree decompositions.

DOMINATING SET II

Recall: For VERTEX COVER we use a 2-coloring C_i to determine which of the bag vertices from bag X_i should be in the vertex cover.

Here, we use a 3-coloring

$$C_i : X_i = \{x_{i_1}, \ldots, x_{i_n}\} \rightarrow \{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\},$$

where

- $\textcolor{red}{\bullet}$ means that the vertex is in the dominating set;
- $\textcolor{blue}{\bullet}$ means that the vertex is already dominated at the current stage of the algorithm;
- $\textcolor{yellow}{\bullet}$ means that, at the current stage of the algorithm, one still asks for a domination of the vertex.

DOMINATING SET III

For each bag X_i with $|X_i| = n_i$, we define a mapping

$$m_i : \{{\color{red}\bullet}, {\color{blue}\bullet}, {\color{yellow}\bullet}\}^{n_i} \rightarrow \mathbb{N} \cup \{+\infty\}.$$

For a coloring C_i, $m_i(C_i)$ stores how many vertices are needed for a minimum dominating set of the graph visited up to the current stage of the algorithm under the restriction that the color assigned to vertex x_{i_t} is $C_i(x_{i_t})$, $t = 1, \ldots, n_i$.

Dominating Set IV

With $m_i : \{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}^{n_i} \to \mathbb{N} \cup \{+\infty\}$, we end up with tables of size 3^{n_i}.

It is not difficult to come up with an algorithm for Dominating Set with a given width-ω and $|I|$-nodes tree decomposition in $O(9^\omega \cdot \omega \cdot |I|)$ time: Comparing two tables for bags X_i and X_j now takes

$$O(3^{|X_i|} \cdot 3^{|X_j|} \cdot \max\{|X_i|, |X_j|\}) = O(9^\omega \cdot \omega)$$

time.

There is room for improvement!

Dominating Set V

In the following, we give an algorithm running in $O(4^{\omega} \cdot \omega \cdot |I|)$ time.

Comparing the $O(4^{\omega} \cdot \omega \cdot |I|)$ algorithm with the $O(9^{\omega} \cdot \omega \cdot |I|)$ algorithm for $|I| = 1000$; assume a computer executing 10^9 instructions per second.

Running time	$\omega = 5$	$\omega = 10$	$\omega = 15$	$\omega = 20$
$9^{\omega} \cdot \omega \cdot \lvert I \rvert$	0.25 sec	10 hours	100 years	$8 \cdot 10^6$ years
$4^{\omega} \cdot \omega \cdot \lvert I \rvert$	0.005 sec	10 sec	4.5 hours	260 days

DOMINATING SET VI

Idea

Define a *partial ordering* $\prec$ for the colorings C_i of X_i and show that the mapping m_i is a *monotonous* function from $(\{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}^{n_i}, \prec)$ to $(\mathbb{N} \cup \{+\infty\}, \leq)$.

By using the monotonicity of m_i, we can “save” some comparisons during the dynamic programming step.

Dominating Set VII

For {🔴, 🔵, 🟡}, define partial ordering $\prec$ by

🟡 $\prec$ 🔵 and $d \prec d$ for all $d \in$ {🔴, 🔵, 🟡}.

Extend $\prec$ to colorings:

For $c = (c_1, \ldots, c_{n_i}), c' = (c'_1, \ldots, c'_{n_i}) \in$ {🔴, 🔵, 🟡}^{n_i},

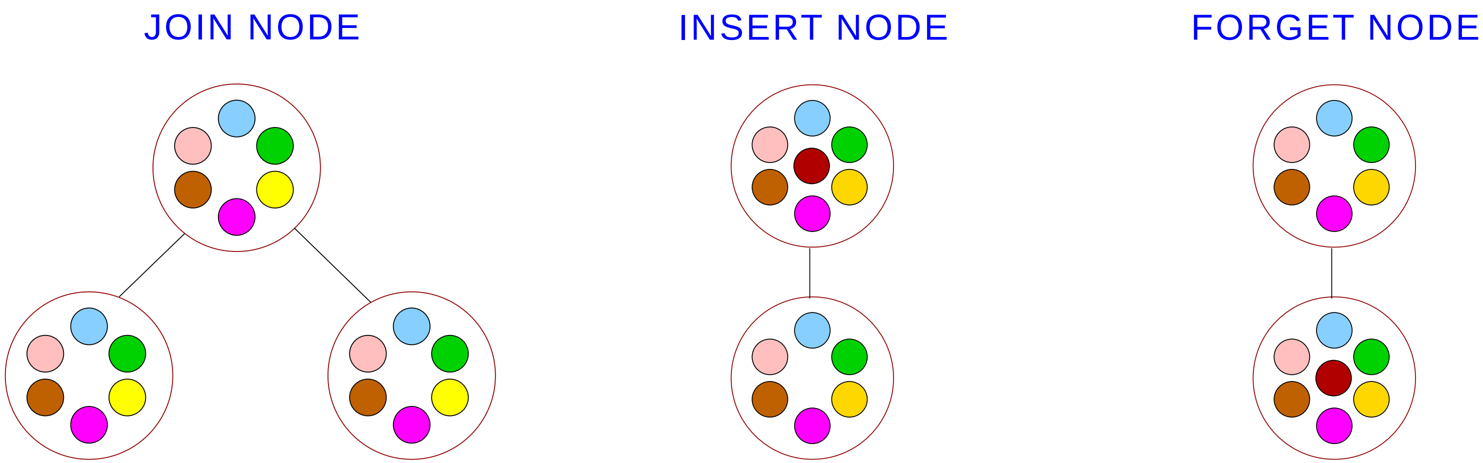
let $c \prec c'$ iff $c_t \prec c'_t$ for all $t = 1, \ldots, n_i$.

Mapping m_i is a monotonous function from ({🔴, 🔵, 🟡}$^{n_i}, \prec$) to $(\mathbb{N} \cup \{+\infty\}, \leq)$ if $c \prec c'$ implies that $m_i(c) \leq m_i(c')$.

Dominating Set VIII

To make things easier, we work with a tree decomposition with a simple structure:

Definition A tree decomposition $< \{X_i \mid i \in I\}, T >$ is called a *nice tree decomposition* if every node of the tree T has at most two children and all inner nodes is of one of the following types:



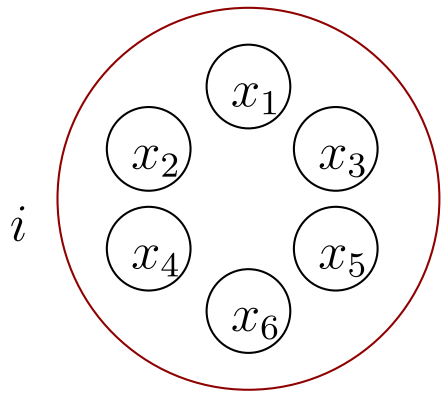
Remark: A width-k and n-nodes tree decomposition can be transformed in $O(n)$ time into a width-k and $O(n)$-nodes nice tree decomposition.

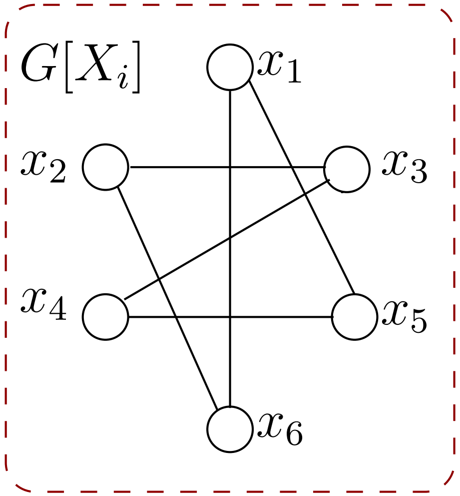
Dominating Set IX

Initialization

Only for leaf nodes. Let i denote a leaf node with bag X_i and $G[X_i]$ is the subgraph of the input graph G induced by the vertices in X_i.

Example:

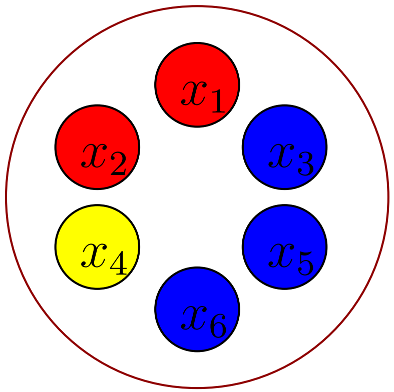


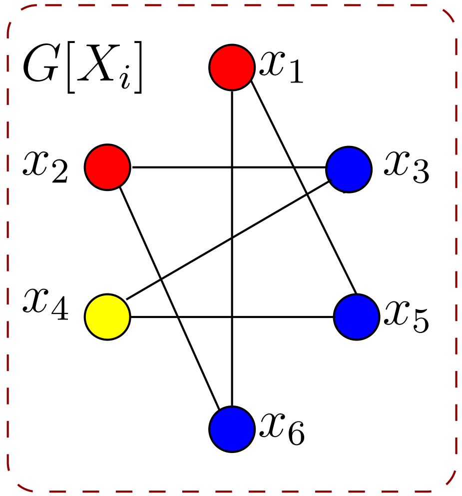


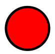
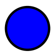
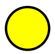
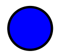
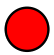
DOMINATING SET X

Initialization (continued)

i

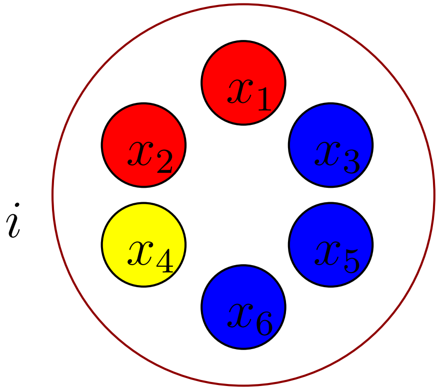


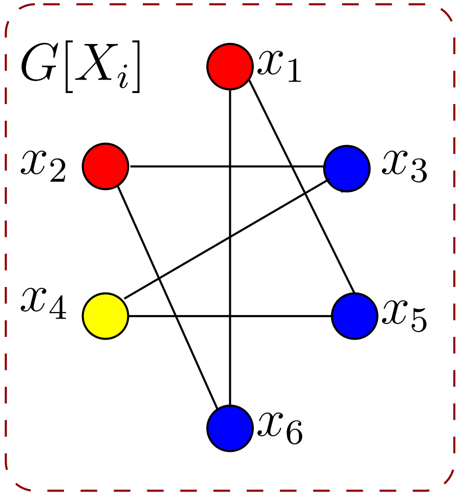


A coloring $c \in \{$, , $\}^{n_i}$ is *locally valid* for a bag X_i, if every -colored vertex in X_i has a -colored neighbor in $G[X_i]$; otherwise, c is *locally invalid*.

Dominating Set XI

Initialization (continued)



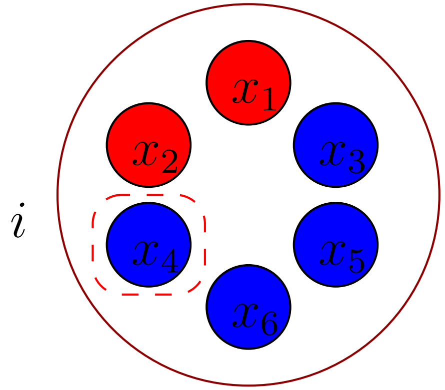


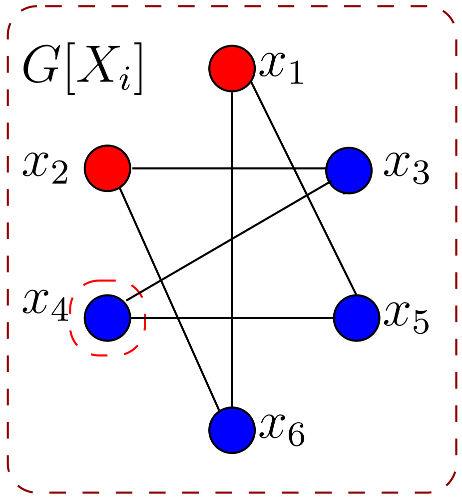
For a *locally valid* coloring $c \in \{$, , $\}^{n_i}$, set

$$m_i(c) := \text{number of } .$$

DOMINATING SET XII

Initialization (continued)





For a *locally invalid* coloring $c \in \{$🔴, 🔵, 🟡$\}^{n_i}$, set

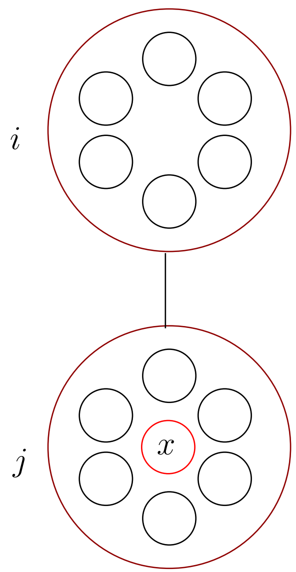
$$m_i(c) := +\infty.$$

Time complexity: $O(3^{|X_i|} \cdot |X_i|)$.

DOMINATING SET XIII

Dynamic Programming

FORGET NODES



$\rightsquigarrow$ Vertex x has to be $\color{red}{\bullet}$ (in the dominating set) or $\color{blue}{\bullet}$ (already dominated in bag j).

For a coloring $c \in \{\color{red}{\bullet}, \color{blue}{\bullet}, \color{yellow}{\bullet}\}^{n_i}$, set

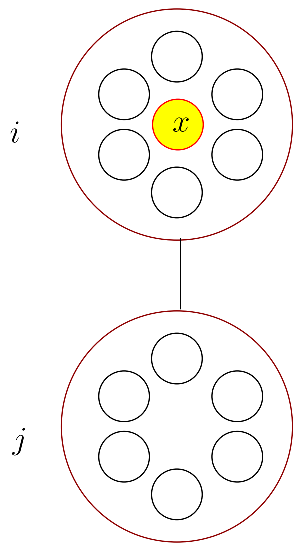
$$m_i(c) := \min_{d \in \{\color{red}{\bullet}, \color{blue}{\bullet}\}} \{m_j(c \times \{d\})\}.$$

Time complexity: $O(3^{|X_i|} \cdot |X_i|)$.

DOMINATING SET XIV

Dynamic Programming

INSERT NODES $X_i = \{x_{j_1}, \ldots, x_{j_{n_j}}, x\}$.



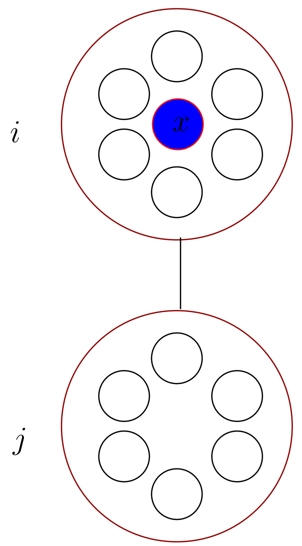
For a coloring $c \in \{$🔴, 🔵, 🟡$\}^{n_j}$, if x is colored as 🟡, then set

$$m_i(c \times \{🟡\}) := m_j(c).$$

DOMINATING SET XV

Dynamic Programming

INSERT NODES (continued) $X_i = \{x_{j_1}, \ldots, x_{j_{n_j}}, x\}$.



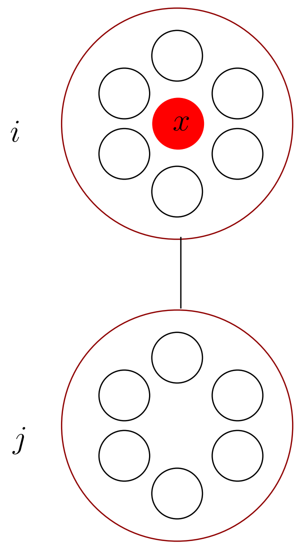
For a coloring $c \in \{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}^{n_j}$, if x is colored as $\textcolor{blue}{\bullet}$, then set

$$m_i(c \times \{\textcolor{blue}{\bullet}\}) := \begin{cases} m_j(c), & \text{if } x \text{ has a neighbor } x_t \text{ in } X_i \\ & \text{with } c_t = \textcolor{red}{\bullet},\ 1 \leq t \leq n_j \\ +\infty, & \text{otherwise.} \end{cases}$$

DOMINATING SET XVI

Dynamic Programming

INSERT NODES (continued) $X_i = \{x_{j_1}, \ldots, x_{j_{n_j}}, x\}$.



For a coloring $c \in \{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}^{n_j}$, if x is colored as $\textcolor{red}{\bullet}$, then set

$$m_i(c \times \{\textcolor{red}{\bullet}\}) := m_j(c') + 1, \qquad \text{(monotonicity of } m_j\text{)}$$

where, for $1 \le t \le n_j$, $c' = (c'_1, \ldots, c'_{n_j})$ and

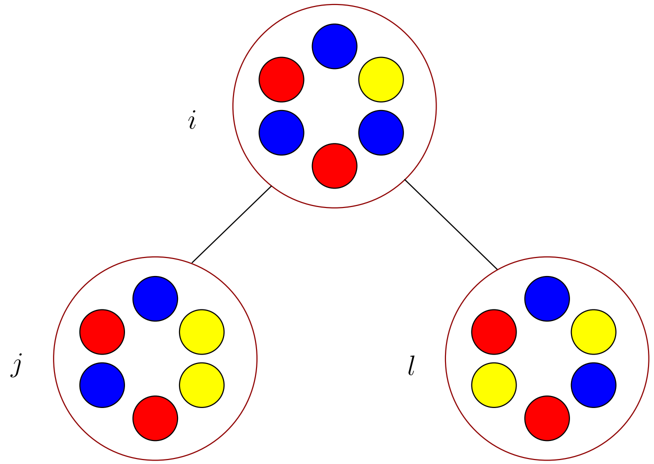
$$c'_t := \begin{cases} \textcolor{yellow}{\bullet}, & \text{if } x_t \text{ is a neighbor of } x \text{ and } c_t \neq \textcolor{red}{\bullet}, \\ c_t, & \text{otherwise.} \end{cases}$$

For all three colorings of x, time complexity: $O(3^{|X_i|} \cdot |X_i|)$.

DOMINATING SET XVII

Dynamic Programming

JOIN NODES $X_i = X_j = X_l$.



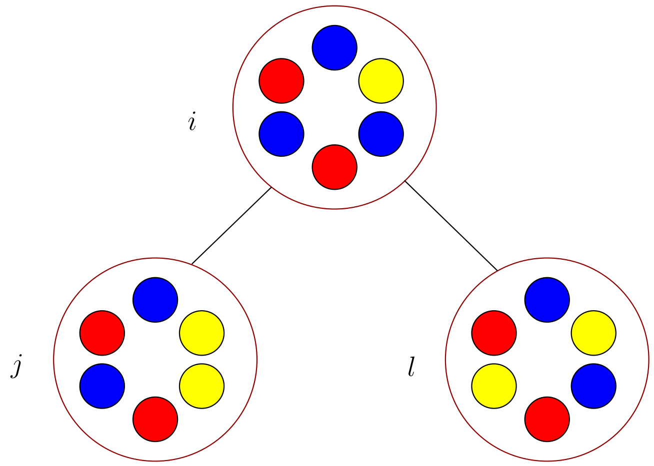
For colorings $c, c', c'' \in \{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}^{n_i}$, c' and c'' *divide* c if, for $1 \leq t \leq n_i$,

1. $c_t \in \{\textcolor{red}{\bullet}, \textcolor{yellow}{\bullet}\} \Rightarrow$
$$c'_t = c''_t = c_t,$$
2. $c_t = \textcolor{blue}{\bullet} \Rightarrow$
$$(c'_t, c''_t \in \{\textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}) \wedge (c'_t = \textcolor{blue}{\bullet} \vee c''_t = \textcolor{blue}{\bullet}).$$

DOMINATING SET XVIII

Dynamic Programming

JOIN NODES (continued) $X_i = X_j = X_l$.



For all colorings $c \in \{{\color{red}\bullet}, {\color{blue}\bullet}, {\color{yellow}\bullet}\}^{n_i}$, set

$$m_i(c) \quad := \quad \min\{m_j(c') + m_l(c'') - \#_{\color{red}\bullet}(c) : c' \text{ and } c'' \text{ divide } c\},$$

where $\#_{\color{red}\bullet}(c)$ denotes the number of the vertices of X_i that are ${\color{red}\bullet}$-colored by c.

Dominating Set XIX

Dynamic Programming

Join nodes (continued) $X_i = X_j = X_l$.

Time complexity:

For a fixed coloring $c \in \{{\color{red}\bullet}, {\color{blue}\bullet}, {\color{yellow}\circ}\}^{n_i}$, the decisive factor is the number of coloring pairs c' and c'' dividing c. From the monotonicity of m_i we can replace condition 2 in the definition of "divide" by

$$2'.\ \ c_t = {\color{blue}\bullet} \Rightarrow (c'_t, c''_t \in \{{\color{blue}\bullet}, {\color{yellow}\circ}\}) \wedge (c'_t \neq c''_t).$$

Then for a given c with $z := \#_{\color{blue}\bullet}(c)$, there are 2^z many pairs (c', c'') that divide c.

Dominating Set XX

Dynamic Programming

JOIN NODES (continued) $X_i = X_j = X_l$.

Since there are $2^{n_i - z} \cdot \binom{n_i}{z}$ many colorings c with $\#_{\textcolor{blue}{\bullet}}(c) = z$, we obtain that

$$\{(c', c'') \mid c \in \{\textcolor{red}{\bullet}, \textcolor{blue}{\bullet}, \textcolor{yellow}{\bullet}\}^{n_i}, c' \text{ and } c'' \text{divide } c\}$$

has size

$$\sum_{z=0}^{n_i} 2^{n_i - z} \cdot \binom{n_i}{z} \cdot 2^z = 4^{n_i}.$$

Thus, evaluating m_i for a join node i can be done in $O(4^{n_i} \cdot n_i)$ time.

DOMINATING SET XXI

The overall minimum domination number is then

$$\min\{m_r(c) \mid c \in \{{\color{red}\bullet}, {\color{blue}\bullet}\}^{n_r}\},$$

where r denotes the root of T.

$\boxed{\text{Theorem}}$ For a graph $G = (V, E)$ with a given tree decomposition $< \{X_i \mid i \in I\}, T >$, an optimal dominating set can be computed in $O(4^\omega \cdot \omega \cdot |I|)$ time. Here, ω denotes the width of the tree decomposition.

Concluding Remarks

- ✗ *Monadic second-order logic* is a powerful tool to decide whether a problem is fixed-parameter tractable on graphs parameterized by treewidth; but the associated running times suffer from huge constant.

- ✗ Treewidth and related concepts offer a new view on parameterization, *structural parameterization*.

- ✗ The *efficient usage of memory* seems to be the bottleneck for implementing tree decomposition based algorithms. One of the first approach solving the memory problem: TREE-LIKE WEIGHTED SET COVER.

Fixed-Parameter Algorithms

❶ Basic ideas and foundations

❷ Algorithmic methods

- ➪ Data reduction and problem kernels
- ➪ Depth-bounded search trees
- ➪ Some advanced techniques
- ➪ Dynamic programming
- ➪ Tree decompositions of graphs

❸ Parameterized complexity theory

Parameterized Complexity Theory

In the following, we present some basic definitions and concepts that lead to a theory of “intractable” parameterized problems.

Overview

☛ Parameterized reduction

☛ Parameterized complexity classes

☛ Complete problems and W-hierarchy

☛ Structural results

- Connection to classical complexity theory
- Connection to approximation

Parameterized Reduction I

Compared to the classical many-to-one reductions, parameterized reductions are more fine-grained and technically more difficult.

Definition Let $L, L' \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. We say L reduces to L' by a standard *parameterized (many-to-one) reduction* if there are functions $k \mapsto k'$ and $k \mapsto k''$ from $\mathbb{N}$ to $\mathbb{N}$ and a function $(x, k) \mapsto x'$ from $\Sigma^* \times \mathbb{N}$ to Σ^* such that

1. $(x, k) \mapsto x'$ is computable in $k'' \cdot |(x, k)|^c$ time for some constant c and
2. $(x, k) \in L$ iff $(x', k') \in L'$.

Parameterized Reduction II

Consider a parameterized variant of the famous SAT problem:

WEIGHTED (3)CNF-SATISFIABILITY

- ☞ Input: A boolean formula F in conjunctive normal form (with maximum clause size three) and a nonnegative integer k.
- ☞ Question: Is there a satisfying truth assignment for F which has weight exactly k, that is, an assignment with exactly k variables set TRUE?

Parameterized Reduction III

Reduction from CNF-SATISFIABILITY to 3CNF-SATISFIABILITY:

The central idea: Replace a clause

$$(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_m)$$

by the expression

$$(\ell_1 \vee \ell_2 \vee z_1) \wedge (\overline{z_1} \vee \ell_3 \vee z_2) \wedge \ldots \wedge (\overline{z_{m-3}} \vee \ell_{m-1} \vee \ell_m),$$

where $z_1, z_2, \ldots, z_{m-3}$ denote newly introduced variables.

It is easy to verify that

> an original CNF-formula is satisfiable iff the 3-CNF formula constructed by replacing all its size-at-least-four clauses in the above way is satisfiable.

Parameterized Reduction IV

But:

> The reduction from CNF-SATISFIABILITY to 3CNF-SATISFIABILITY is **not** a parameterized one!

Assume that the original CNF-formula has a weight-k satisfying truth assignment, making *exactly* one literal ℓ_j in clause $(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_m)$ TRUE. To satisfy the corresponding 3-CNF formula associated with $(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_m)$, however, one has to set all variables $z_1, z_2, \ldots, z_{j-2}$ to TRUE.

$\leadsto$ The weight of a satisfying truth assignment for the constructed 3-CNF formula does not *exclusively* depend on the original parameter k!

Parameterized Reduction V

Consider the graph problems, VERTEX COVER, INDEPENDENT SET, and CLIQUE.

- Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
- Question of VERTEX COVER: Is there a subset C of V with $|C| \leq k$ such that each edge in E has at east one of its endpoints in C?
- Question of INDEPENDENT SET: Is there a vertex subset I with at least k vertices that form an independent set, that is, I induces an edgeless subgraph of G?
- Question of CLIQUE: Is there a vertex subset S with at least k vertices such that S forms a clique, that is, S induces a complete subgraph of G?

Parameterized Reduction VI

A reduction from Vertex Cover to Independent Set:

> A graph has a size-k vertex cover iff it has a size-$(n-k)$ independent set.

This reduction is not a parameterized reduction: Whereas Vertex Cover has parameter value k, Independent Set receives parameter value $n-k$, a value that does not *exclusively* depend on k but also on the number n of vertices in the input graph.

Parameterized Reduction VI

The following relationship between INDEPENDENT SET and CLIQUE yields parameterized reductions in both directions:

> A graph G has a size-k independent set iff its complement graph $\overline{G}$ has a size-k clique.

This means that, from the parameterized complexity viewpoint, INDEPENDENT SET and CLIQUE are “equally hard”.

Parameterized Reduction VII

Remarks

- ✗ Parameterized reductions are transitive.
- ✗ Very few classical reductions are parameterized reductions.
- ✗ If a parameterized problem is shown to be fixed-parameter intractable, then there cannot be parameterized reductions from this problem to fixed-parameter tractable problems such as VERTEX COVER.

Parameterized Complexity Classes I

Class FPT contains all fixed-parameter tractable problems, such as VERTEX COVER.

The basic degree of parameterized intractability is the class *W[1]*.

Definition

1. The class *W[1]* contains all problems that can be reduced to WEIGHTED 2-CNF-SATISFIABILITY by a parameterized reduction.

2. A parameterized problem is called *W[1]-hard* if WEIGHTED 2-CNF-SATISFIABILITY can be reduced to it by a parameterized reduction.

3. A problem that fulfills both above properties is *W[1]-complete*.

The class *W[2]* is defined analogously by replacing

WEIGHTED 2-CNF-SATISFIABILITY with WEIGHTED CNF-SATISFIABILITY.

Parameterized Complexity Classes II

Example INDEPENDENT SET (analogously CLIQUE) is in W[1]:

A graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ has a size-k independent set iff the 2-CNF formula

$$\bigwedge_{\{i,j\} \in E} (\overline{x_i} \vee \overline{x_j})$$

has a weight-k satisfying truth assignment. Clearly, the parameterized reduction works in polynomial time.

Parameterized Complexity Classes III

Example DOMINATING SET is in W[2]:

A graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ has a size-k dominating set iff the CNF-formula

$$\bigwedge_{i \in V} \bigvee_{j \in N[i]} x_j$$

has a weight-k satisfying truth assignment. Note that the size of the closed neighborhood $N[i]$ of vertex i cannot be bounded from above by a constant, hence an unbounded OR might be necessary here.

Parameterized Complexity Classes IV

To define *W[t]* for $t > 2$ we need the following definition:

Definition A Boolean formula is called *t-normalized* if it can be written in the form of a "product-of-sums-of-products-..." of literals with $t - 1$ alternations between products and sums.

2-CNF formulas are 1-normalized and CNF-formulas are 2-normalized.

Parameterized Complexity Classes V

Definition

1. *W[t]* for $t \geq 1$ is the class of all parameterized problems that can be reduced to WEIGHTED t-NORMALIZED SATISFIABILITY by a parameterized reduction.
2. *W[SAT]* is the class of all parameterized problems that can be reduced to WEIGHTED SATISFIABILITY by a parameterized reduction.
3. *W[P]* is the class of all parameterized problems that can be reduced to WEIGHTED CIRCUIT SATISFIABILITY by a parameterized reduction.

Parameterized Complexity Classes VI

Definition A parameterized language L belongs to the class *XP* if it can be determined in $f(k) \cdot |x|^{g(k)}$ time whether $(x, k) \in L$ where f and g are computable functions only depending on k.

Overview of parameterized complexity hierarchy.

$$\text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[\text{Sat}] \subseteq W[P] \subseteq XP.$$

Conjecture: FPT $\neq$ W[1].

The Complexity Class W[1] I

WEIGHTED ANTIMONOTONE 2-CNF SATISFIABILITY:

- ☞ Input: An antimonotone boolean formula F in conjunctive normal form and a nonnegative integer k.
- ☞ Question: Is there a satisfying truth assignment for F which has weight exactly k, that is, an assignment with exactly k variables set TRUE?

A Boolean formula is called *antimonotone* if it exclusively contains negative literals.

WEIGHTED 2-CNF SATISFIABILITY can be reduced to WEIGHTED ANTIMONOTONE 2-CNF SATISFIABILITY by a parameterized reduction. Thus, WEIGHTED ANTIMONOTONE 2-CNF SATISFIABILITY is W[1]-complete.

The Complexity Class W[1] II

Theorem INDEPENDENT SET is W[1]-complete.

Proof: Reduction from WEIGHTED ANTIMONOTONE 2-CNF SATISFIABILITY. Let F be an antimonotone 2-CNF formula. We construct a graph G_F where each variable in F corresponds to a vertex in G_F and each clause to an edge.

Then, F has a weight-k satisfying assignment iff G_F has a size-k independent set.

Corollary CLIQUE is W[1]-complete.

The Complexity Class W[1] III

It can be shown that

> WEIGHTED q-CNF SATISFIABILITY is W[1]-complete for every constant $q \geq 2$.

Remark:

$$\text{P} = \text{NP} \Rightarrow \text{FPT} = \text{W[1]}$$

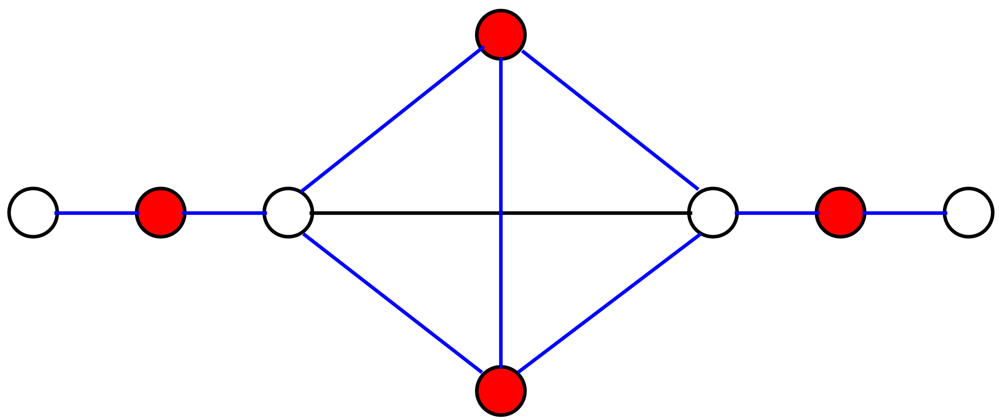
but the reverse direction seems not to hold.

Concrete Parameterized Reductions I

A generalization of VERTEX COVER: PARTIAL VERTEX COVER

- Input: A graph $G = (V, E)$ and two positive integers k and t.
- Question: Does there exist a subset $C \subseteq V$ with at most k vertices such that C covers at least t edges?

Example $t = 9$ and $k = 4$.



Concrete Parameterized Reductions II

Perhaps surprisingly, PARTIAL VERTEX COVER is W[1]-hard with respect to the size k of the partial cover C.

Theorem INDEPENDENT SET can be reduced to PARTIAL VERTEX COVER by a parameterized reduction.

Proof: Let $(G = (V, E), k)$ be an instance of INDEPENDENT SET and let $\deg(v)$ denote the degree of a vertex v in G. Construct a new graph $G' = (V', E')$ from G:

> For each vertex $v \in V$, insert $(|V| - \deg(v) - 1)$ new vertices into G and connect these new vertices with v.

The instance of PARTIAL VERTEX COVER is then (G', k) with $t = k \cdot (|V| - 1)$.

Concrete Parameterized Reductions III

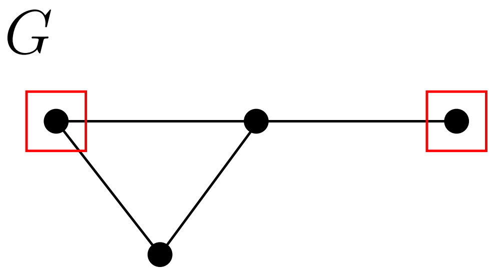
Example of the reduction

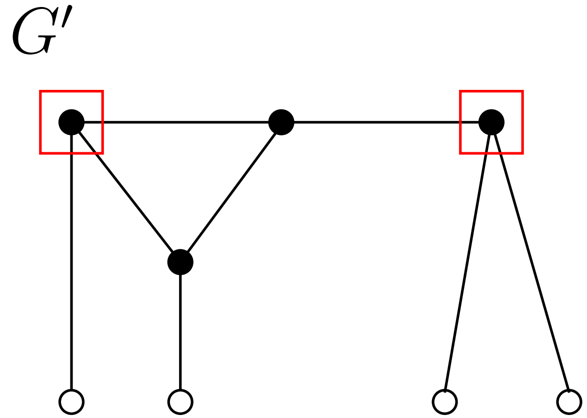
PARTIAL VERTEX COVER instance

with $t = k \cdot (|V| - 1) = 6$ and $k = 2$:

INDEPENDENT SET instance

with $k = 2$:





Concrete Parameterized Reductions IV

Proof of Theorem (continued)

Every size-k independent set I of G is clearly an independent set of G'. Since every vertex in G' has degree $|V|-1$, the vertices in I cover $k \cdot (|V|-1)$ edges in G'.

Given a size-k partial vertex cover C of G' which covers $k \cdot (|V|-1)$ edges, none of the newly inserted vertices in G' can be in C. Moreover, no two vertices in C can be adjacent. Thus, C is an independent set of G.

Concrete Parameterized Reductions V

DOMINATING SET can be expressed as a weighted CNF-Satisfiability problem $\rightsquigarrow$ DOMINATING SET is in W[2].

With a fairly complicated construction, WEIGHTED CNF-SATISFIABILITY can be reduced to DOMINATING SET by a parameterized reduction.

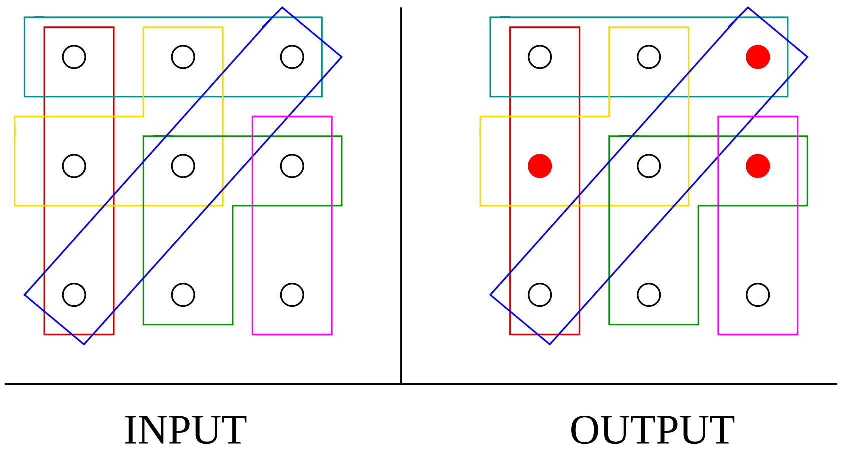
Theorem DOMINATING SET is W[2]-complete with respect to the size of the solution set.

We will use the theorem as the starting point for several W[2]-hardness results.

Concrete Parameterized Reductions VI

HITTING SET

- Input: A collection $\mathcal{C}$ of subsets of a base set S and a nonnegative integer k.
- Question: Is there a subset $S' \subseteq S$ with $|S'| \leq k$ such that S' contains at least one element from each subset in $\mathcal{C}$?



Concrete Parameterized Reductions VII

Note: The special case d-HITTING SET for constant d is fixed-parameter tractable. An unbounded subset size leads to W[2]-hardness.

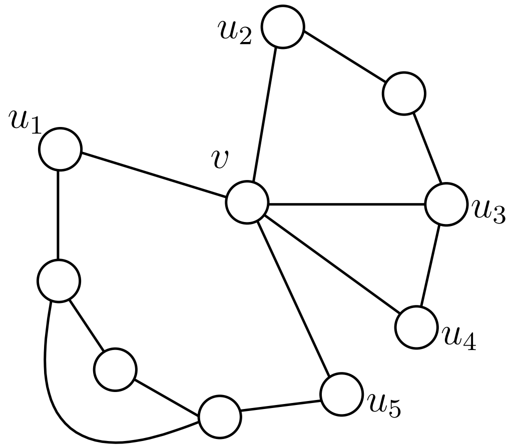
Theorem HITTING SET is W[2]-hard with respect to the size of the solution set.

Proof: A parameterized reduction from DOMINATING SET to HITTING SET: Given a DOMINATING SET instance $(G = (V, E), k)$, construct a HITTING SET instance $(V, \mathcal{C}, k)$ with $\mathcal{C} := \{N[v] \mid v \in V\}$.

Concrete Parameterized Reductions VIII

Proof of the theorem: (continued)

An illustration of the reduction from DOMINATING SET to HITTING SET:



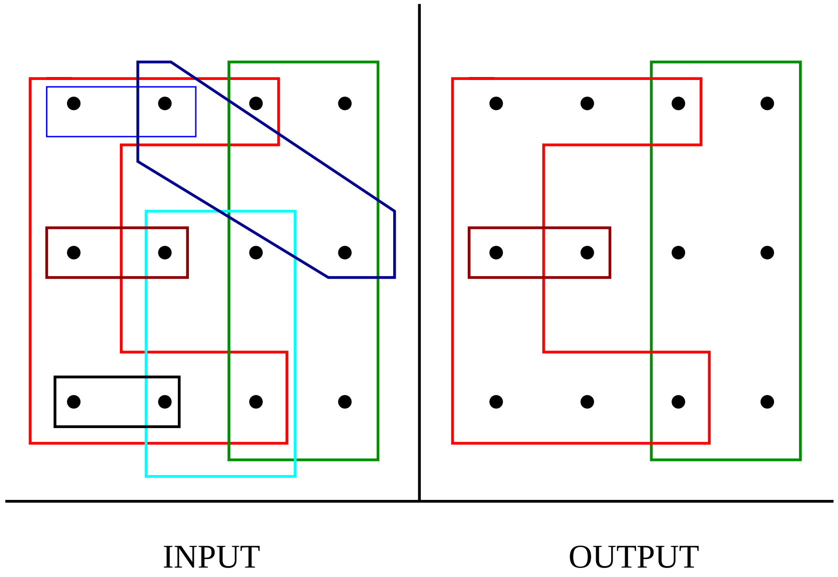
For vertex $v \in V$, add $\{v, u_1, u_2, u_3, u_4, u_5\}$ into $\mathcal{C}$.

G has a dominating set of size $k \Leftrightarrow \mathcal{C}$ has a hitting set of size k.

Concrete Parameterized Reductions IX

SET COVER

- Input: A base set $S = \{s_1, s_2, \ldots, s_n\}$, a collection $\mathcal{C} = \{c_1, \ldots, c_m\}$ of subsets of S such that $\bigcup_{1 \leq i \leq m} c_i = S$, and a nonnegative integer k.
- Question: Is there a subset $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| \leq k$ which covers all elements of S, that is, $\bigcup_{c \in \mathcal{C}'} c = S$?



Concrete Parameterized Reductions X

Theorem SET COVER is W[2]-hard with respect to the size of the solution set.

Proof: The W[2]-hardness of SET COVER follows from a well-known equivalence between SET COVER and HITTING SET.

Let $(S = \{s_1, \ldots, s_n\}, \mathcal{C} = \{c_1, \ldots, c_m\}, k)$ be a HITTING SET instance. Define

$$\hat{\mathcal{C}} = \{c'_1, \ldots, c'_n\}$$

where

$$c'_i = \{t_j \mid 1 \leq j \leq m, s_i \in c_j\}.$$

Herein, $S' := \{t_1, t_2, \ldots, t_m\}$ forms the base set in the SET COVER instance.

$\mathcal{C}$ has a size-k hitting set $\Leftrightarrow$ $\hat{\mathcal{C}}$ has a size-k set cover.

Concrete Parameterized Reductions XI

POWER DOMINATING SET is a problem motivated by electrical power networks.

To define the POWER DOMINATING SET problem, we need two “observation rules”.

Observation Rule 1 (OR1): A vertex in the power dominating set observes itself and all of its neighbors.

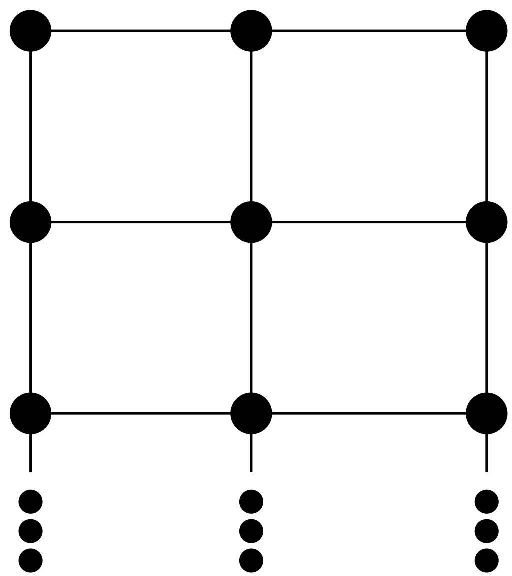
Observation Rule 2 (OR2): If an observed vertex v of degree $d \geq 2$ is adjacent to $d - 1$ observed vertices, then the remaining unobserved vertex becomes observed as well.

POWER DOMINATING SET is defined as follows:

- Input: A graph $G = (V, E)$ and a nonnegative integer k.
- Question: Is there a subset $C \subseteq V$ with at most k vertices that observes all vertices in V with respect to the two observation rules OR1 and OR2?

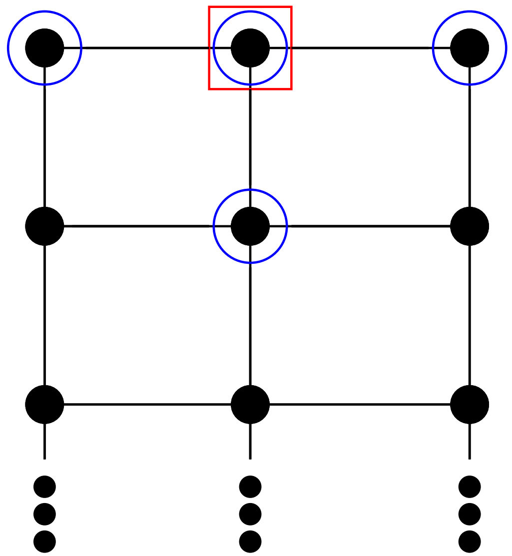
Concrete Parameterized Reductions XII

Example An $(m \times 3)$-grid:



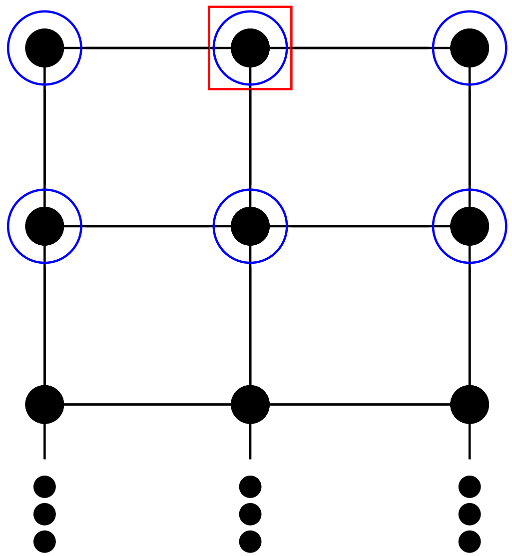
Concrete Parameterized Reductions XII

Example An $(m \times 3)$-grid:



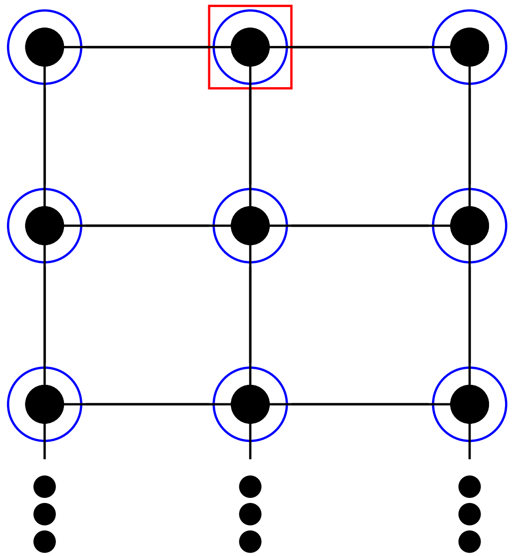
Concrete Parameterized Reductions XII

Example An $(m \times 3)$-grid:



Concrete Parameterized Reductions XII

Example An $(m \times 3)$-grid:



An $(m \times 3)$-grid has a size-one power dominating set.

Concrete Parameterized Reductions XIII

A useful lemma:

Lemma If G is a graph with at least one vertex of degree at least three, then there is always a minimum power dominating set which contains only vertices with degree at least three.

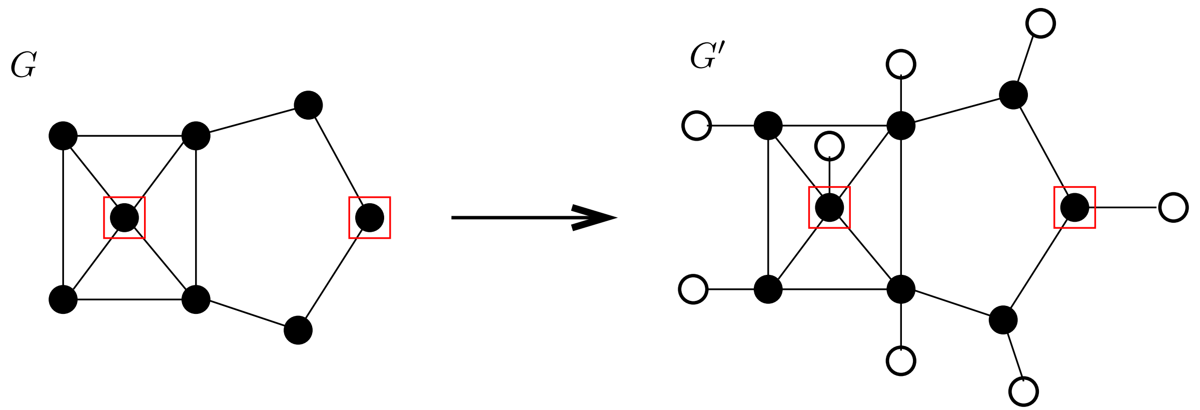
Theorem POWER DOMINATING SET is W[2]-hard with respect to the size of the solution set.

Concrete Parameterized Reductions XIV

Proof of Theorem: Reduce DOMINATING SET to POWER DOMINATING SET:

Given an instance $(G = (V, E), k)$ of DOMINATING SET, we construct a POWER DOMINATING SET instance $(G' = (V \cup V_1, E \cup E_1), k)$ by simply attaching newly introduced degree-1 vertices to all vertices from V.

An illustration:



G has a size-k dominating set $\Leftrightarrow$ G' has a size-k power dominating set.

LONGEST COMMON SUBSEQUENCE I

We finish "Parameterized Complexity Theory" with the LONGEST COMMON SUBSEQUENCE problem:

- Input: A set of k strings $s_1, s_2, \ldots, s_k$ over an alphabet Σ and a positive integer L.
- Question: Is there a string $s \in \Sigma^*$ of length at least L that is a subsequence of every s_i, $1 \leq i \leq k$?

Example $\Sigma = \{$a,b,c,d$\}$, $k = 3$, and $L = 5$.

$s_1 =$ abcadbccd

$s_2 =$ adbacbddb

$s_3 =$ acdbacbcd

$s =$ abacd.

Longest Common Subsequence II

Three natural parameters:

- the number k of input strings,
- the length L of the common subsequence,
- and, somewhat aside, the size of the alphabet Σ.

In case of *constant-size* alphabet Σ, Longest Common Subsequence is

- fixed-parameter tractable with respect to parameter L;
- W[1]-hard with respect to parameter k.

Longest Common Subsequence III

Three natural parameters:

- the number k of input strings,
- the length L of the common subsequence,
- and, somewhat aside, the size of the alphabet Σ.

In case of *unbounded alphabet size*, Longest Common Subsequence is

- W[t]-hard for all $t \geq 1$ with respect to parameter k.
- W[2]-hard with respect to parameter L;
- W[1]-hard with respect to the combined parameters k and L.

Ongoing New Developments

- ✗ Substructuring of the class of fixed-parameter tractable problems has recently been initiated.
- ✗ Lower bounds on the running time of exact algorithms for fixed-parameter tractable problems as well as for W[1]-hard problems are another interesting research issue.
- ✗ Several further parameterized complexity classes are known and lead to numerous challenges concerning structural complexity investigations.
- ✗ So far, only few links have been established between parameterized intractability theory and inapproximability theory.

Connections to Approximation Algorithms I

In favor of polynomial-time approximation algorithms:

- ✗ No matter what the input is, efficiency in terms of polynomial-time complexity is always guaranteed.
- ✗ The approximation factor provides a worst-case guarantee, and in practical applications the actual approximation might be much better, thus turning approximation algorithms into useful heuristic algorithms as well.
- ✗ There is a huge arsenal of methods and techniques developed over the years in conjunction with studying approximation algorithms, and there is a strong and deep theoretical foundation for impossibility results particularly concerning lower bounds for approximation factors.

Connections to Approximation Algorithms II

In favor of fixed-parameter algorithms:

- ✗ There is a lot of freedom in choosing the “right” parameterization.
- ✗ Fixed-parameter algorithms stick to worst-case analysis but in practical applications they might turn out to run much faster than predicted by worst-case upper-bounds.
- ✗ Although not that much developed and matured as inapproximability theory with the famous PCP theorem, also parameterized complexity already made worthwhile contributions to structural complexity theory.

Connections to Approximation Algorithms III

Definition A minimization problem has

- a *polynomial-time approximation scheme (PTAS)* if, for any constant $\epsilon > 0$, there is a factor-$(1+\epsilon)$ approximation algorithm running in polynomial time;
- an *efficient polynomial-time approximation scheme (EPTAS)* if, for any constant $\epsilon > 0$, there is a factor-$(1+\epsilon)$ approximation algorithm running in $f(1/\epsilon) \cdot |X|^{O(1)}$ time for any computable function f only depending on $1/\epsilon$;
- a *fully polynomial-time approximation scheme (FPTAS)* if, for any constant $\epsilon > 0$, there is a factor-$(1+\epsilon)$ approximation algorithm running in $(1/\epsilon)^{O(1)} \cdot |X|^{O(1)}$ time.

Connections to Approximation Algorithms III

Definition Let x be an input instance of an optimization problem where we want to minimize the goal function $m(x)$. Then its *standard parameterization* is the pair (x, k) which means that we ask whether $m(x) \leq k$ for a given parameter value k.

Theorem If a minimization problem has an EPTAS, then its standard parameterization is fixed-parameter tractable.

Corrollary If the standard parameterization of an optimization problem is not fixed-parameter tractable, then there is no EPTAS for this optimization problem.