Overview

Introduction

Parameterized algorithms are a method for the exact solution of hard problems.

Other such methods:

- Heuristics
- Simulated annealing
- Approximation algorithms
- Genetic algorithms
- Branch- and Bound
- Backtracking
- Total enumeration

NP-complete Problems

Many problems encountered in practice are NP-complete.

We know from complexity theory:

Definition

A language L is NP-complete, if

- $L \in NP$
- Every problem in NP can be reduced to L in polynomial time.

Theorem

If there is a polynomial time algorithm for an NP-complete problem, then $P = NP$.

Question: Does that mean that NP-complete problems are hard to solve in practice?

NP-complete problems

Why is SAT (satisfiability) NP-complete?

Because the computation of a nondeterministic Turing-maching can be simulated by a combinatorial circuit.

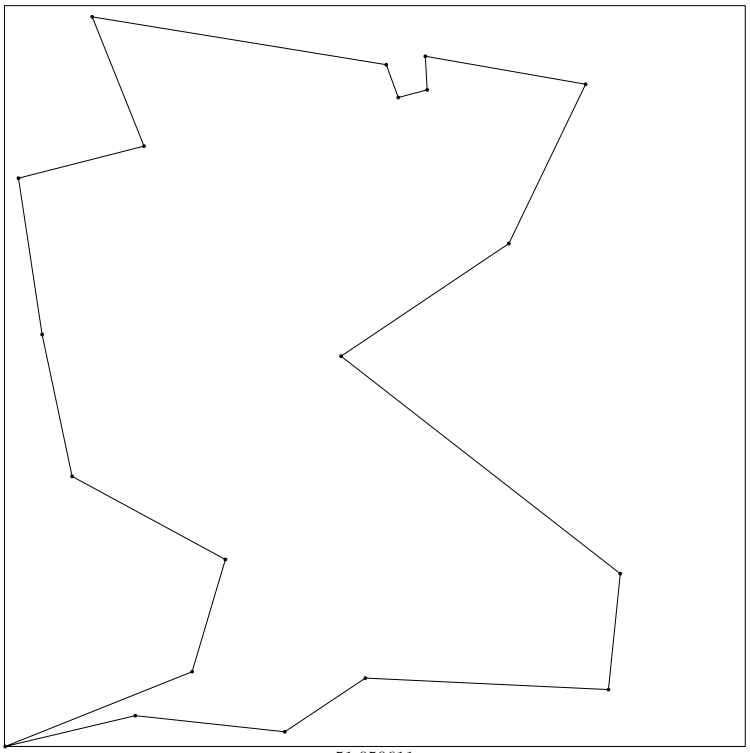
The existence of a successful computation of a Turingmachine can be reduced to the existence of a satisfying assignment for a circuit.

Therefore there are formulas whose satisfiability is as hard to determine as to solve any problem in NP.

If look at the set of all formulas, then some of them are indeed very hard.

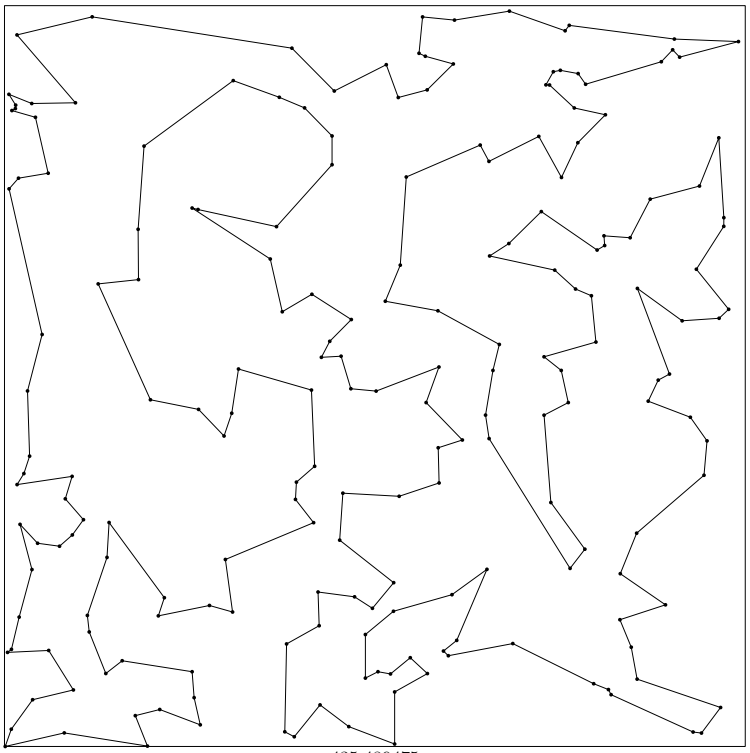
But most formulas are not constructed in this way!

Example: TSP



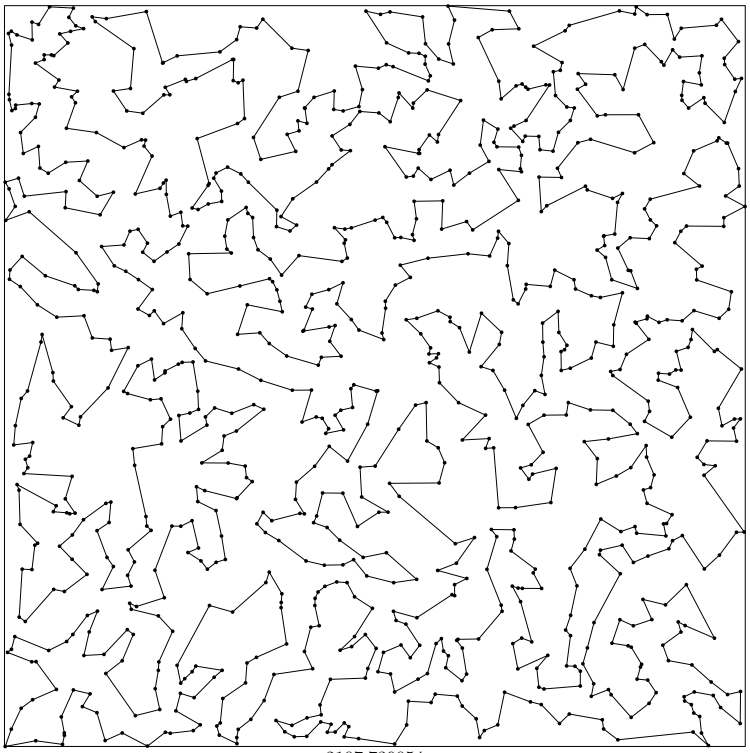
51.050611

Example: TSP



435.489475

Example: TSP



2107.739054

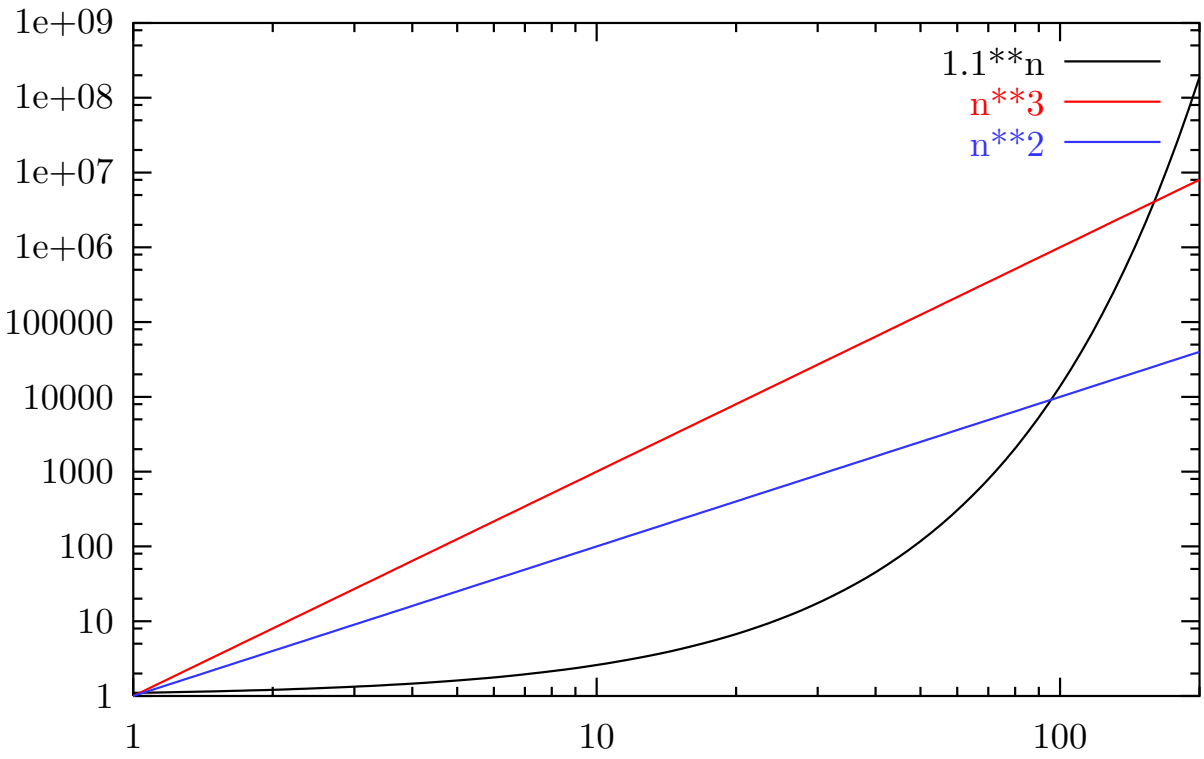
Running Times

NP-complete problems are hard in practice because there are no algorithms that always go in the right direction.

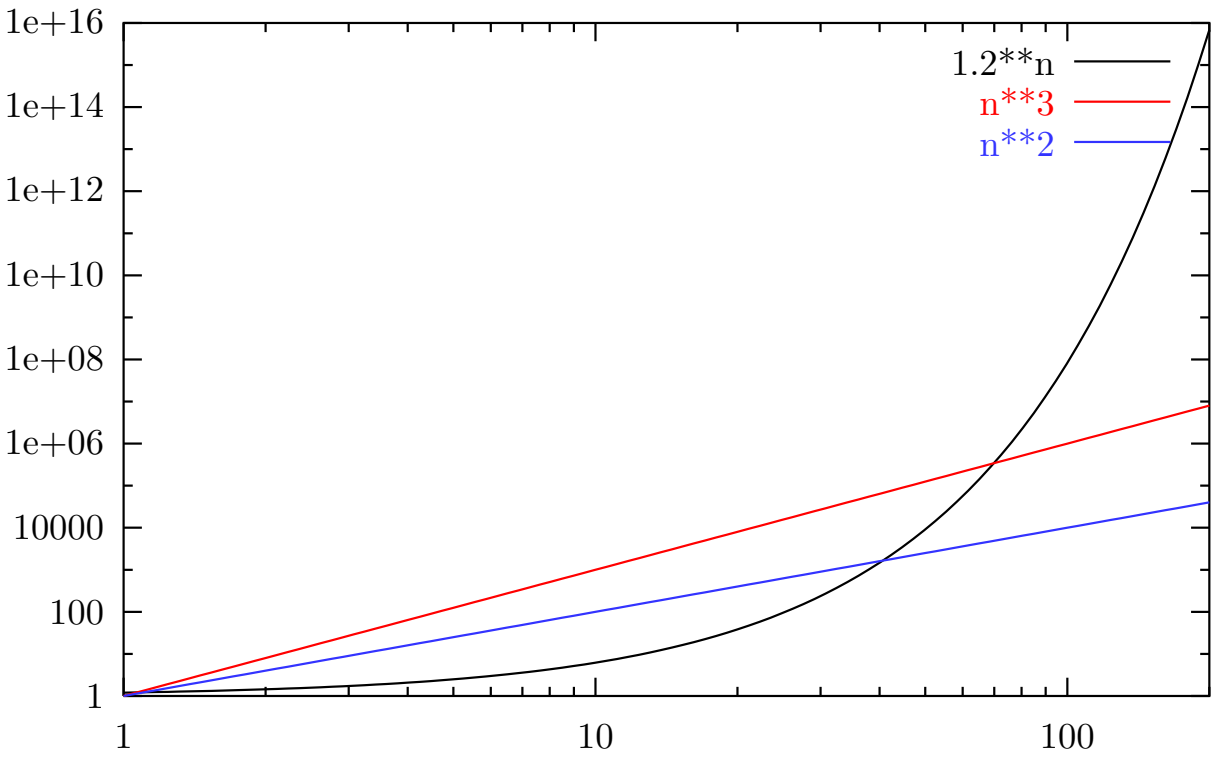
- Greedy-Algorithmen
- Divide-and-Conquer
- Dynamic Programming

Hence, many wrong partial solutions have to be considered, leading to exponential running times.

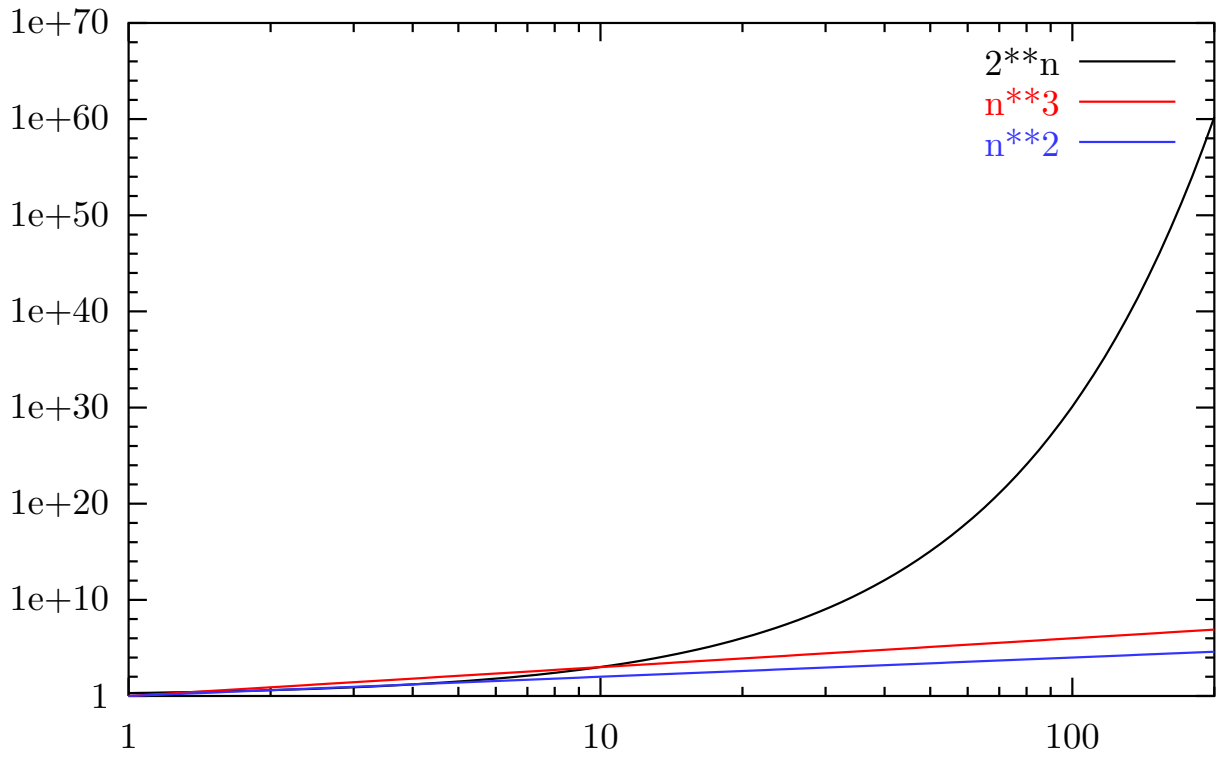
Comparing Running Times



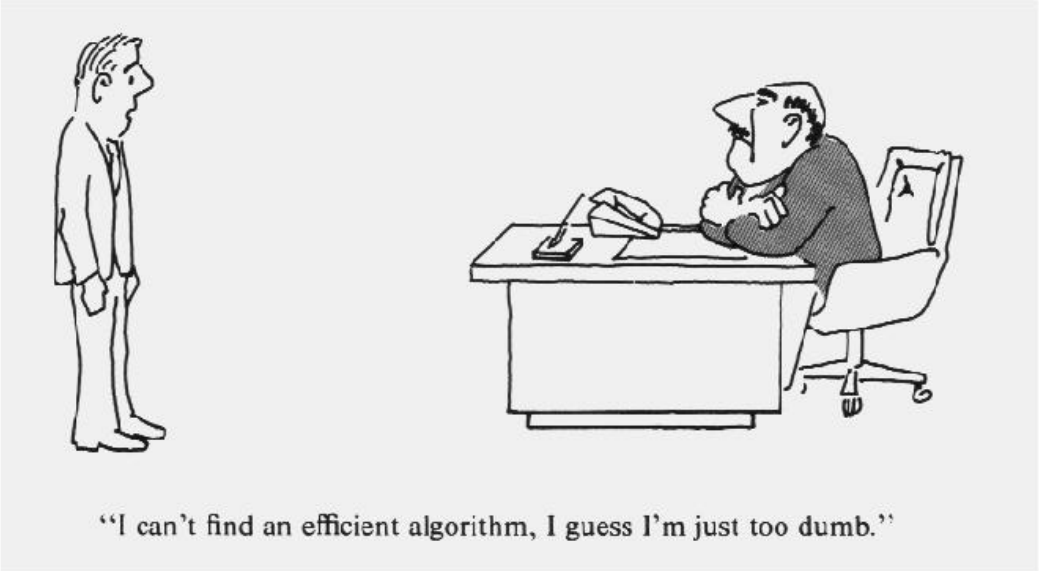
Comparing Running Times



Comparing Running Times



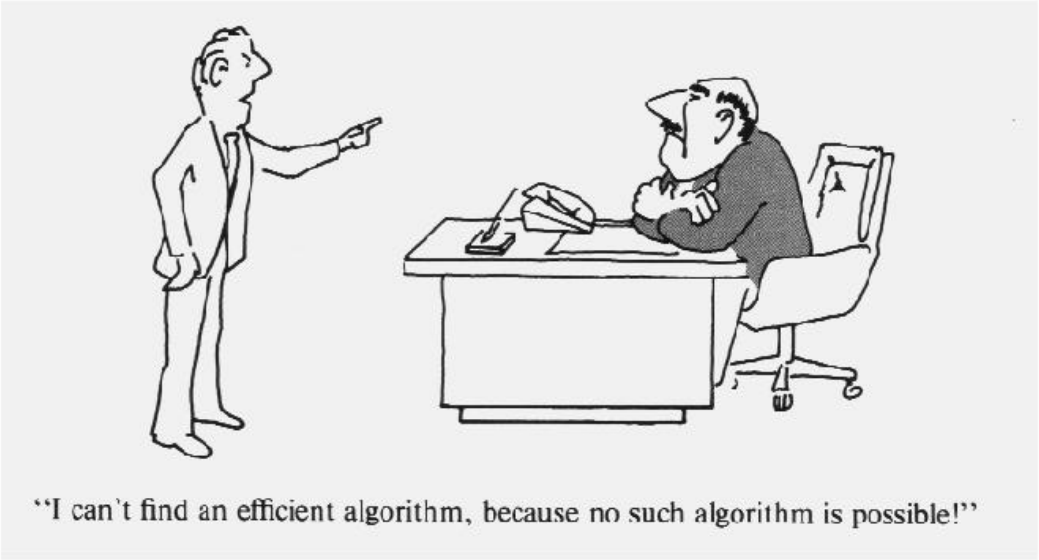
NP-Completeness as an Excuse



"I can't find an efficient algorithm, I guess I'm just too dumb."

Garey and Johnson. Computers and Intractability.

NP-Completeness as an Excuse



"I can't find an efficient algorithm, because no such algorithm is possible!"

Garey and Johnson. Computers and Intractability.

NP-Completeness as an Excuse

"I can't find an efficient algorithm, but neither can all these famous people."

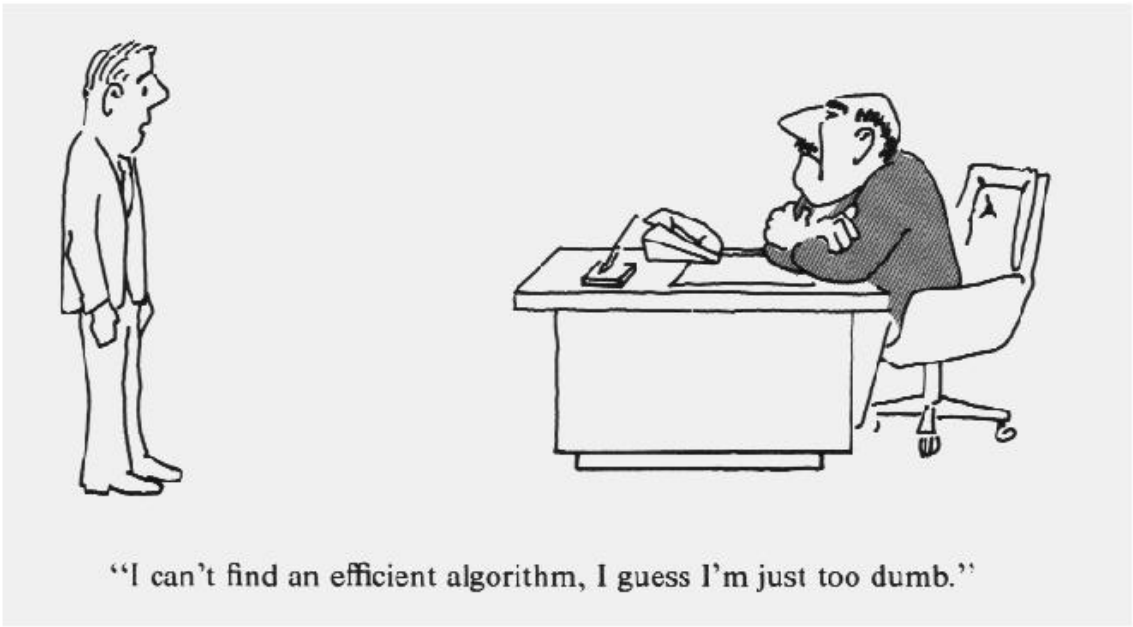
Garey and Johnson. Computers and Intractability.

NP-Completeness as an Excuse

Molecular biologist Joseph Felsenstein:

> *About ten years ago, some computer scientists came by and said they heard we have some really cool problems. They showed that the problems are NP-complete and went away!*

NP-Completeness as an Excuse



"I can't find an efficient algorithm, I guess I'm just too dumb."

Overview

Easy and Hard Instances

- Exponential running time in the worst case
- Running time needs to be huge only for some instances
- Practical instances might be easy
- How can we distinguish between hard and easy instances?

Parameter

We assign a number, the parameter, to each instance.

Our hope:

- Good running times for small parameters
- Instances occuring in practice have small parameters

There is no contradiction to the NP-completeness of the problem!

Main Definition

Let there be an algorithmic problem.

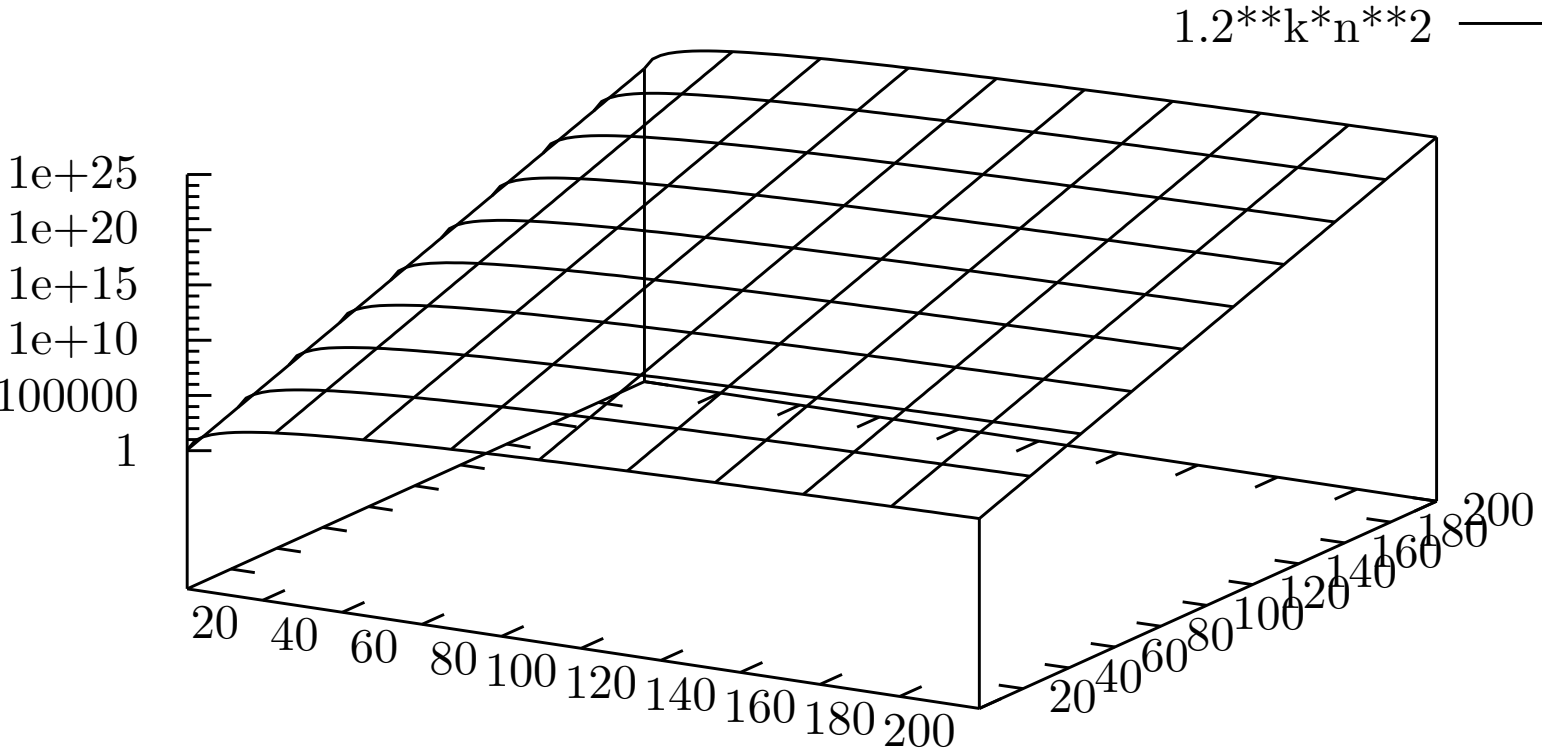
Let n be the size of some instance and k the corresponding parameter.

The problem is fixed parameter tractable, if there is an algorithm solving the problem whose running time is

$$O(f(k)n^c).$$

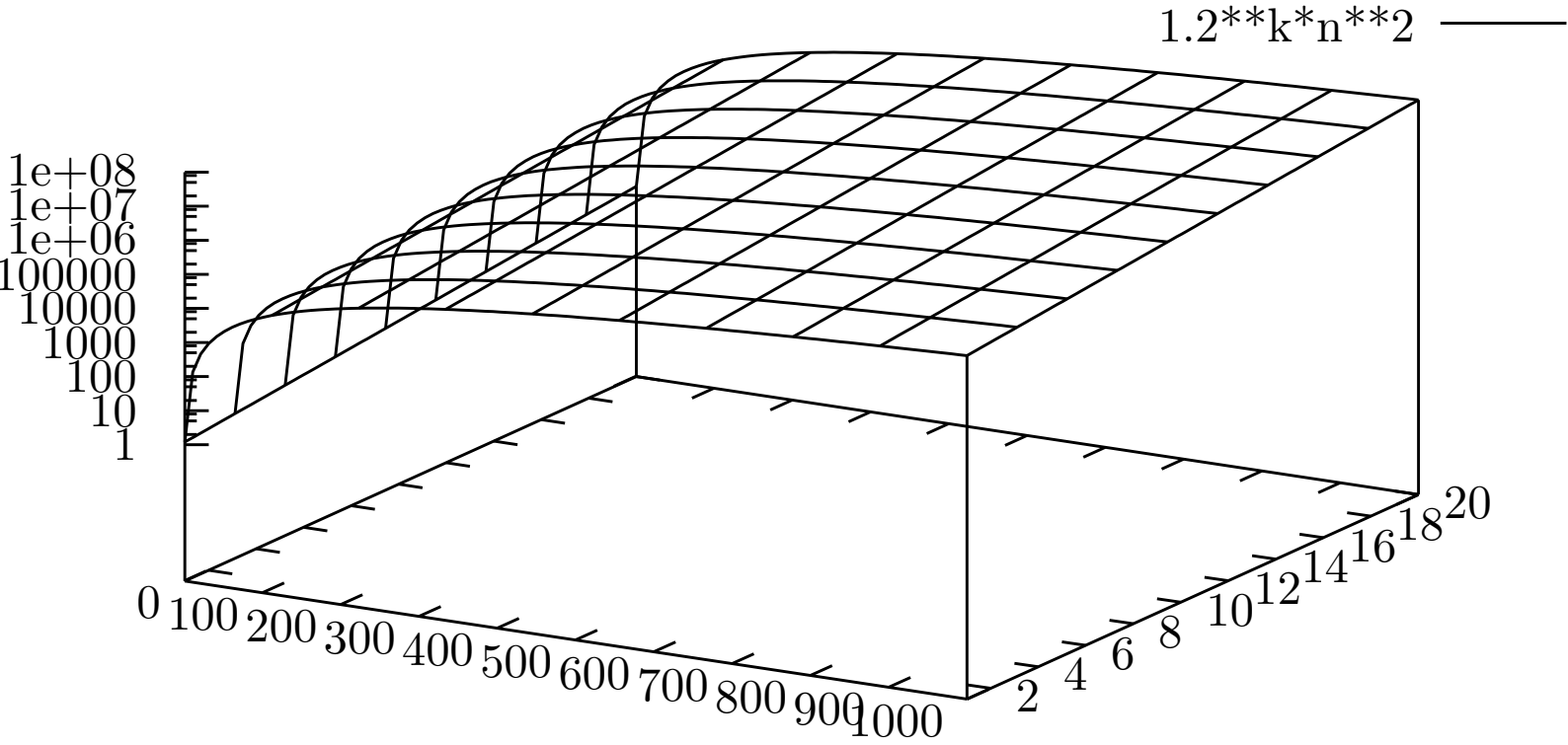
Here c is a constant and f an arbitrary function.

Running Time of a Parameterized Algorithm



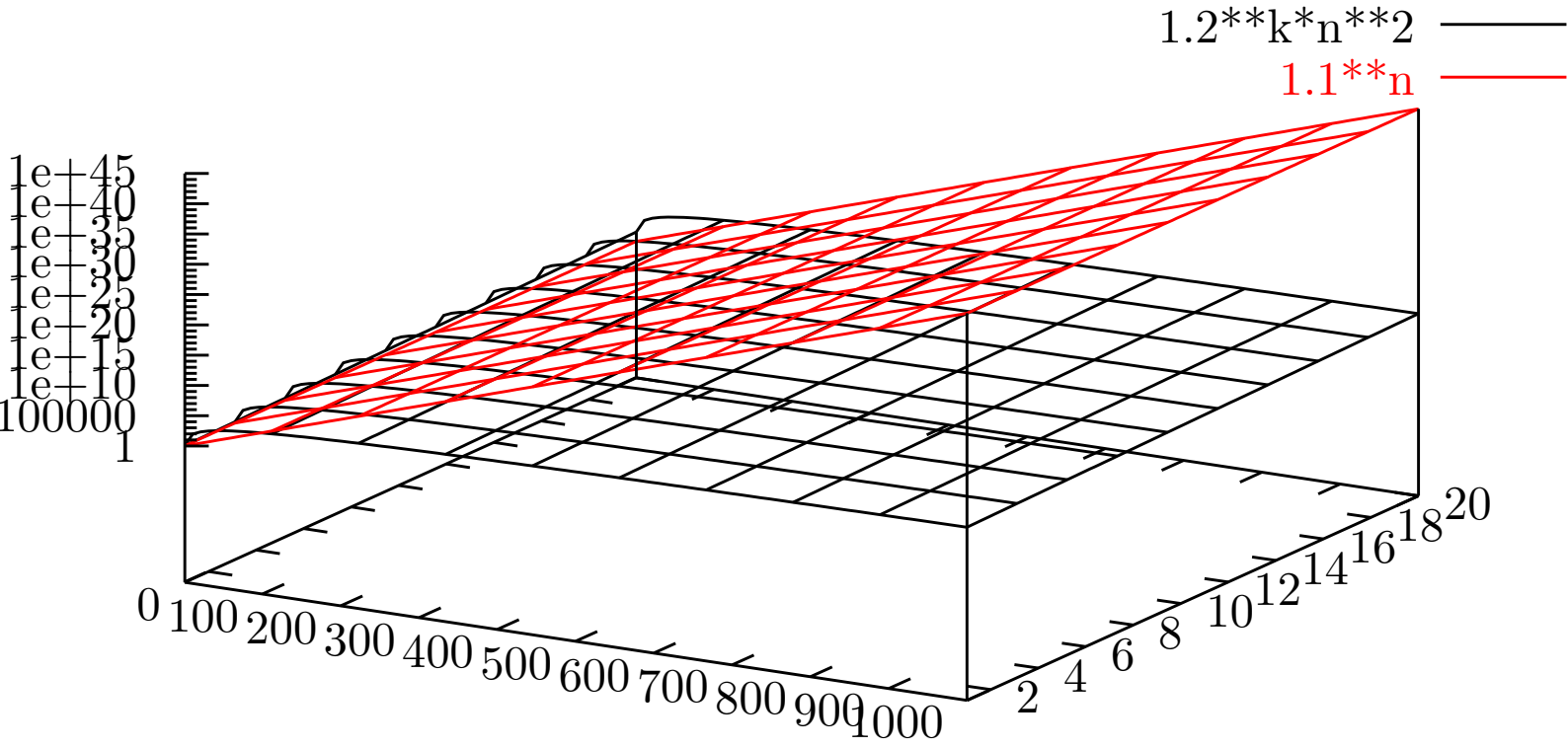
The running time is $1.2^k n^2$. The parameter is between 1 and n.

Running Time of a Parameterized Algorithm



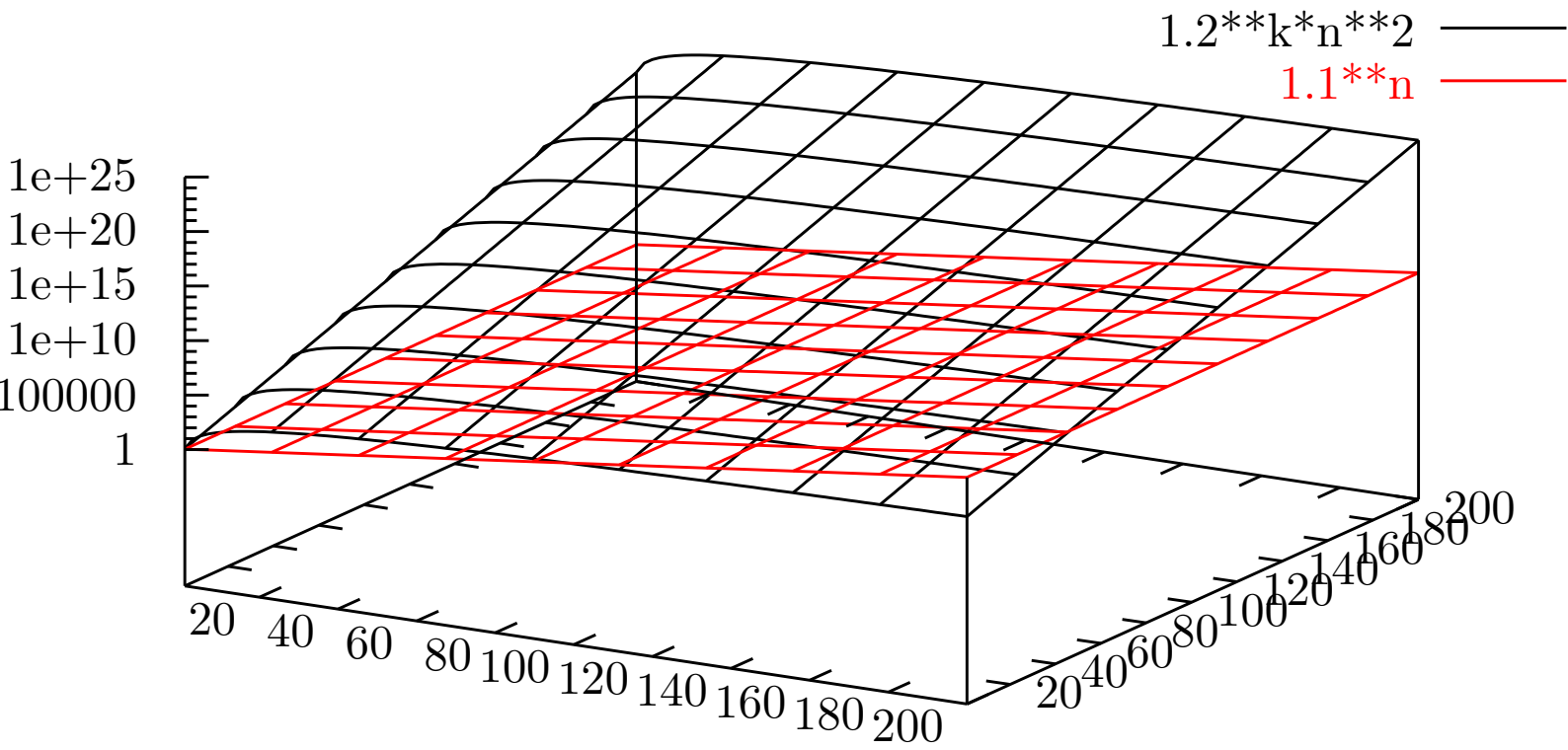
The running time is $1.2^k n^2$. The parameter is small.

Running Time of a Parameterized Algorithm



The running time is $1.2^k n^2$. The parameter is small. The non-parameterized algorithm has running time 1.1^n.

Running Time of a Parameterized Algorithm



Example: Vertex Cover

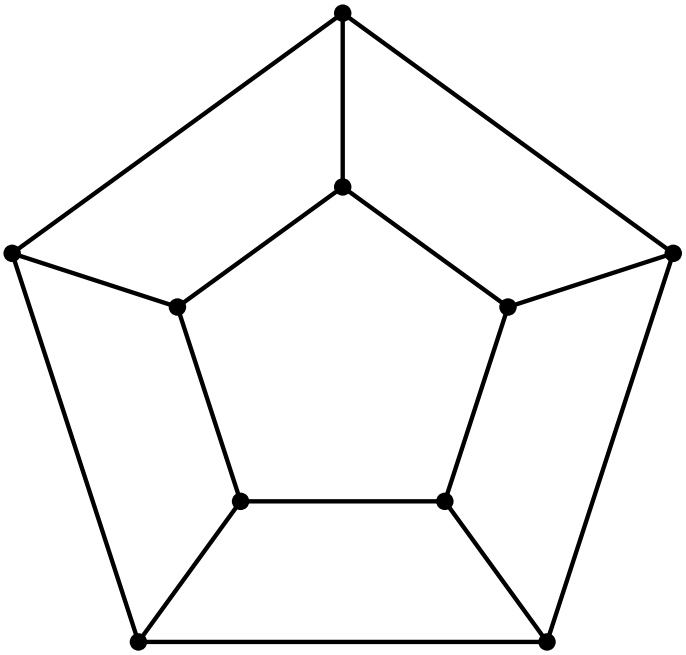
Input: A graph $G = (V, E)$.

Output: A minimal Vertex Cover $C \subseteq E$.

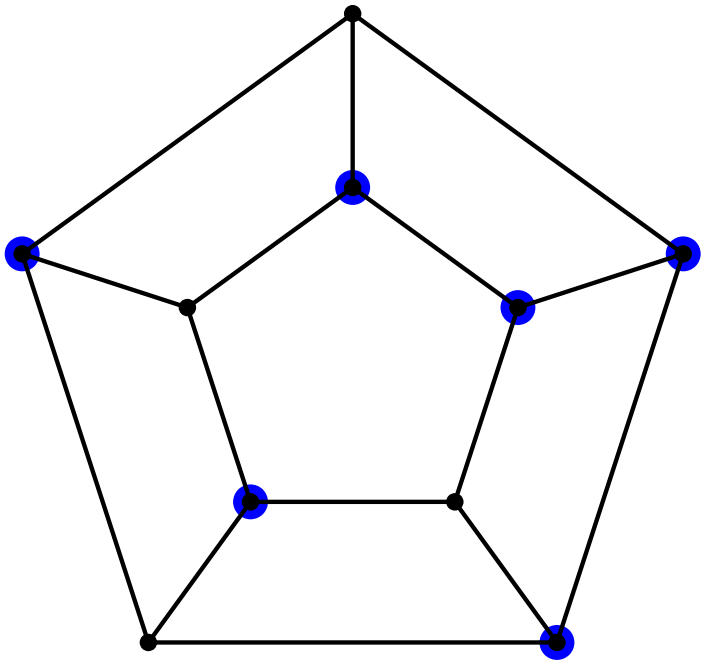
Definition

A set $C \subseteq V$ is a Vertex Cover of $G = (V, E)$, if at least one vertex of each edge in E is in C.

Example



Example



Expressing Vertex Cover as an ILP

Let $G = (V, E)$ be a graph with $V = \{v_1, \ldots, v_n\}$.

$$\begin{aligned} \text{Minimize } & v_1 + \ldots + v_n \\ \text{subject to } & 0 \leq v_i \leq 1 \text{ for } i = 1, \ldots, n \\ & v_i + v_j \geq 1 \text{ for } \{v_i, v_j\} \in E \\ & v_i \in \mathbf{Z} \text{ for } i = 1, \ldots, n \end{aligned}$$

Every NP-complete problem can be reduced to an ILP (but often it is a bad idea to do so).

British Museum Method

Many important NP-complete problems are indeed search problems. In some (very big) search space the solutions are well hidden.

One possible plan of attack is consequently to exhaustively search the whole search space.

In the case of vertex cover this amounts to looking at all $C \subseteq V$.

That makes $2^{|V|}$ different subsets.

The running time is $O(|E|2^{|V|})$.

Backtracking

Consider some vertex $v \in V$.

There are the two possibilities $v \in C$ or $v \notin C$.

If $v \notin C$, then $N(v) \subseteq C$, because all edges incident to v must be covered.

($N(v)$ is the neighborhood of v, i.e., all nodes adjacent to v.)

These simple observations lead immeadiately to an algorithm.

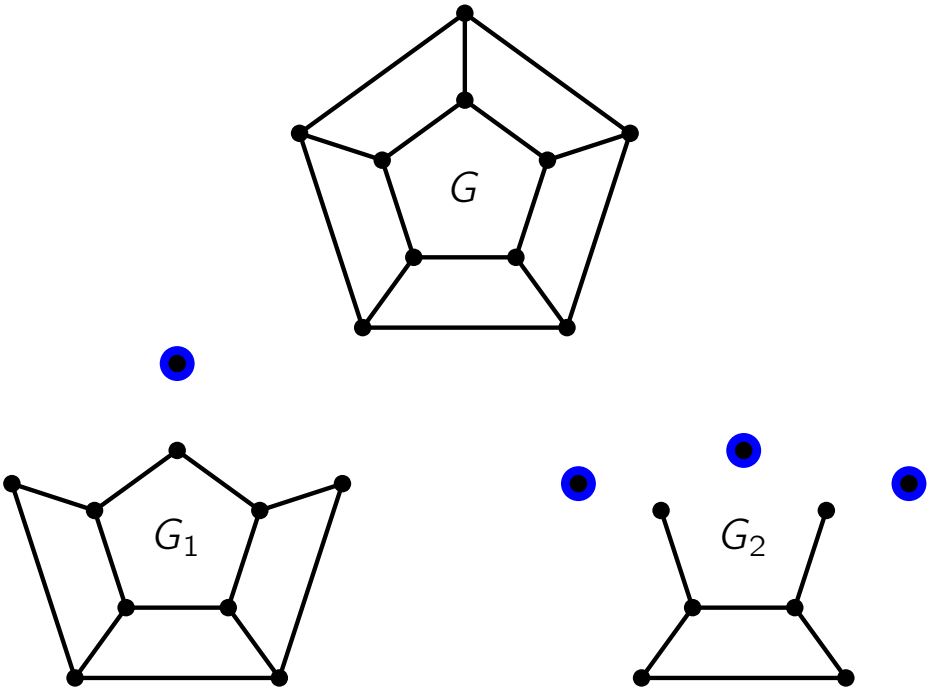
Backtracking

Input: $G = (V, E)$

Output: An optimal vertex cover $VC(G)$

if $V = \emptyset$ **then return** $\emptyset$
Choose an arbitrary node $v \in G$
$G_1 := (V - \{v\}, \{ e \in E \mid v \notin e \})$
$G_2 := (V - \{v\} - N(v), \{ e \in E \mid e \cap N(v) = \emptyset \})$
if $|\{v\} \cup VC(G_1)| \leq |N(v) \cup VC(G_2)|$
then return $\{v\} \cup VC(G_1)$
else return $N(v) \cup VC(G_2)$

Backtracking



Backtracking (a different approach)

Every edge $e = \{v_1, v_2\}$ must be covered by v_1 of v_2.

Hence, we can look at an edge $\{v_1, v_2\}$ and try recursively both possibilities:

- $v_1 \in C$
- $v_2 \in C$

This again leads to immeadiately to a simple algorithm:

Backtracking (a different approach)

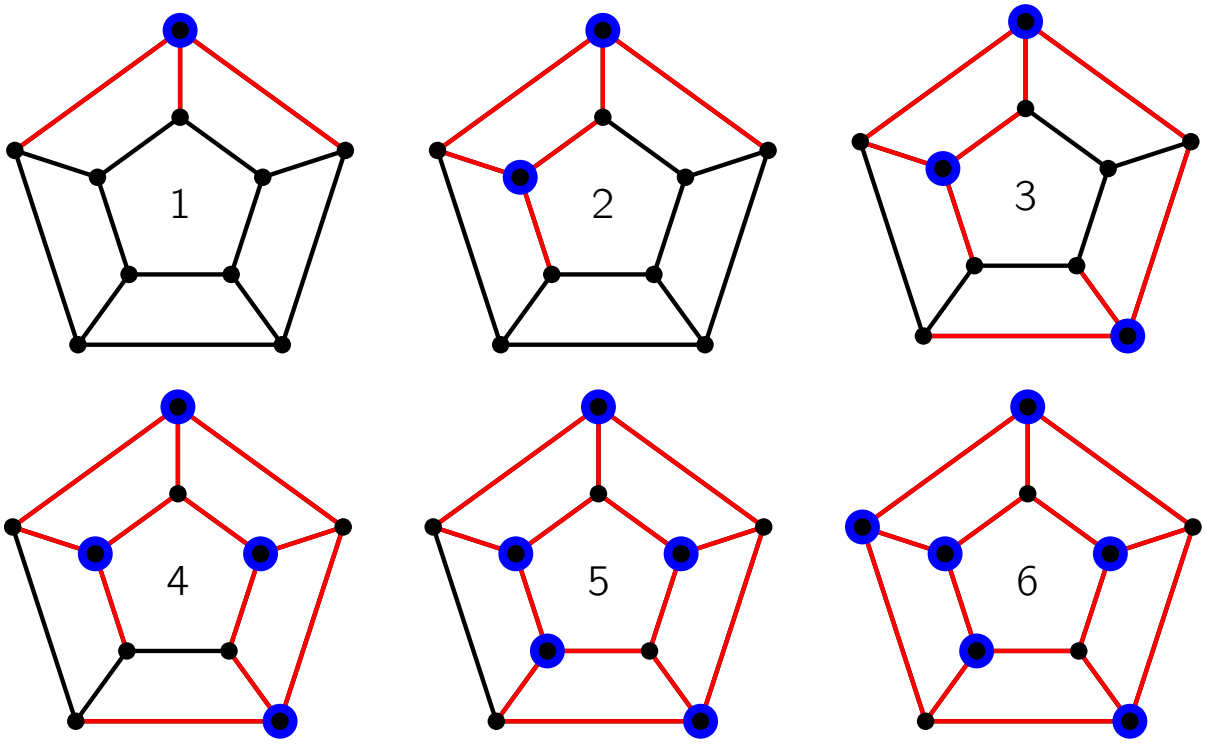
Input: $G = (V, E)$

Output: An optimal vertex cover $VC(G)$

if $E = \emptyset$ **then return** $\emptyset$
Choose some edge $\{v_1, v_2\} \in E$
$G_1 := (V - \{v_1\}, \{ e \in E \mid v_1 \notin e \})$
$G_2 := (V - \{v_2\}, \{ e \in E \mid v_2 \notin e \})$
if $|\{v_1\} \cup VC(G_1)| \leq |\{v_2\} \cup VC(G_2)|$
then return $\{v_1\} \cup VC(G_1)$
else return $\{v_2\} \cup VC(G_2)$

This recursive algorithms computes an optimal vertex cover.

Heuristics



Always choose a vertex with maximal degree (greedy).

Approximation algorithms

Every edge has to be covered by at least one of its vertices.

Problem: Which one?

Solution: Take both.

- Naturally there is no guarentee that we find an optimal solution.
- The vertex cover found in this way can be at most twice as big as an optimal one.

Approximation algorithms

The algorithm might look like this:

$C := \emptyset$;
while $E \neq \emptyset$ **do**
  Choose some $e \in E$;
  $V := V - e$;
  $C := C \cup e$;
  $E := \{ e' \in E \mid e \cap e' = \emptyset \}$
od;
return C

Parameterized Algorithm

Input: $G = (V, E), k$

Parameter: k

Output: A vertex cover $VC(G, k)$ of size k or smaller, if it exists.

if $E = \emptyset$ **then return** $\emptyset$
if $k = 0$ **then return** "no solution"
Choose some edge $\{v_1, v_2\} \in E$
$G_1 := (V - \{v_1\}, \{ e \in E \mid v_1 \notin e \})$
$G_2 := (V - \{v_2\}, \{ e \in E \mid v_2 \notin e \})$
if $|\{v_1\} \cup VC(G_1, k-1)| \leq |\{v_2\} \cup VC(G_2, k-1)|$
then return $\{v_1\} \cup VC(G_1, k-1)$
else return $\{v_2\} \cup VC(G_2, k-1)$

Parameterized Algorithm

Input: $G = (V, E), k$

Parameter: k

Output: A vertex cover $VC(G, k)$ of size k or smaller, if it exists.

if $E = \emptyset$ **th**
if $k = 0$ **the**
Choose som
$G_1 := (V -$
$G_2 := (V -$
if $|\{v_1\} \cup V$
then return
else return

Questions:

1. What does "no solution" mean?
2. Why is the running time $O(f(k)n^c)$?
3. What exactly is $f(k)$?
4. Do we always find an optimal vertex cover?
5. Can we simplify the last lines of the algorithm?

Parameterized Algorithm

Input: $G = (V, E), k$

Output: A vertex cover $VC(G, k)$ of size k or smaller, if it exists.

if $E = \emptyset$ **then return** $\emptyset$
if $k = 0$ **then return** "no solution"
Choose some edge $\{v_1, v_2\} \in E$
$G_1 := (V - \{v_1\}, \{ e \in E \mid v_1 \notin e \})$
$G_2 := (V - \{v_2\}, \{ e \in E \mid v_2 \notin e \})$
if $VC(G_1, k-1) \neq$ "no solution"
then return $\{v_1\} \cup VC(G_1, k-1)$
else return $\{v_2\} \cup VC(G_2, k-1)$

Parameterized Algorithm — Running Time

Every recursive call requires only polynomial time.

How many recursive calls are there?

Every incarnation is a leaf in the recursion tree or has two children.

- The root has parameter k
- The parameter of a child is at least one smaller compared to the parent
- The parameter never becomes negative

Therefore the height of the recursion tree is at most k

Its size is then at most 2^k.

The Long Road to Vertex Cover

- Fellows & Langston (1986): $O(f(k)n^3)$
- Robson (1986): $O(1.211^n)$
- Johnson (1987): $O(f(k)n^2)$
- Fellows (1988): $O(2^k n)$
- Buss (1989): $O(kn + 2^k k^{2k+2})$
- Downey, Fellows, & Raman (1992): $O(kn + 2^k k^2)$
- Balasubramanian, Fellows, & Raman (1996): $O(kn + 1.3333^k k^2)$
- Balasubramanian, Fellows, & Raman (1998): $O(kn + 1.32472^k k^2)$

The Long Road to Vertex Cover

- Downey, Fellows, Stege (1998): $O(kn + 1.31951^k k^2)$
- Niedermeier & R. (1998): $O(kn + 1.292^k)$
- Chen, Kanj, & Jia (1999): $O(kn + 1.271^k k^2)$
- Chen, Kanj, & Jia (2001): $O(kn + 1.285^k)$
- Niedermeier & R. (2001): $O(kn + 1.283^k)$
- Chandran & Grandoni (2004): $O(kn + 1.275^k k^{1.5})$
- Chen, Kanj, & Xia (2005): $O(kn + 1.274^k)$

Bounded Search Trees

A Bounded search tree algorithm must fulfil these condition on its recursion tree:

- Every node is labeled by some natural number
- The root is labeled by some function of the parameter
- The number of children of a node is limited by some function of the parent's label
- Children are labeled by smaller numbers than the parent

Correctness

Theorem

Let an algorithm be a bounded search tree algorithm.

Then there is a function f, such that every search tree for an input with parameter k has at most $f(k)$ many nodes.

Proof of Correctness

Proof

We define a function $S(k)$ that is an upper bound on the number of leaves in a subtree whose root is labeled by k.

- Assume that the root is labeled with at most $w(k)$
- Assume that every node with label k hat at most $b(k)$ many children
- The existence of w and b is guaranteed by the definition of bounded search trees.

Proof of Correctness (cont.)

Proof

$$S(k) \leq b(k)S(k-1),$$

because there are at most $b(k)$ children whose subtrees have each at most $S(k-1)$ many leaves.

With $S(0) = 1$ the solution of this recurrence is

$$S(k) \leq \prod_{i=1}^{k} b(i).$$

The total number of leaves consequently is at most $S(w(k))$.

Example Closest String

Let u and v be two strings of length n.

We define $h(u, v)$, called Hamming distance of u and v, as the number of positions on which u and v differ.

Example:

$$h(agctcagtaccc, agctcataacgc) = 3$$

Example Closest String

The Closest string problem is defined as follows:

Input: k strings $s_1, \ldots, s_k$ of length n, a number m

Question: Is there a string s with $h(s, s_i) \leq m$ for all $1 \leq i \leq k$?

The parameter is m

Motivation: Construct a chemical marker that closely fits to a set of DNA sequences

In practice m is small, e.g. 5

Example Closest String

agcacagtacgcaatagtgtcgcaggt
agctcagtagccaatagagtcccaggt
agatcagttcccaatagagtcgcacgt
agctcagtaaaaaatagagtcgcaggt
agcgcagtacacaatagagtcgcaagt

Example Closest String

agcacagtacgcaatagtgtcgcaggt
agctcagtagccaatagagtcccaggt
agatcagttcccaatagagtcgcacgt
agctcagtaaaaaatagagtcgcaggt
agcgcagtacacaatagagtcgcaagt

---

agctcagtacccaatagagtcgcaggt

Example Closest String

gctaggagtcagaagtaggcgttgcat
gcaatgaatcagaactgggcctagcat
gctagggatcagaactaggcctagcat
gcaaggaatcataactaggcctagcat
gcaaggaattagaaataggcctagcat
gcaagaaatcagaactagccctagcat

Example Closest String

gctaggagtcagaagtaggcgttgcat
gcaatgaatcagaactgggcctagcat
gctagggatcagaactaggcctagcat
gcaaggaatcataactaggcctagcat
gcaaggaattagaaataggcctagcat
gcaagaaatcagaactagccctagcat

---

gcaaggagtcagaactaggcctagcat

An Algorithm for Closest String

Input: Strings $s_1, \ldots, s_k$, a number m.

Algorithm center(s, l) finds out, if there is an s', such that

- $h(s, s') \leq l$
- $h(s', s_i) \leq m$ für $1 \leq i \leq k$

With center we can easily solve the closest string problem:

Just call center(s_1, m)!

An Algorithm for Closest String

We can implement center(s, l) as follows:

Choose some string s_i with $h(s, s_i) > m$.

(If no such string exists, then s is a solution and we answer Yes.)

Choose a set P of $m + 1$ positions, where s and s_i differ.

Try all positions $p \in P$. Each time let s' be the same as s except for position p, where s' coincides with s_i.

Each time call center$(s', l - 1)$. If one of them answers Yes, then answer Yes.

An Algorithm for Closest String

The size of the search tree is at most $(m+1)^m$.

- The root is labeled with m
- Children are labeled with smaller numbers than the parent
- If the label is 0, we find a solution in polynomial time.
- Every node has at most $m+1$ children.

This algorithm is efficient and works well in practice.

An Algorithm for Closest String

The size of t

- The roc
- Children are labeled with smaller numbers than the parent
- If the label is 0, we find a solution in polynomial time.
- Every node has at most $m + 1$ children.

It has been known for a long time that this problem is *fixed parameter tractable*, if *both* k and m are parameters.

This algorithm is efficient and works well in practice.

An Algorithm for Closest String

The size of t

- The roc
- Children are labeled with smaller numbers than the parent
- If
- E

This a

It has been known for a long time that this problem is *fixed parameter tractable*, if *both* k and m are parameters.

For applications m is the crucial parameter.

Nevertheless, it is also interesting to consider the parameter k.

Question: Is Closest String fixed parameter tractable, if k is the parameter?

(Both questions, for k and m, were open for a long time.)

Analysis of Bounded Search Tree Algorithms

If

1. the root of a tree is labeled with k,
2. every node has at most two children,
3. no label is negative,
4. children are labeled with smaller numbers than the parent,

then it is quite clear that the tree has at most 2^k many leaves.

How can we generalize this obvious fact?

Branching vectors

If every inner node has two children and their labels are exactly one smaller, we get the recurrence relation

$$B_k = B_{k-1} + B_{k-1}.$$

The corresponding branching vector is $(1, 1)$.

A recurrence

$$B_k = B_{k-z_1} + B_{k-z_2} + \cdots + B_{k-z_m}$$

corresponds to the branching vector $(z_1, \ldots, z_m)$.

We can succinctly describe bounded search trees with branching vectors.

Branching Vectors

If the two branching vectors $(1, 1)$ and $(2, 2, 3)$ occur in a bounded search tree algorithms, we get the recurrence

$$B_k = \max\{2B_{k-1}, 2B_{k-2} + B_{k-3}\}.$$

We would like to analyse bounded search tree algorithms with multiple branching vectors. For this end we have to solve recurrences as above.

Linear Recurrence Equations with Constant Coefficients

For a branching vector the corresponding recurrence is a linear recurrence equation with constant coefficients.

Its general form is

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \text{ für } n \geq t.$$

We develop a simple method to solve such recurrence equations.

Linear Recurrence Equations with Constant Coefficients

Let us assume there is a solution of the form $a_n = \alpha^n$, where $\alpha \in \mathbf{C}$ can be a complex number. If we insert this solution into the recurrence and set $n = t$, we get

$$\alpha^t = c_1\alpha^{t-1} + c_2\alpha^{t-2} + \cdots + c_{t-1}\alpha + c_t$$

meaning that α is a root of the *characteristic polynomial*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1}z - c_t.$$

Linear Recurrence Equations with Constant Coefficients

On the other hand, $a_n = \alpha^n$ is a solution of the recurrence, if α is a root of

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t.$$

This is easy to see if we insert it into the recurrence:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t}$$

Linear Recurrence Equations with Constant Coefficients

If α is a k-fold root of χ, then $a_n = n^j \alpha^n$ for $0 \leq j < k$ are also solutions of the recurrence. We can check this again by inserting it into the recurrence:

$$n^j \alpha^n = \sum_{r=1}^{t} c_r (n-r)^j \alpha^{n-r} \text{ resp. } n^j \alpha^t - \sum_{r=1}^{t} c_r (n-r)^j \alpha^{t-r} = 0.$$

The left hand side is a linear combination of $\chi(\alpha)$, $\chi'(\alpha)$, $\chi''(\alpha)$, $\ldots$, $\chi^{(j)}(\alpha)$. The first $k-1$ derivatives of χ become 0 at α because α ist a k-fold root of χ.

Linear Recurrence Equations with Constant Coefficients

Theorem

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \ for\, n \geq t$$

has the solutions $a_n = n^j \alpha^n$, *for every root* α *of the characteristic polynomial*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

and for all $j = 0, 1, \ldots, k-1$, *where* k *is the order of the root* α. *All these solutions are linearly independent. They form a base of the vector space of solutions.*

The Size of Search Trees

Theorem

A bounded search tree with branching vector $(r_1, \ldots, r_m)$*, whose root is labeled with* k*, has size*

$$k^{O(1)}\alpha^k,$$

where α *is the root with biggest absolute value of the characteristic polynomial*

$$\chi(z) = z^t - z^{t-r_1} - z^{t-2} - \cdots - z^{t-r_m},$$

where $t = \max\{r_1, \ldots, r_m\}$.

The Size of Search Trees

Example:

The branching vector $(1, 3)$ has the characteristic polynomial

$$z^3 - z^2 - 1.$$

The largest root is approximately 1.465571.

The size of search tree is $O(1.465572^k)$.

The Size of Bounded Search Trees

Another example:

The branching vector $(1, 2, 2, 3, 6)$ has the characteristic polynomial

$$z^6 - z^5 - z^4 - z^4 - z^3 - 1.$$

The largest real root is 2.160912.

The size of the search tree is therefore $O(2.160913^k)$.

The Reflected Characteristic Polynomial

To determine the characteristic polynomial

$$z^6 - z^5 - z^4 - z^4 - z^3 - 1$$

from the branching vector

$$(1, 2, 2, 3, 6)$$

is not easy and error-prone.

The reflected characteristic polynomial is

$$1 - z - z^2 - z^2 - z^3 - z^6.$$

The Reflected Characteristic Polynomial

Theorem

The characteristic polynomial has a root α iff the reflected characteristic polynomial has the root $1/\alpha$.

The Reflected Characteristic Polynomial

Theorem

A search tree with branching vector $(r_1, \ldots, r_m)$*, whose root is labeled with* k*, has the size*

$$k^{O(1)}\alpha^{-k},$$

where α *is the root with minimum absolute value of the reflected characteristic polynomial*

$$\chi(z) = 1 - z^{r_1} - z^{r_2} - \cdots - z^{r_m}.$$

Branching Numbers

Definition

For each branching vector there is a corresponding branching number which is the reciprocal of the smallest root of the characteristic polynomial.

Theorem

A search tree with branching number α whose root is labeled k has size

$$k^{O(1)}\alpha^k.$$

If the root is simple then the size is $O(\alpha^k)$.

Branching Numbers — Example 1

- Consider a very simple algorithm for Vertex Cover.
- The branching vector is $(1, 1)$.
- The reflected characteristic polynomial is $1 - 2z$.
- The branching number is 2.
- The size of the search tree is $O(2^k)$.

Branching Numbers — Example 2

- If all nodes of a graph have degree 2 or lower, we can find an optimal vertex cover in polynomial time.
- An improved algorithm can choose a node for branching with degree at least 3.
- This gives us the branching vector $(1, 3)$.
- The corresponding branching number is 1.465571.
- The size of the search tree is $O(1.465572^k)$.

Multiple Branching Vectors

Theorem

Let M be a set of branching vectors. A search tree whose branchings correspond to some branching vector from M each and whose root is labeled with k has size

$$k^{O(1)}\alpha^k,$$

where α is the biggest branching number of all branching vectors in M.

Problem Kernels

Let L be a parameterized problem.

Sometimes you can answer the question $(w, k) \in L$ as follows:

- If k is very big, use brute force.
- If k is small and w is complicated, then (w, k) cannot be a solution.
- If k is small and w is simple, then we can easily solve $(w, k) \in L$.

Problem Kernels

Definition

A function $f : \Sigma^* \times \mathbf{N} \to \Sigma^* \times \mathbf{N}$ is a reduction to a problem kernel for a parameterized problem L, if

- $(w, k) \in L$ iff $f(w, k) \in L$,
- there is a function $f' \colon \mathbf{N} \to \mathbf{N}$, such that $|w'| \leq f'(k)$, if $f(w, k) = (w', k')$,
- f can be computed in polynomial time.

In a nutshell: A reduction to a problem whose size is limited by a function of the parameter.

Example Vertex Cover

Assume some graph has a vertex cover of size k.

Let v be a vertex whose degree is at least $k + 1$.

Question:

Must v belong to the vertex cover of size k?

Reduction to a problem kernel:

If there is a node with degree $> k$, remove it. The original graph has a VC of size k iff the reduced graph has a VC of size $k - 1$.

Example Vertex Cover

Assume some graph has a vertex cover of size k.

Let v be a vertex whose degree is at least $k + 1$.

Question:

Must v belong to the vertex cover of size k?

Reduction to a problem kernel:

If there is a node with degree $> k$, remove it. The original graph has a VC of size k iff the reduced graph has a VC of size $k - 1$.

Example Vertex Cover

Question:

How big is the resulting graph at most?

(if we also remove isolated vertices)

Answer:

- The vertex cover itself consists of only k nodes.
- Each of these k nodes can have at most k neighbors.
- There can be at most $k(k+1)$ nodes in total.

Example Vertex Cover

Question:

How big is the resulting graph at most?

(if we also remove isolated vertices)

Answer:

- The vertex cover itself consists of only k nodes.
- Each of these k nodes can have at most k neighbors.
- There can be at most $k(k+1)$ nodes in total.

A smaller Problem Kernel

Theorem (Nemhauser and Trotter)

Let $G = (V, E)$ be a graph of n nodes and m edges.

It takes only polynomial time to find two disjoint node sets C_0 and V_0 such that

1. *If $D \subseteq V_0$ is a vertex cover of $G[V_0]$, then $D \cup C_0$ is a vertex cover of G.*
2. *There is an optimal vertex cover of G containing all of C_0.*
3. *Every vertex cover of $G[V_0]$ has size at least $|V_0|/2$.*

A smaller Problem Kernel

Theorem (Nemhauser and Trotter)

Let $G = (V, E)$ be a graph of n nodes and m edges.

It takes only polynomial time to find two disjoint node sets C_0 and V_0 such that

1. *If $D \subseteq V_0$ is a* ...*rtex cover of G.*
2. *There is an op* ...*$_0$.*
3. *Every vertex d* ...

If G has a vertex cover of size k, then $|V_0| + |C_0| \leq 2k$

Why???

A smaller Problem Kernel

Theorem (Nemhauser and Trotter)

Let $G = (V, E)$ be a graph of n nodes and m edges.

It takes only polynomial time to find two disjoint node sets C_0 and V_0 such that

1. *If $D \subseteq$ cover o*
2. *There is*
3. *Every vertex*

An optimal vertex cover of $G[V_0]$ combined with C_0 is an optimal vertex cover of G.

Why???

Why???

A smaller Problem Kernel

This results in the following algorithm that reduces (G, k) to $(G[V_0], k')$.

- Compute C_0 and V_0
- Let $k' = k - |C_0|$
- G now has a vertex cover of size k if and only if $G[V_0]$ has a vertex cover of size k'.

If $2k' < |V_0|$, then G cannot have a vertex cover of size k.

A smaller Problem Kernel

The following algorithm solves Vertex Cover:

1. Compute V_0 and C_0
2. Output No if $2(k - |C_0|) < |V_0|$
3. Compute an optimal vertex cover C_1 of $G[V_0]$
4. If $|C_1| + |C_0| \leq k$ output Yes and No otherwise

Running time: $n^{O(1)} + O(k2^k)$

Proof of the Nemhauser–Trotter Theorem

An algorithm that computes C_0 and V_0:

Let $G = (V, E)$, V' be a disjoint copy of V, and $G_B = (V, V', E_B)$ be the bipartite subgraph such that

$$\{x, y'\} \in E_B \iff \{x, y\} \in E.$$

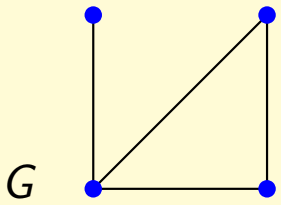
- Compute an optimal vertex cover C_B for G_B.
- Let $C_0 = \{\, x \mid x \in C_B \text{ and } x' \in C_B \,\}$.
- Let $V_0 = \{\, x \mid \text{ either } x \in C_B \text{ or } x' \in C_B \,\}$.

Proof of the Nemhauser–Trotter Theorem

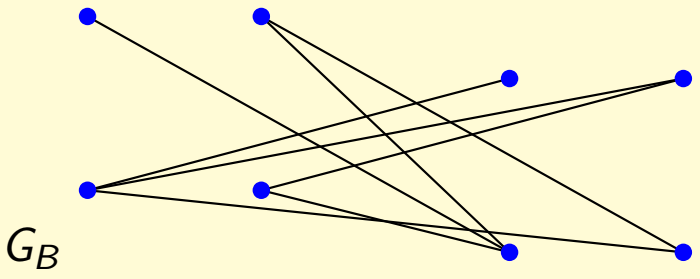
An algorithm that computes C_0 and V_0:

Let $G = (V, E)$, V' be a
be the bipartite subgraph

$\{x, y'\}$



- Compute an optima
- Let $C_0 = \{ x \mid x \in C$
- Let $V_0 = \{ x \mid$ eithe

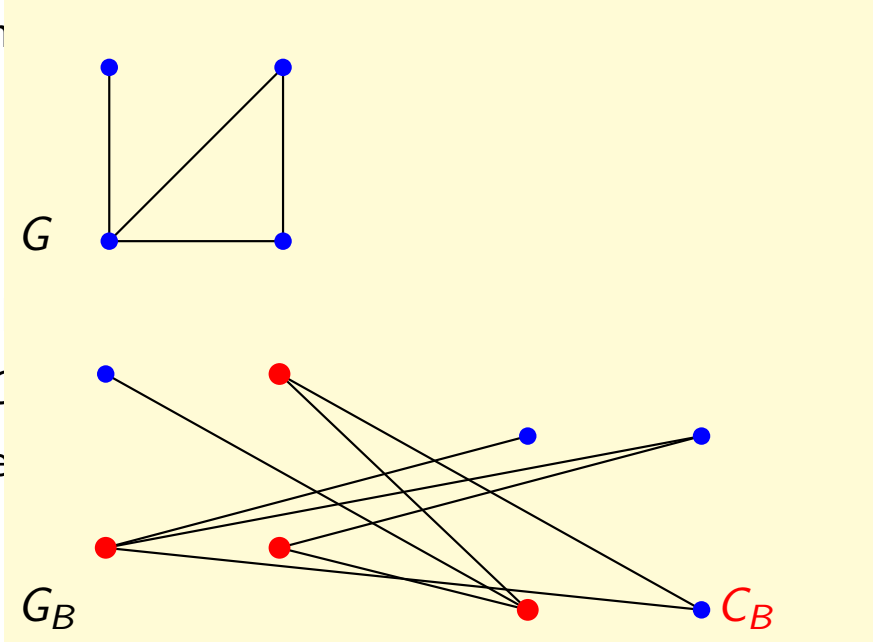


Proof of the Nemhauser–Trotter Theorem

An algorithm that computes C_0 and V_0:

Let $G = (V, E)$, V' be a disjoint copy of V, and $G_B = (V, V', E_B)$ be the bipartite subgraph

$$\{x, y'\}$$



- Compute an optima
- Let $C_0 = \{ x \mid x \in C$
- Let $V_0 = \{ x \mid$ eithe

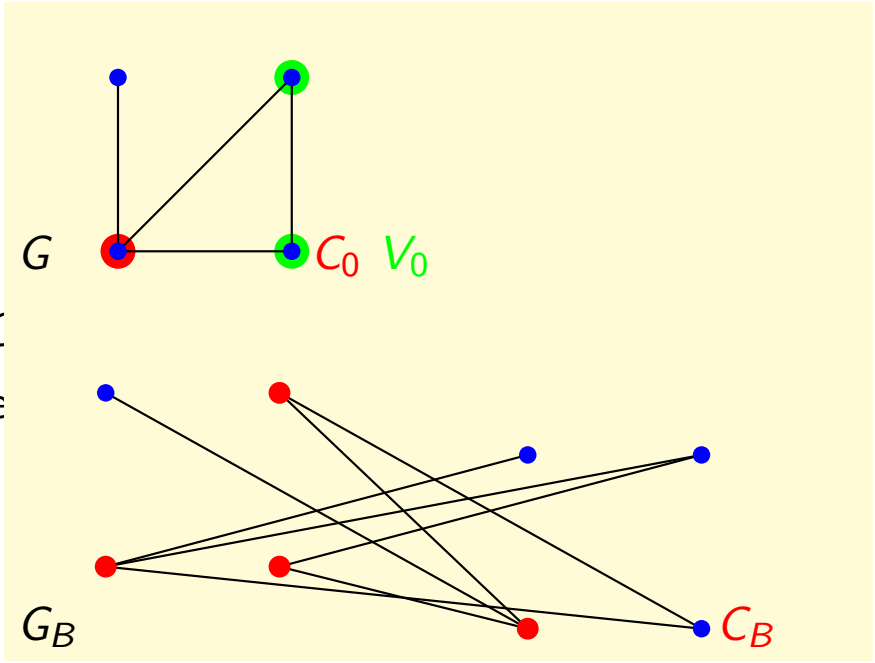
Proof of the Nemhauser–Trotter Theorem

An algorithm that computes C_0 and V_0:

Let $G = (V, E)$, V' be a disjoint copy of V, and $G_B = (V, V', E_B)$ be the bipartite subgraph such that

$$\{x, y'\}$$

- Compute an optima
- Let $C_0 = \{ x \mid x \in C$
- Let $V_0 = \{ x \mid$ eithe



Proof of the Nemhauser–Trotter Theorem

Obviously,

- C_0 and V_0 are disjoint
- C_0 and V_0 can be computed in polynomial time

We need to prove the three statements of the theorem:

1. If $D \subseteq V_0$ is a vertex cover of $G[V_0]$, then $D \cup C_0$ is a vertex cover of G.
2. There is an optimal vertex cover of G containing all of C_0.
3. Every vertex cover of $G[V_0]$ has size at least $|V_0|/2$.

Statement 1

Claim: If $D \subseteq V_0$ is a vertex cover of $G[V_0]$, then $D \cup C_0$ is a vertex cover of G.

Let $D \subseteq V_0$ a vertex cover of $G[V_0]$ and $e = \{x, y\} \in E$ an arbitrary edge.

Let $I_0 = V - V_0 - C_0$.

- If an endpoint of e is in $C_0 \ldots$ okay
- If both endpoints are in $V_0 \ldots$ okay
- $x \in I_0 \Rightarrow y, y' \in C_B \Rightarrow y \in C_0, \ldots$ okay

Statement 2

Claim: There is an optimal vertex cover of G containing all of C_0.

Let S an optimal vertex cover and $S_V = S \cap V_0$, $S_C = S \cap C_0$, $S_I = S \cap I_0$, $\bar{S}_I = I_0 - S_I$.

Lemma

$(V - \bar{S}_I) \cup S'_C$ is a vertex cover of C_B.

Proof

Let $\{x, y'\} \in E_B$.

If $x \notin \bar{S}_I$, then $x \in (V - \bar{S}_I) \cup S'_C$.

If $x \in \bar{S}_I$, then $x \in I_0$, $x \notin S \Rightarrow y \in S$, $y, y' \in C_B \Rightarrow$
$\Rightarrow y \in C_0 \Rightarrow y \in S \cap C_0 = S_C \Rightarrow y' \in S'_C$.

Statement 2

$$
\begin{aligned}
|V_0| + 2|C_0| &= |V_0 \cup C_0 \cup C_0'| \\
&= |C_B| \\
&\leq |(V - \bar{S}_I) \cup S_C'| \text{ due to the lemma} \\
&= |V - \bar{S}_I| + |S_C'| \\
&= |V_0 \cup C_0 \cup I_0 - (I_0 - S_I)| + |S_C'| \\
&= |V_0| + |C_0| + |S_I| + |S_C|
\end{aligned}
$$

It follows that $|C_0| \leq |S_I| + |S_C| = |S| - |S_V|$ and thus $|C_0 \cup S_V| \leq |S|$.

Statement 3

Claim: Every vertex cover of $G[V_0]$ has size at least $|V_0|/2$.

Let S_0 an optimal vertex cover of $G[V_0]$.

$C_0 \cup C_0' \cup S_0 \cup S_0'$ is a vertex cover of G_B, because $C_0 \cup S_0$ is a vertex cover of G.

$$|V_0| + 2|C_0| = |C_B| \leq |C_0 \cup C_0' \cup S_0 \cup S_0'| = 2|C_0| + 2|S_0|$$

The claim follows.

Graph Properties

Definition

A graph property Π is a class of graphs that is closed under graph isomorphisms.

That is, if two graphs G_1 and G_2 are isomorphic, both belong to Π or both don't.

Graph Properties

Example

- Connected graphs
- Trees
- Graphs containing a clique of size 100
- Planar graphs
- Regular graphs
- Finite graphs

Graph Properties

These are not graph properties:

- Graphs whose nodes are natural numbers
- Every nonemtpy finite set of graphs
- (For Logicians: Each set of graphs)

Hereditary Graph Properties

A graph property Π is called hereditary if the following holds:

Let $G \in \Pi$ and H be an induced subgraph of G.

Then $H \in \Pi$ as well.

In other words: Π is closed under taking induced subgraphs.

Hereditary Graph Properties

A graph property Π is called hereditary if the following holds:

Let $G \in \Pi$ and H be an induced subgraph of G.

Then $H \in \Pi$ as well.

In other wor

Questions:

1. Does the empty graph belong to every hereditary graph property?
2. Are graph properties lattices with respect to the induced subgraph relation?

Hereditary Graph Properties

Which graph properties are hereditary?

- Bipartite graphs
- Complete graphs
- Planar graphs
- Trees
- Connected graphs
- Graphs of diameter at most d
- Regular graphs

Hereditary Graph Properties

Which graph properties are hereditary?

- Forests
- Graphs containing an independent set of size 8
- Graphs with at least 17 nodes
- Graphs containing no matching of size 35
- 5-regular graphs
- Infinite graphs
- Chordal graphs

Characterization by Obstruction Sets

Definition

A graph property Π has a characterization by obstruction sets if there is a graph property $\mathcal{F}$ such that $G \in \Pi$ if and only if $\mathcal{F}$ does not contain an induced subgraph of G.

Characterization by Obstruction Sets

Definition

A graph property Π has a chara if there is a graph property $\mathcal{F}$ suc does not contain an induced subgrap

Question:

Does every hereditary graph property have a characterization by obstruction sets?

Characterization by Obstruction Sets

Definition

A graph property Π has a chara if there is a graph property $\mathcal{F}$ suc does not contain an induced subgrap

Question:

Does every hereditary graph

Answer:

Yes. Choose $\mathcal{F} = \mathcal{G} - \Pi$ with $\mathcal{G}$ containing all graphs.

Finite Obstruction Sets

Definition

A graph property Π has a finite characterization by obstruction sets if it has a characterization by $\mathcal{F}$, and $\mathcal{F}$ contains only a finite number of non-isomorphic graphs.

Finite Obstruction Sets

Which graph properties have a finite characterization by obstruction sets?

- Graphs containing an independent set of size 7?
- Bipartite graphs?
- Forests?
- Planar graphs?
- 5-colorable graphs?
- Graphs containing a vertex cover of size k?

Finite Obstruction Sets

Which graph properties have a finite characterization by obstruction sets?

- Triangle-free graphs?
- Graphs without any k-cliques?
- Graphs of diameter at most d?
- Cycle-free graphs?
- Graphs not containing any cycle of length k?

Graph Modification Problems

Let Π be a graph property. There are the following well-known graph modification problems for an input G:

1. Edge Deletion Problem: Can we obtain a graph in Π by deleting k edges from G?
2. Node Deletion Problem: Can we obtain a graph in Π by deleting k nodes from G?
3. Node/Edge Deletion Problem: Can we obtain a graph in Π by deleting k nodes and l edges from G?
4. Edge Insertion Problem: Can we obtain a graph in Π by inserting k edges in G?

Generalization

Definition

$\Pi_{i,j,k}$-Graph Modification Problem

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

Parameter: $i, j, k \in \mathbf{N}$

Question: Can we obtain a graph in Π by removing up to i nodes, removing up to j edges, and inserting up to k edges in G?

The Leizhen Cai Theorem

Theorem

Let Π be a graph property with a finite characterization by obstruction sets.

Then the $\Pi_{i,j,k}$-Graph Modification Problem can be solved in $O(N^{i+2j+2k}|G|^{N+1})$ steps and is thus fixed paramater tractable.

N is the number of nodes in the largest graph in the obstruction set, i.e., a constant.

Proof of the Leizhen Cai Theorem

Lemma

Let Π *be a hereditary graph property that can be checked in* $T(G)$ *steps.*

Then it takes $O(|V|T(G))$ *many steps to find a minimal forbidden induced subgraph for any* $G = (V, E) \notin \Pi$.

In this context, the term "minimal" refers to the order "induced subgraph".

Proof of the Leizhen Cai Theorem

Proof of the lemma:

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

$H := G$
for $i = 1, \ldots, n$ **do**
  if $H - \{v_i\} \notin \Pi$ **then** $H := H - \{v_i\}$
od

Upon termination, H is a minimal forbidden induced subgraph.

Proof of the Leizhen Cai Theorem

Input: $G = (V, E)$

Parameter: $i, j, k \in \mathbf{N}$

Question: $G \in \Pi_{i,j,k}$

while $i + j + k > 0$ **do**
  $H :=$ minimal forbidden induced subgraph of G
  Modify G by removing an edge or node or by inserting an edge from/in H
  Let $i := i - 1$, $j := j - 1$, or $k := k - 1$.
  if $G \in \Pi$ **then** answer YES
od

Answer NO

Proof of the Leizhen Cai Theorem

Running time:

Find H: $O(|V| \cdot |V|^N)$ according to the lemma

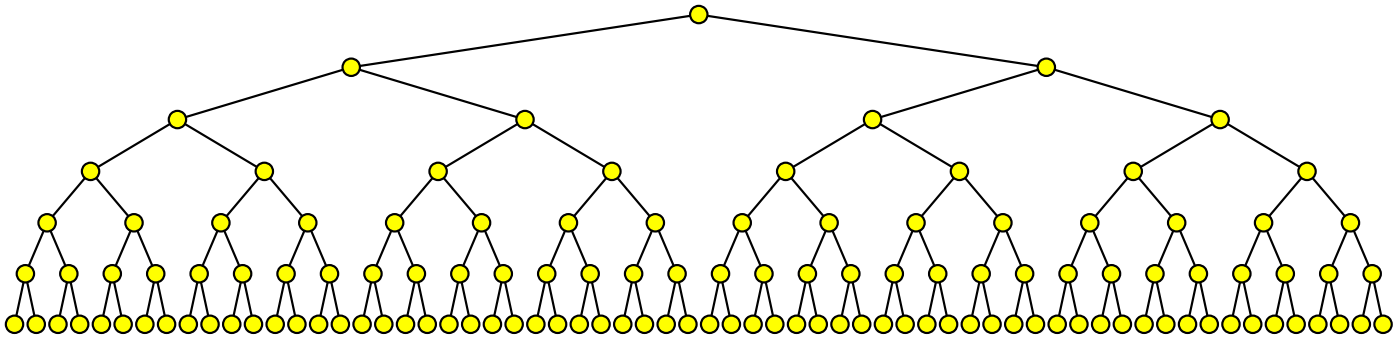
There are only N ways of removing a node from H

There are only $\binom{N}{2}$ ways of deleting or inserting an edge from/in H

Total running time

$$O(N^{i+2j+2k}|V|^{N+1}).$$

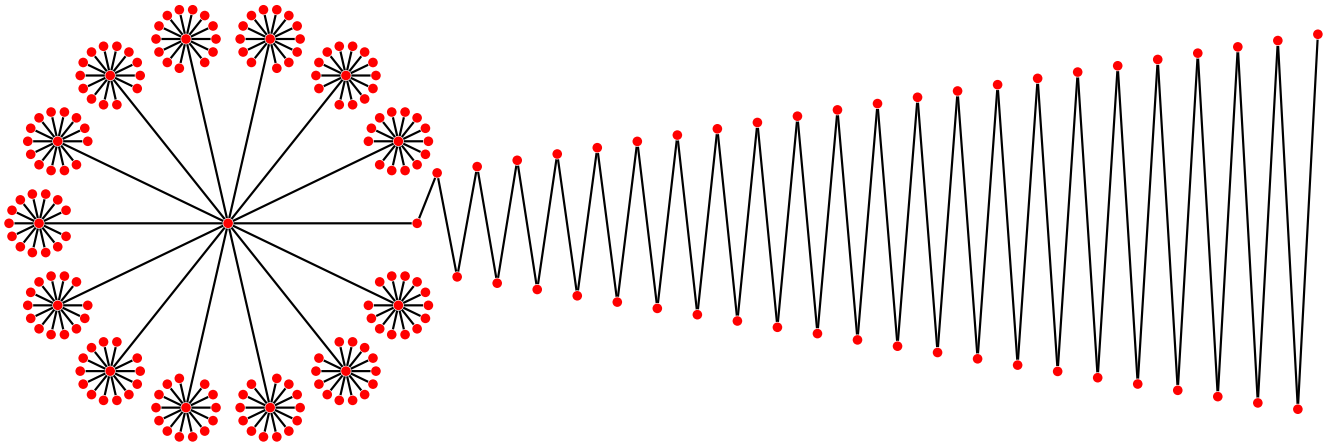
Interleaving — Search Trees and Problem Kernels



In a search tree, the parameter decreases as it approaches the leaves, whereas this is not necessarily the case for the size of the instance (which could even grow).

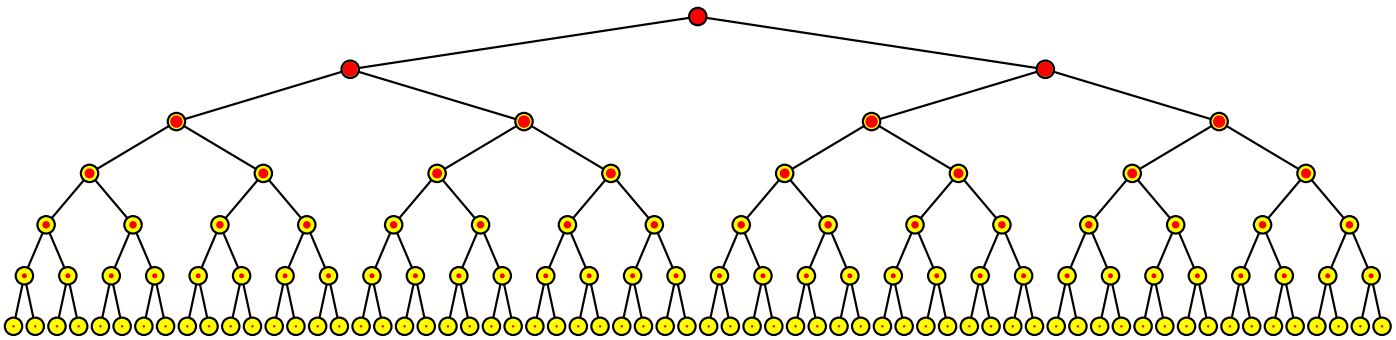
If the expansion of a node in the search tree takes $p(n)$ steps, then the total running time becomes $O(s(k, n)p(n))$, where $s(k, n)$ denotes the size of the search tree.

Interleaving



The size of this graph never drops below $n/2$ within the entire search tree of a vertex cover algorithm, provided that no reduction to the problem kernel is performed.

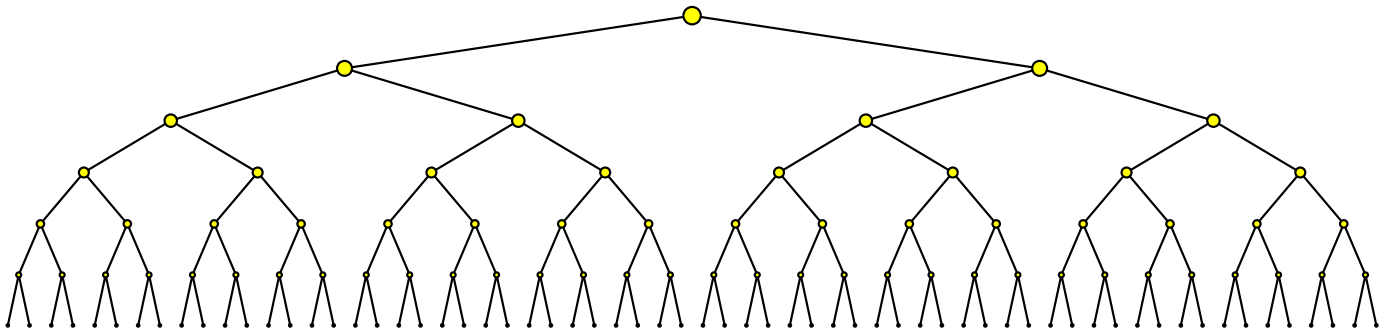
Interleaving



The size of the parameter is reflected by the size of the red dots. The recursion stops when the parameter is small. In the leaves, the parameter is bounded by a constant.

Nearly all nodes are close to a leaf $\Longrightarrow$ nearly all nodes have a small parameter.

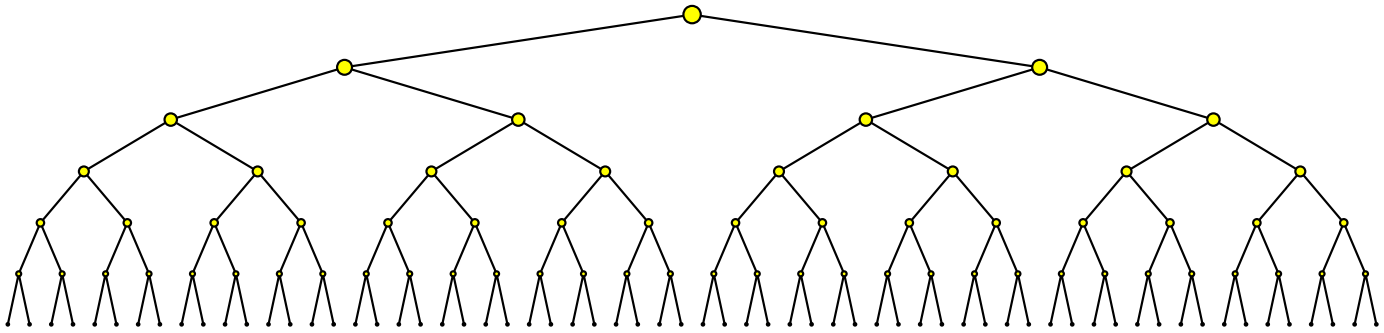
Interleaving



If we perform a reduction to problem kernel after each expansion of a node in the search tree, then the instances decrease in size as we approach the leaves.

A more detailed analysis reveals that the total running time is only $O(s(n,k) + t(n) + r(n))$ rather than $O(s(n,k)t(n))$, where $r(n)$ denotes the time required for the reduction to problem kernel.

Search Trees and Dynamic Programming



If all nodes are close to leaves and there are many of them, then some must be identical.

$\Longrightarrow$ We can improve the running time by computing the respective solutions only once and storing them in a database.

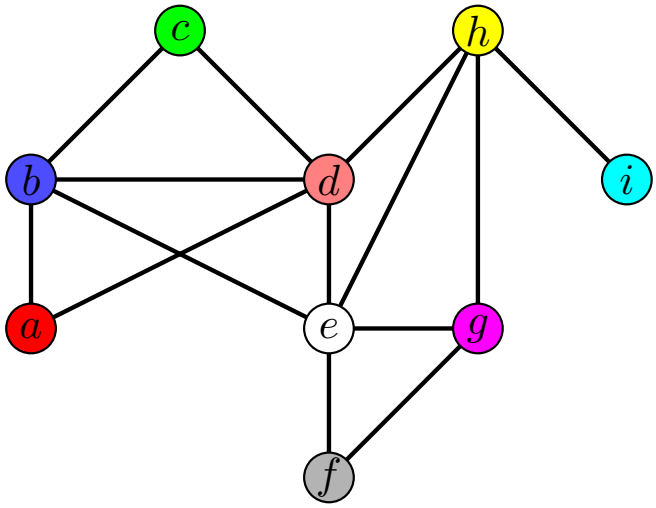
Search Trees and Dynamic Programming

Example: Vertex Cover and the 2^k algorithm

- Each node of the search tree is an induced subgraph.
- After a reduction to problem kernel, the size of a graph is bounded by $2k'$ from above if we are looking for a solution of size k'.
- There are at most $O\left(\binom{2k}{2k'}\right)$ induced subgraphs of size at most $2k'$.
- The running time is $O\left(2^{k-k'}\binom{2k}{2k'}\right)$ if we store solutions of size $2k'$ in the database.
- If we choose the value of k' optimally, the running time becomes 1.886^k.

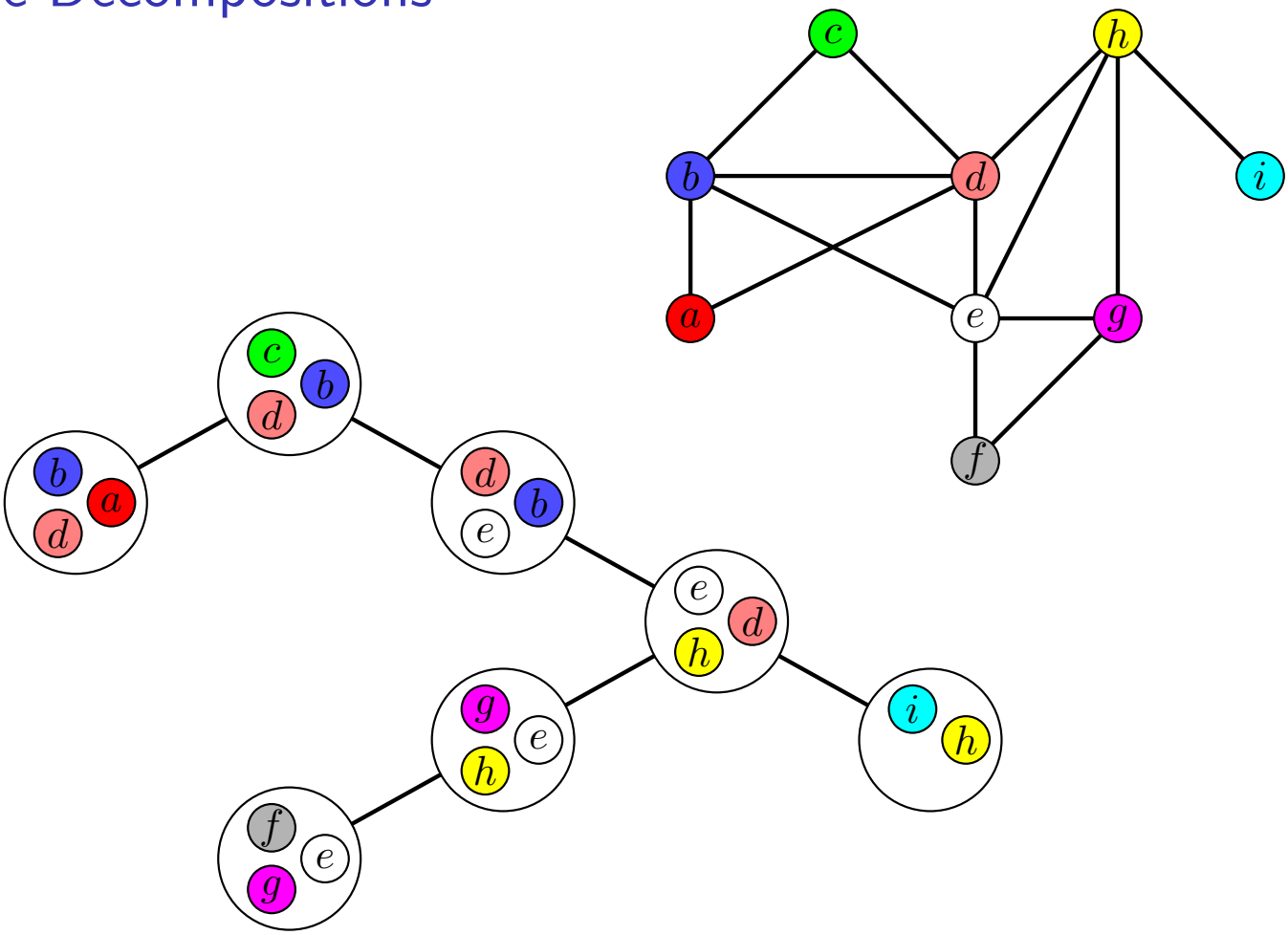
Tree Decompositions

A tree decomposition of a graph G is a tree, whose nodes are called bags. Every bag is a set of nodes from G.

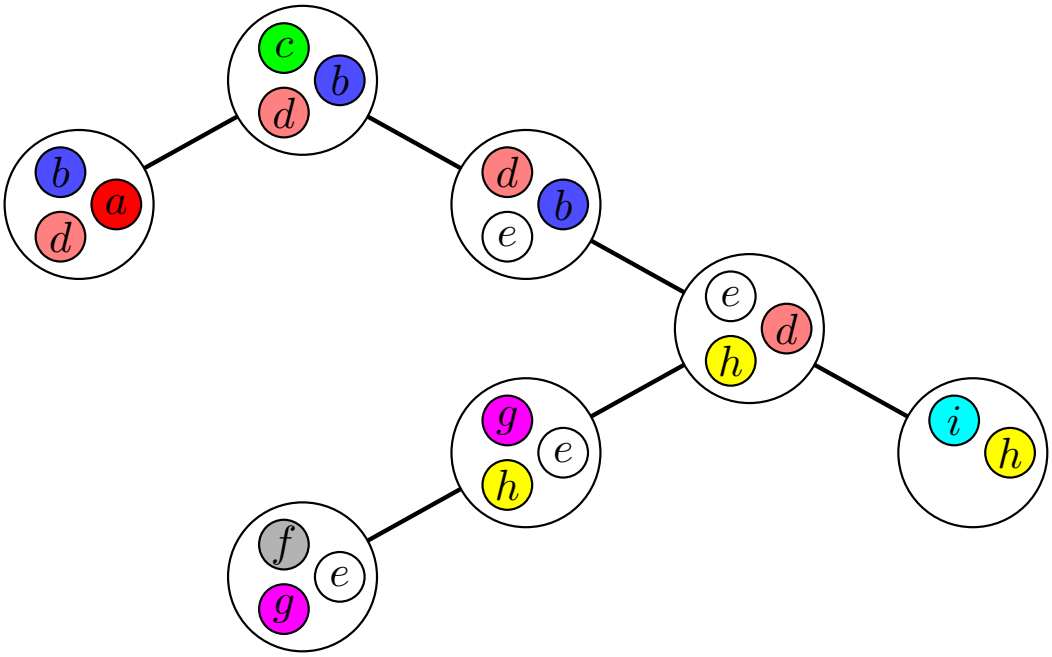


- Any node and any edge from G ist contained in at least one bag.
- A node contained in two bags A, B must be contained in any bag between A and B.

Tree Decompositions



Tree Decompositions and Treewidth

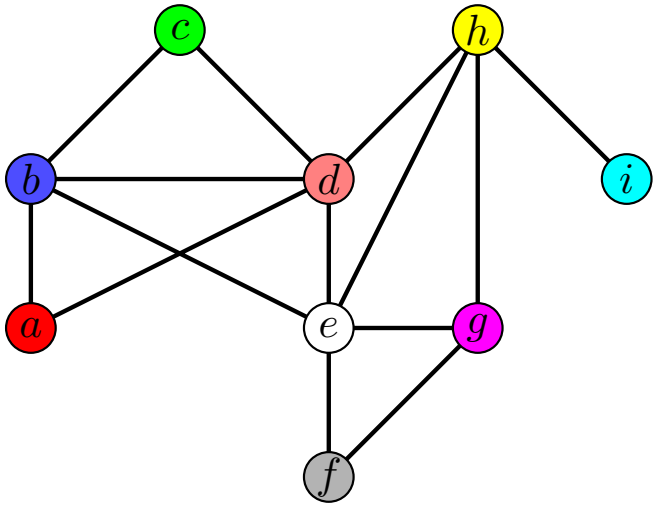


The width of a tree decompostion is the size of the largest bag minus 1.

$\Rightarrow$ Here, the treewidth is 2.

Tree Decompositions and Treewidth

Alternative definition:



The treewidth of G is the minimum number of cops, needed to catch a robber in G, minus 1.

Tree Decompositions and Treewidth

Given a tree decomposition of G with width w, many optimization problems on G can be solved in time $c^w \cdot poly(n)$ using dynamic programming on the tree decomposition.

Many problems can be solved fast, if a tree decomposition of small width can be found.

General Result

Any problem with the following properties is fixed parameter tractable:

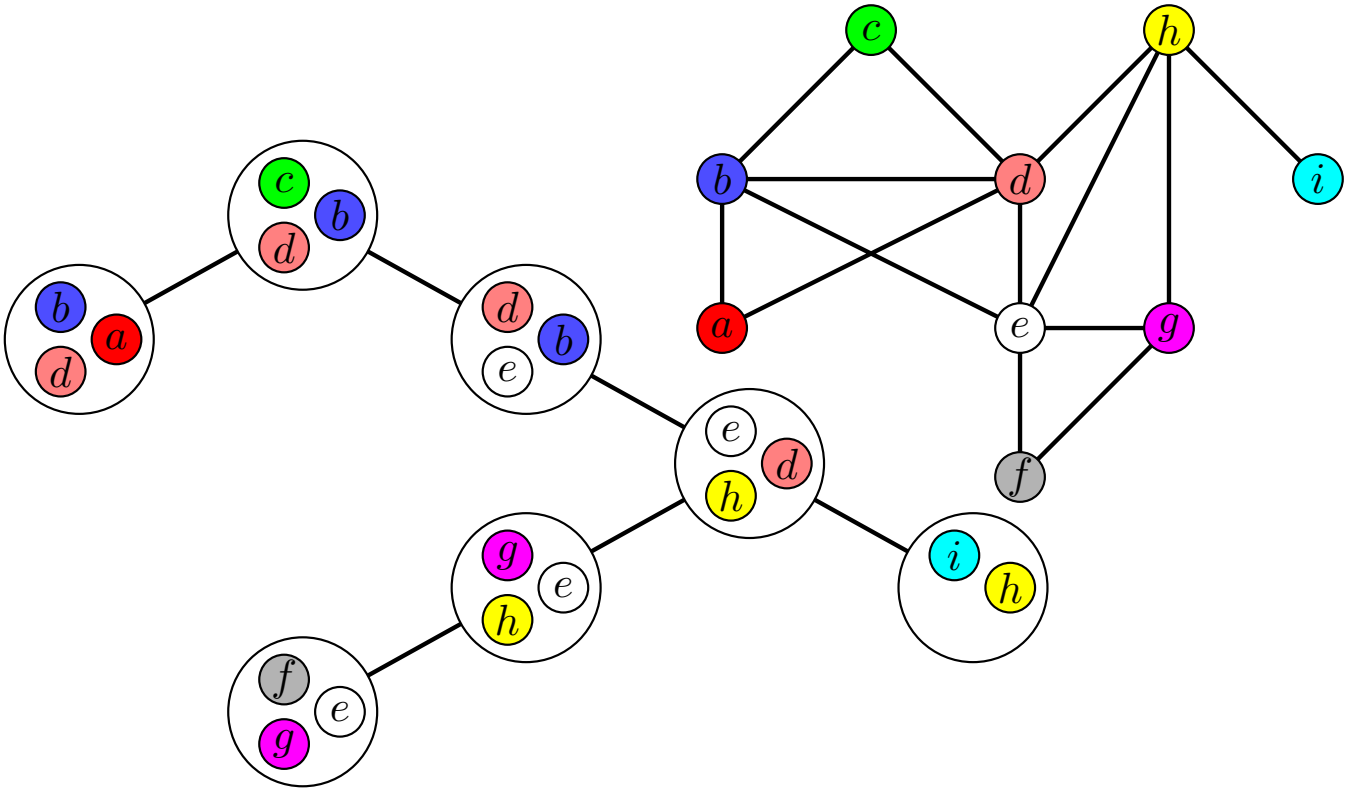
- Let $G = (V, E)$ a planar graph and k a number. Question: Exists some $S \subseteq V$ of size k with a certain property (e.g. S is a vertex cover).
- There is a constant c such that the distance between any node and S is bounded by c.
- Given a tree decomposition of width w, the problem can can be solved in $f(w)n^{O(1)}$ steps.

Special cases are Vertex Cover, Independent Set, and Dominating Set.

Proof Idea

- Since any node is at most c steps away from a node in S, there is no path of length more than $2c|S|$.
- Hence, the diameter is $O(k)$, if there exists some S of size k.
- The treewidth of a planar graph with diameter d is at most $3d$ (without proof).
- If the diameter is larger than $2ck$, the output is no.
- Otherwise, we obtain a tree decomposition of width $6ck$ and can use it to solve the problem.

Dynamic Programming on a Tree Decomposition

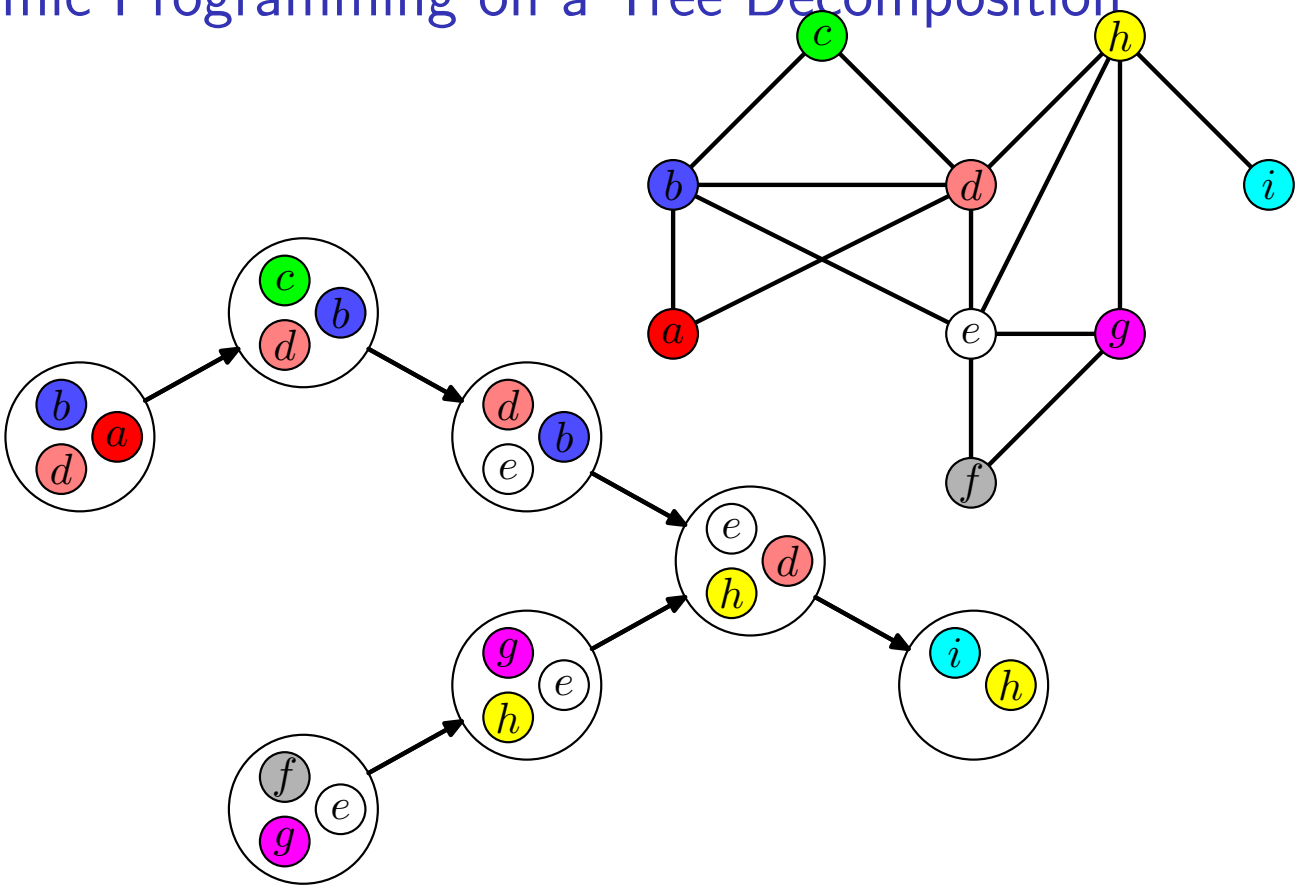


Dynamic Programming on a Tree Decomposition

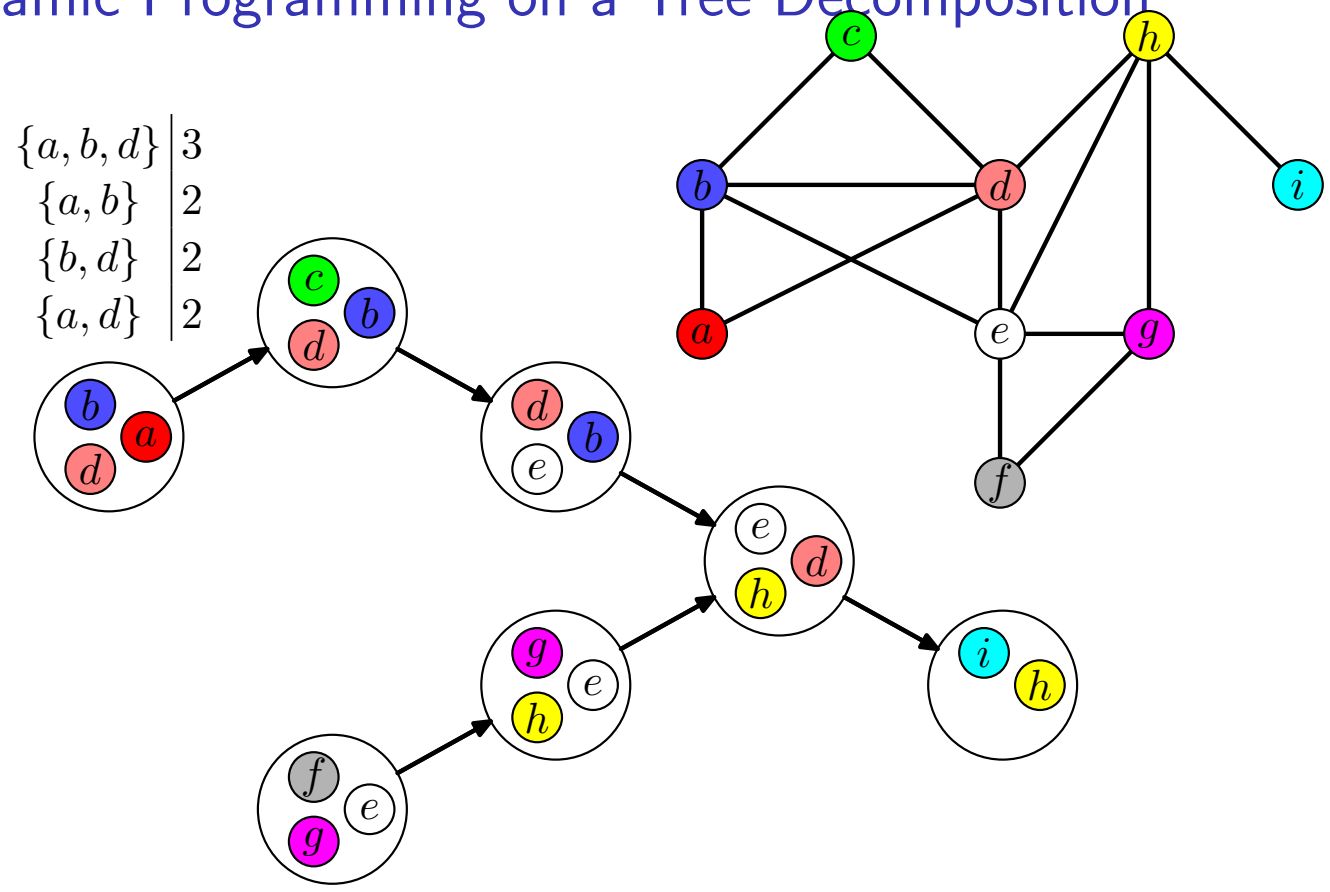
General approach:

- The tree decomposition is transfered into a rooted tree: An arbitrary node becomes the root. Children point to their parents.
- A bag represents the subgraph induced by its children.
- For any bag a table is calculated, showing the optimal solutions for its subgraphs.
- The children's tables are calculated first.

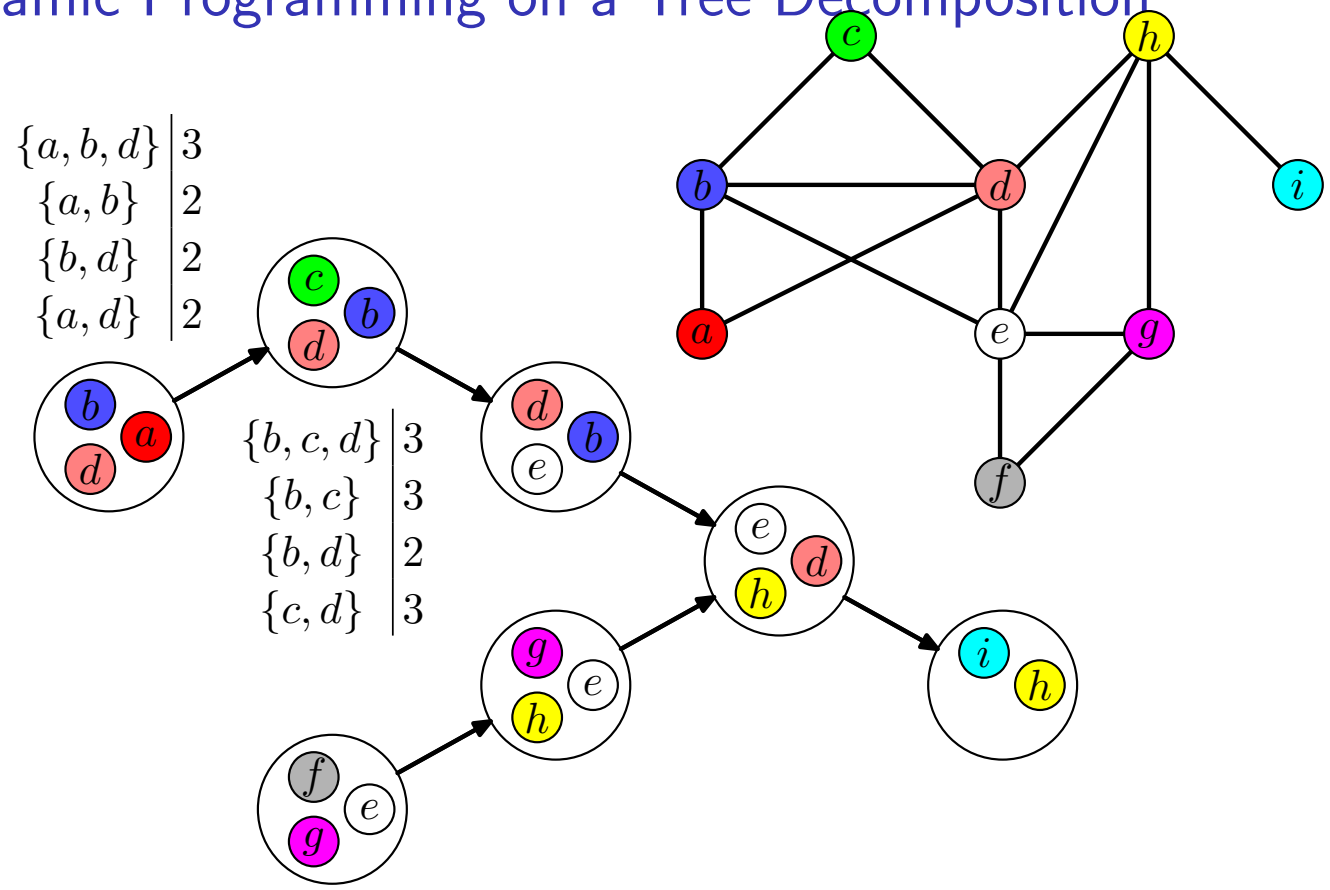
Dynamic Programming on a Tree Decomposition



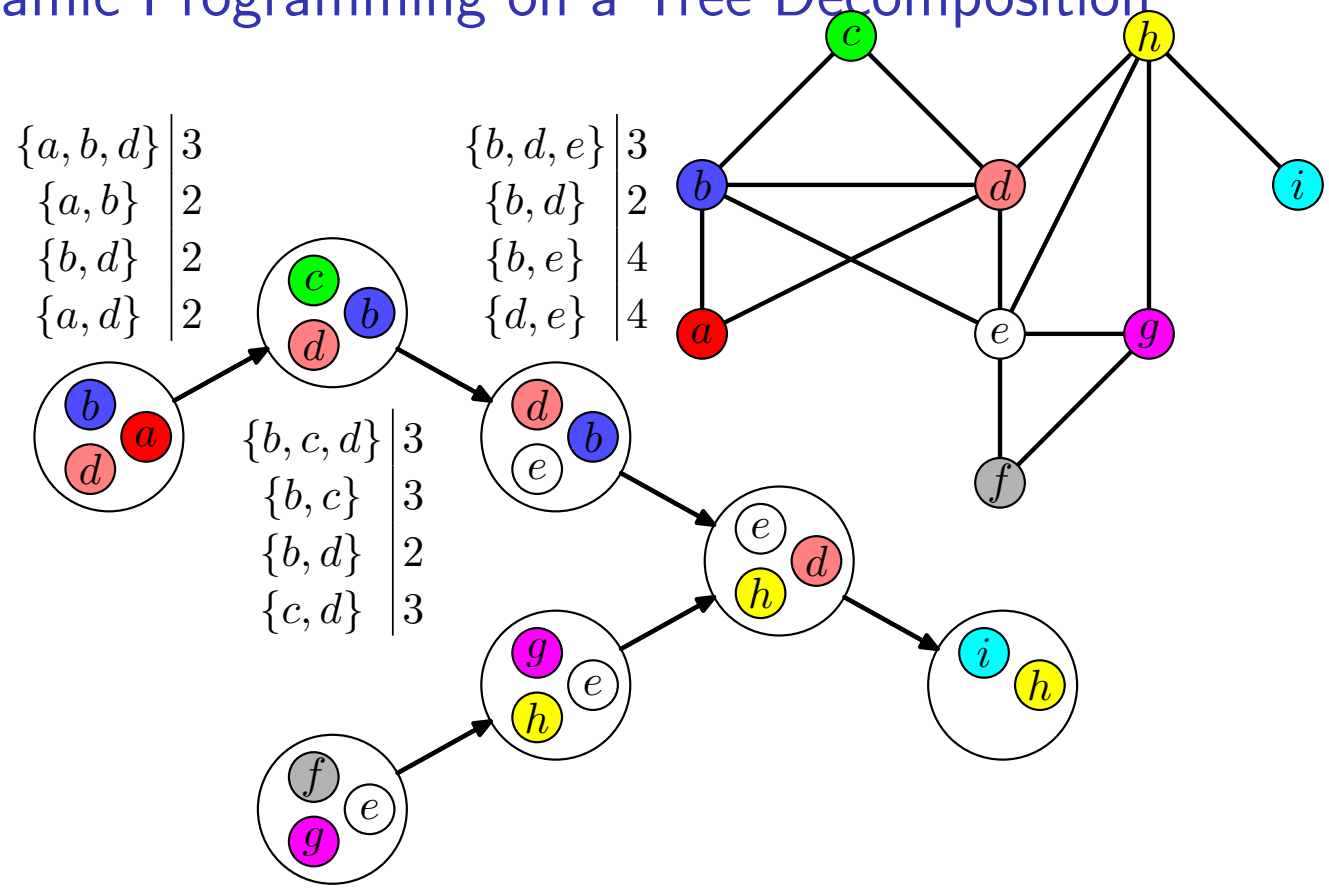
Dynamic Programming on a Tree Decomposition



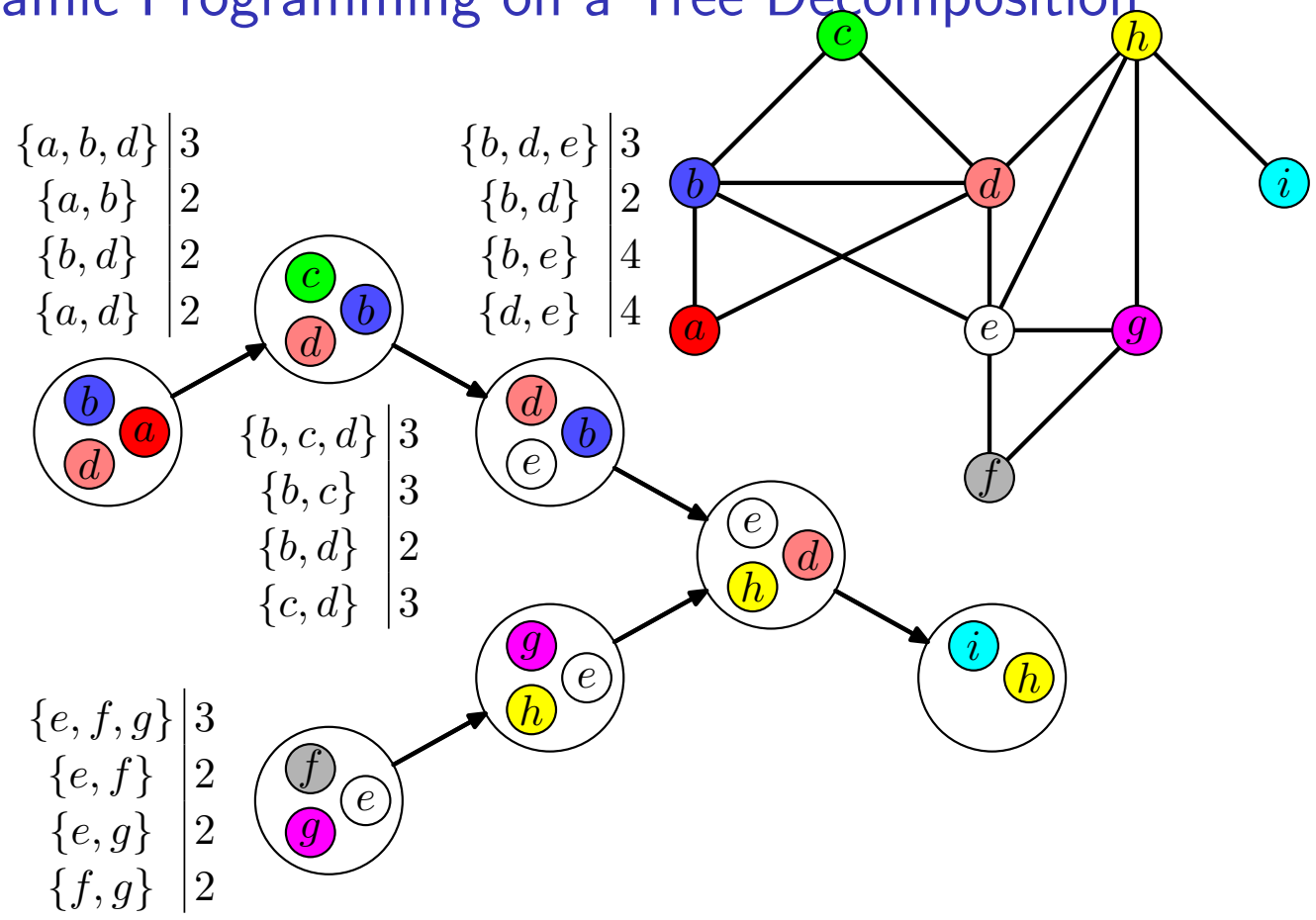
Dynamic Programming on a Tree Decomposition



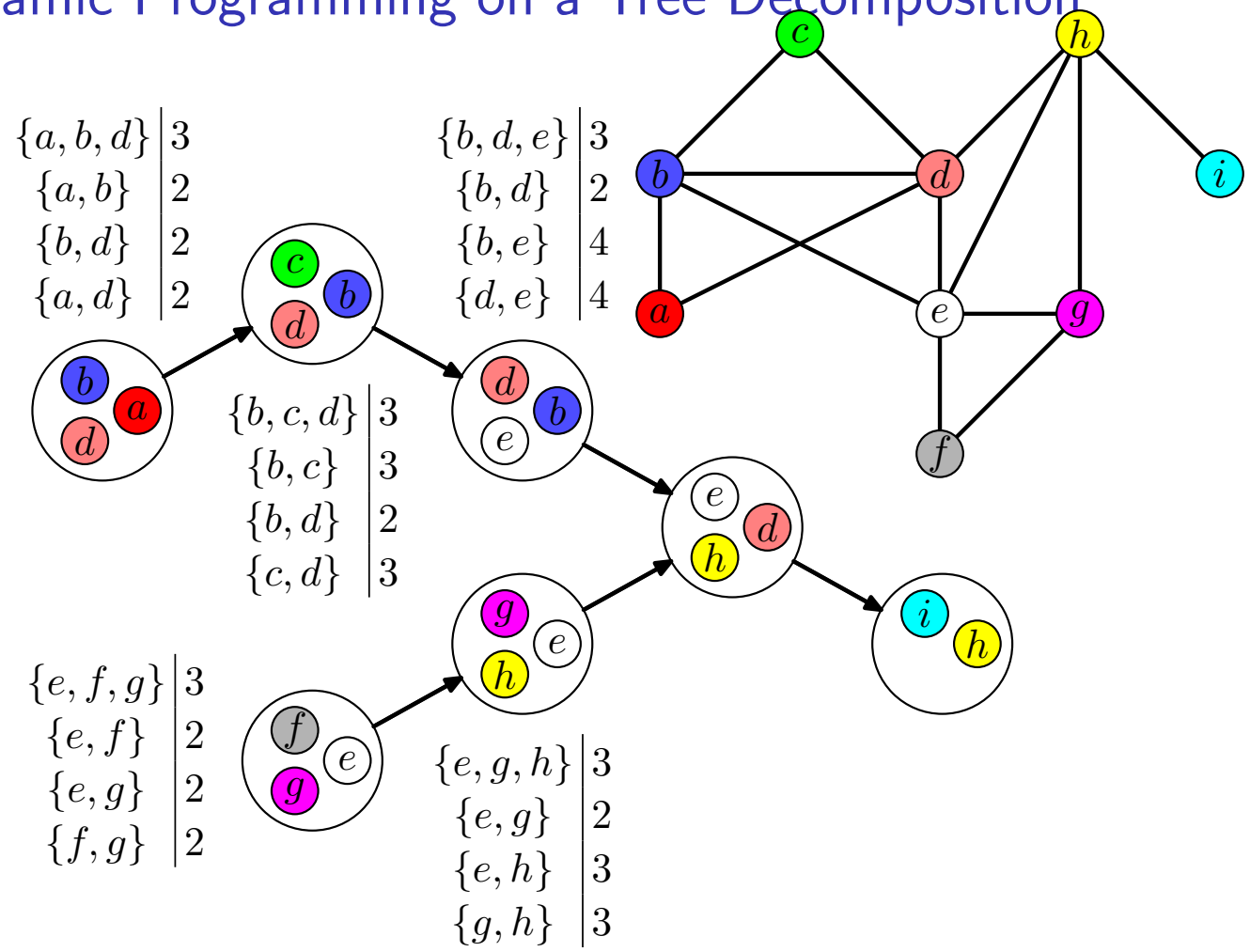
Dynamic Programming on a Tree Decomposition



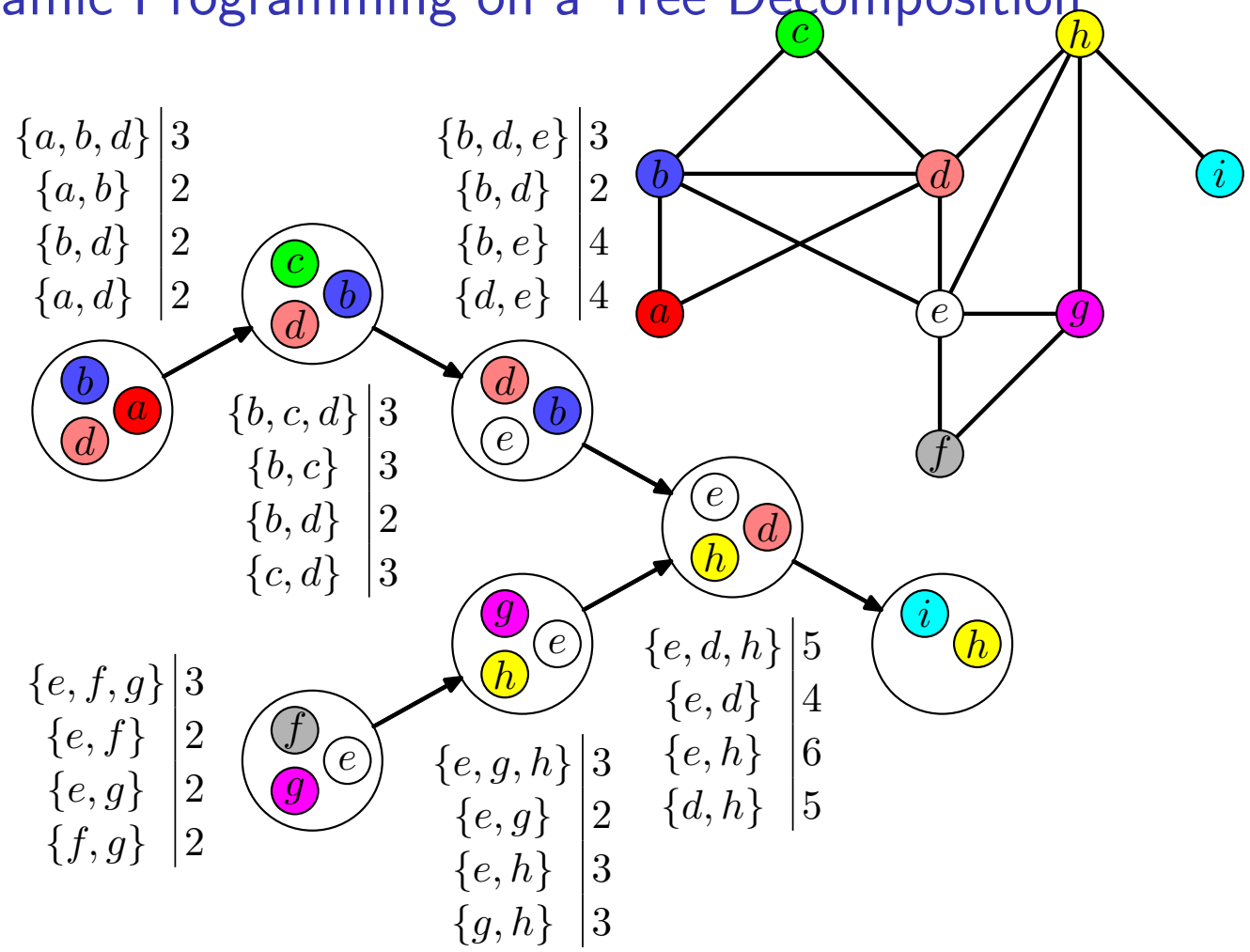
Dynamic Programming on a Tree Decomposition



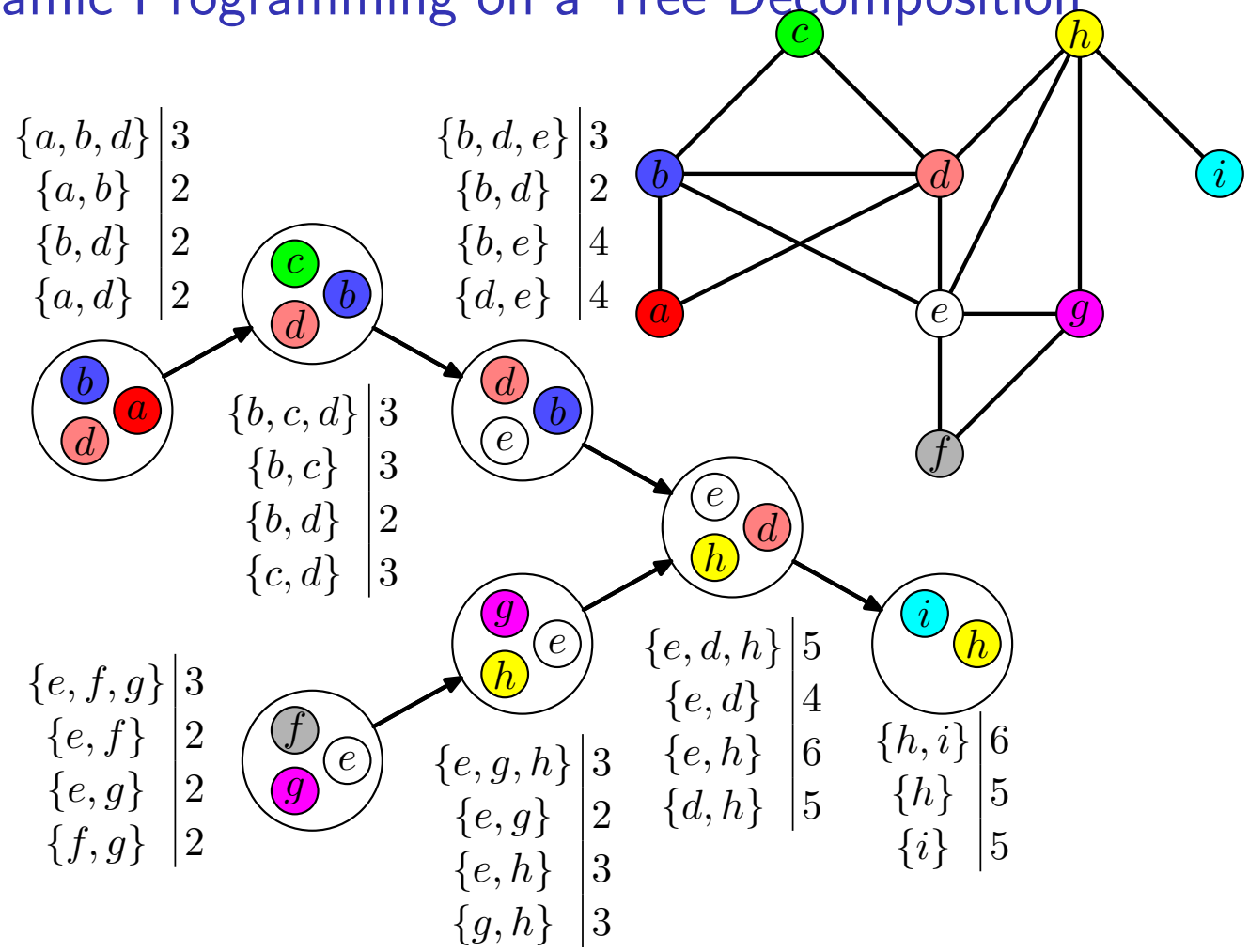
Dynamic Programming on a Tree Decomposition



Dynamic Programming on a Tree Decomposition



Dynamic Programming on a Tree Decomposition



Extended Monadic Second Order Graph Theory

We introduce a new logic called MSO_2-logic.

This logic contains variables for nodes, edges, sets of nodes, and sets of edges.

There are the quantifiers $\exists$ and $\forall$ and operators $\wedge$, $\vee$ and $\neg$.

Furthermore, the following relations are included:

$$u \in U,\ d \in D,\ inc(d, u),\ adj(u, v),\ \cdot = \cdot$$

where u, v are node variables, d is an edge variable, U is a node set variable, and D is an edge set variable.

Extended Monadic Second Order Graph Theory

A graph can either satisfy a formula or not. This allows for a description of graph classes by formulas.

Example

Which graphs satisfy the following formula:

$$\exists u \exists v \exists w (adj(u, v) \wedge adj(u, w) \wedge adj(v, w))$$

Is there a formula describing bipartite graphs ?

Courcelle's Theorem

Theorem

Let $\mathcal{G}$ be a graph class, described by a formula in MSO_2-logic.

The following proplem is fixed parameter tractable:

Input: Graph G with treewidth k

Parameter: k

Question: Does G belong to $\mathcal{G}$

Proof

difficult. . .

Courcelle's Theorem

Theorem

Let $\mathcal{G}$ be a graph class, described by a formula in MSO_2-logic.

The following proplem is fixed parameter tractable:

Input: Graph G with treewidth k

Parameter: k

Question: Does G belong to $\mathcal{G}$

Proof

difficult...

Courcelle's Theorem

We could solve VertexCover, when parameterized by treewidth.

Using Courcelle's Theorem, the size of the formula depends on the size of the vertex cover, we are searching:

$$\exists v_1 \ldots \exists v_k \forall e (inc(e, v_1) \vee \cdots \vee inc(e, v_k)$$

Why constitutes this a problem?

Courcelle's Theorem (Extension)

We extend the MSO_2-logic:

We allow the following new quantifier:

$$\min U \; \forall e \exists u (u \in U \wedge inc(e, u))$$

Whenever the treewidth is bounded, a minimal set of nodes U, satisfying an arbitrary MSO_2 formula $F(U)$, can be calculated in polynomial time.

Courcelle's Theorem (Extension)

Let G be a graph with edge lables from $\{1, \ldots, c\}$. The corresponding sets of nodes are $V_1, \ldots, V_c\}$.

We can express inclusion in V_i.

Example

$$\max U \subseteq V_1 \forall x \in V_2 \exists y (adj(x, y) \vee y \notin U)$$

(Exists a set U of red nodes, such that any blue node has a neighbor not belonging to U.)

This problem is called Red Blue Nonblocker.

Treewidth and Courcelle's Theorem

A minor of a graph G, is graph obtained from G by contraction of edges and removal of nodes and edges.

Lemma

Any planar graph is a minor of a grid.

Proof

Simple. For example by Induction.

Treewidth and Courcelle's Theorem

Theorem

Let H be a finite, planar graph and $\mathcal{G}$ a class of graphs, not containing H as a minor.

Then there is a constant c_H, such that the treewidth of any graph in $\mathcal{G}$ is at most c_H.

Proof

very difficult and long

Treewidth and Courcelle's Theorem

Theorem

Let H be a finite, planar graph and $\mathcal{G}$ a class of graphs, not containing H as a minor.

Then there is a constant c_H, such that the treewidth of any graph in $\mathcal{G}$ is at most c_H.

Proof

very difficult and long

Treewidth and Courcelle's Theorem

Corollary

"A graph with large treewidth contains a large grid":

If $tw(G) > t$ holds, G contains the grid $Q_f(t)$ as minor, where f is a monotone, unbounded function.

Proof

direct consequence of the last theorem

Treewidth and Courcelle's Theorem

Example

Input: A planar graph G and k pairs (s_i, t_i) of nodes from G. (Parameter is k)

Question: Are there edge disjoint paths connecting each s_i with t_i?

The problem belongs to FPT:

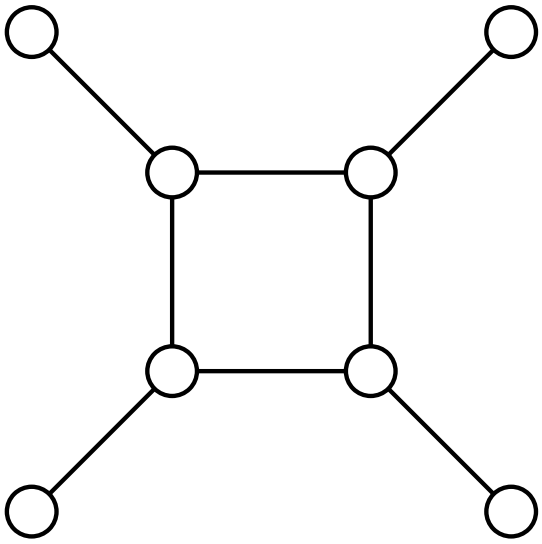
If the treewidth is small, we can apply Courcelle's Theoremn.

If the treewidth is large, a large grid is contained as a minor. Removing a node from this grid does not "harm" this structure.

Color Coding

Problem: Does a given graph contain a cycle of length k?

This problem is NP-complete, because Hamilton Cycle is a special case.



Question: Is it fixed parameter tractable?

Color Coding

Algorithm:

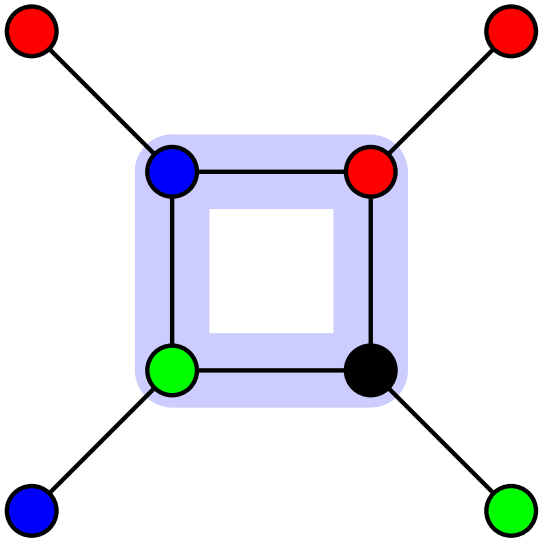
1. Randomly color each node in one of k colors
2. Check for a colorful cycle of length k, i.e., a cycle in which no two nodes have the same color

Analysis:

The probability that a cycle of length k becomes colorful is

$$k!/k^k \sim \sqrt{2\pi k}\, e^{-k}.$$

Color Coding



The cycle is colorful with probability $4!/4^4 = 3/32$.

Color Coding

After using the above algorithm to find a cycle of length k for N times, the probability that it failed to detect a cycle every time is

$$\left(1 - \frac{k!}{k^k}\right)^N.$$

Letting $N = Mk^k/k! \sim Me^k/\sqrt{k}$ yields

$$\left(1 - \frac{M}{N}\right)^N \sim e^{-M}.$$

This failure probability can be made arbitrarily small by the choice of M.

Color Coding

A question remains:

How do you check for a colorful cycle?

Color Coding

Answer:

Create a table $P(u, v, l)$.

$P(u, v, l)$ contains all sets of pairwise distinct nodes that constitute a path from u to v of length l.

$P(u, v, l)$ can be computed from $P(u, v, l-1)$.

Time required: $2^k \cdot poly(n)$

Color Coding

Definition

A k-perfect family of hash functions is a family $\mathcal{F}$ of functions $\{1, \ldots, n\} \to \{1, \ldots, k\}$ such that for every $S \subseteq \{1, \ldots, n\}$ with $|S| = k$ there exists an $f \in \mathcal{F}$ that is bijective when restricted to S.

Let us first assume we had such a family of perfect hash functions. . .

Color Coding

Deterministic algorithm:

- Color the graph using each $f \in \mathcal{F}$.
- For each coloring, check for a colorful cycle of length k.

This algorithm works if we can construct a k-perfect family of hash functions.

This algorithm is fast if the family is small, can be expressed in little space, and its functions can be evaluated quickly.

Color Coding

Fortunately, there are k-perfect families of hash functions consisting of no more than $O(1)^k \log n$ functions.

They can be stored compactly.

They can be evaluated quickly: Each $f(i)$ can be computed fast.

That is, there is a deterministic FPT algorithm for finding cycles of length k.

Integer Linear Programming

Input: An integer linear program with k variables.

Parameter: k

Question: Does this ILP have a solution?

This Problem is fixed parameter tractable.

The running time is $f(k)n^{O(1)}$, but the $f(k)$ are painfully large.

Proof: very involved...

Feedback Vertex Set

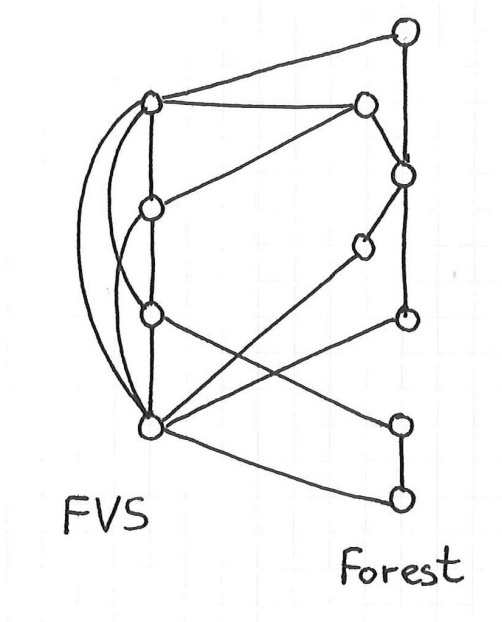
Input: A graph G and a number k

Parameter: k

Question: Are there $\leq k$ nodes whose removal makes G acyclic?

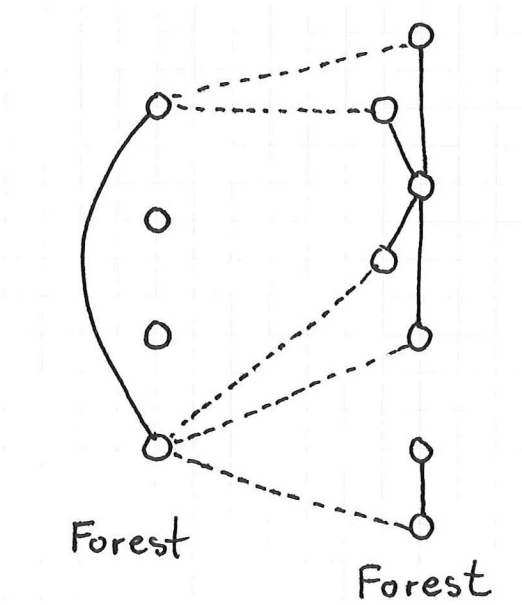
Is FVS fixed parameter tractable?

Iterative Compression



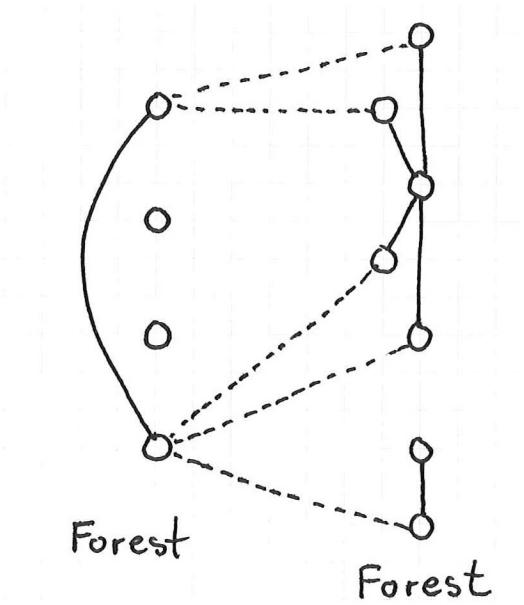
Assume we already know a FVS of size k.

Does this help to find a FVS of size $k-1$?



Step 1: Find a subset of the FVS to keep

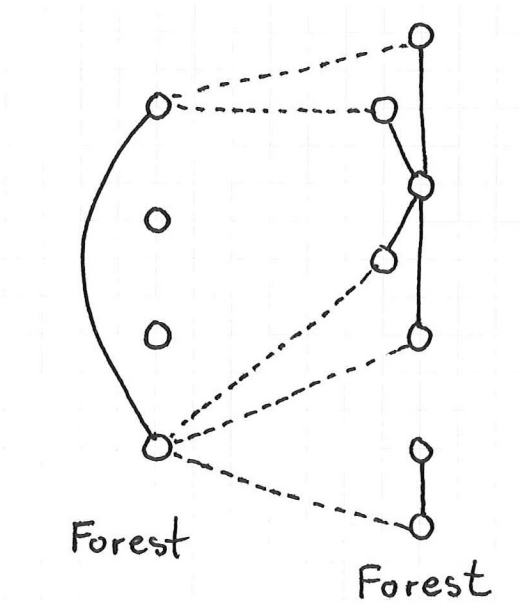
Plan: Add vertices from the forest to this FVS



Step 2: Apply reduction rules

Contract components of the FVS into one vertex

Contract degree-2 vertices in the forest



Step 3: Branching algorithm

If a leaf in the forest has two neighbors in the FVS:

a) put it into the FVS

b) delete it and decrease k

Running time

Size of the branching tree 4^k

Total size of all branching trees:

$$\sum_{j=0}^{k} \binom{k}{j} 4^k = 5^k$$

Total running time $5^k n^{O(1)}$

Overview

Depth-First Search Trees

Input: A graph G and a number k

Parameter: k

Question: Is there a path of length k in G?

Construct a depth-first search tree.

What is the helpful property of a DFS tree?

Depth-First Search Trees

Input: A graph G and a number k

Parameter: k

Question: Is there a path of length k in G?

Construct a depth-first search tree.

What is the helpful property of a DFS tree?

A Simple Theorem

Theorem

Let G be a graph and k a number.

Then it takes only polynomial time to find one of these:

1. *A cycle of length at least k*
2. *A tree decomposition of treewidth at most k*

Proof

$k + 1$ cops slowly traverse the DFS tree

A Simple Theorem

Theorem

Let G be a graph and k a number.

Then it takes only polynomial time to find one of these:

1. *A cycle of length at least k*
2. *A tree decomposition of treewidth at most k*

Proof

$k + 1$ cops slowly traverse the DFS tree

Long Paths

The theorem allows us to find paths of length k easily:

1. If we find a cycle longer than k, there obviously is a path of length k as well
2. Otherwise we use the tree decomposition and Courcelle's theorem:
   $\exists x_1 \ldots \exists x_{k+1}(inc(x_1, x_2) \wedge \cdots \wedge inc(x_k, x_{k+1}) \wedge x_1 \neq x_2 \ldots)$

Long Paths

The theorem allows us to find paths of length k easily:

1. If we find a cycle longer than k, there obviously is a path of length k as well
2. Otherwise we use the tree decomposition and Courcelle's theorem:
   $\exists x_1 \ldots \exists x_{k+1}(inc(x_1, x_2) \wedge \cdots \wedge inc(x_k, x_{k+1}) \wedge x_1 \neq x_2 \ldots)$

Long Paths

The theorem allows us to find paths of length k easily:

1. If we find a cycle longer than k, there obviously is a path of length k as well
2. Otherwise we use the tree decomposition and Courcelle's theorem:
   $\exists x_1 \ldots \exists x_{k+1}(inc(x_1, x_2) \wedge \cdots \wedge inc(x_k, x_{k+1}) \wedge x_1 \neq x_2 \ldots)$

Question: Can we solve Vertex Cover this way?

A Complicated Theorem

Theorem (Bodlaender)

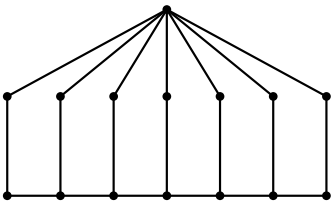
Let G be a graph and k, l some numbers.

It takes $f(k,l)|G|$ steps to find one of these:

1. *A subdivision of the $2 \times k$ grid*
2. *A subdivision of the l-circus graph*
3. *A tree decompositon of G of treewidth $2(k-1)^2(l-1)+1$.*

Proof

Again, using a DFS tree.



7-circus graph

A Complicated Theorem

Theorem (Bodlaender)

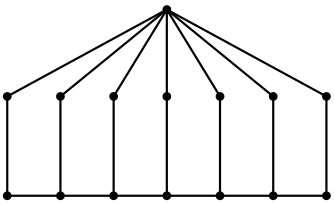
Let G be a graph and k, l some numbers.

It takes $f(k,l)|G|$ steps to find one of these:

1. *A subdivision of the $2 \times k$ grid*
2. *A subdivision of the l-circus graph*
3. *A tree decompositon of G of treewidth $2(k-1)^2(l-1)+1$.*

Proof

Again, using a DFS tree.



7-circus graph

A Complicated Theorem

Theorem (Bodlaender)

Let G be a graph and k, l some numbers.

It takes $f(k,l)|G|$ steps to find one of these:

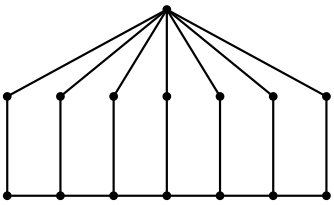
1. *A subdivision of the $2 \times k$ grid*
2. *A subdivision*
3. *A tree decom*

Question: Can we solve Dominating Set this way?

$-1)+1$.

Proof

Again, using a DFS tree.



7-circus graph

A Complicated Theorem

Theorem (Bodlaender)

Let G be a graph and k, l some numbers.

It takes $f(k,l)|G|$ steps to find one of these:

1. *A subdivision of the $2 \times k$ grid*
2. *A subdivision*
3. *A tree decom*

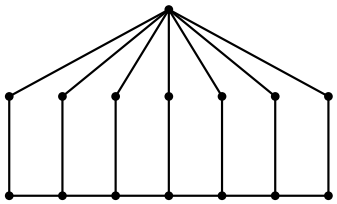
Question: Can we solve Dominating Set this way?

$-1)+1$.

Proof

Again, using

Why did it work for planar graphs?



7-circus graph

Application 1

Max-Leaf-Spanning-Tree:

Input: A graph G and a number k

Parameter: k

Question: Does G have a spanning tree with at least k leaves?

Both the $2 \times k$ grid and the k-circus graph contain a tree with k leaves.

That is, Max-Leaf-Spanning-Tree is fixed parameter tractable.

Application 1

Max-Leaf-Spanning-Tree:

Input: A graph G and a number k

Param

Questi ast k leaves?

Does the following statement hold?

If a graph contains a tree with k leaves, then it also contains a spanning tree with at least k leaves.

Both the $2 \times k$ grid and the k-circus graph contain a tree with k leaves.

That is, Max-Leaf-Spanning-Tree is fixed parameter tractable.

Application 2

Feedback Vertex Set:

Input: A graph G and a number k

Parameter: k

Question: Are there $\leq k$ nodes whose removal makes G acyclic?

Theorem

Feedback Vertex Set is fixed parameter tractable.

Application 2

Feedback Vertex Set:

Input: A graph G and a number k

Parameter: k

Question: Are there $\leq k$ nodes whose removal makes G acyclic?

Theorem

Feedback Vertex Set is fixed parameter tractable.

Feedback Vertex Set

Theorem

Feedback Vertex Set is fixed parameter tractable.

Proof

Apply Bodlaender's theorem.

1. Small tree decomposition: Courcelle
2. $2 \times 3k$ grid: No
3. $4k$-circus graph: Remove the tip and check for a FVS of size $k - 1$.

Feedback Vertex Set

Theorem

Feedback Vertex Set is fixed parameter tractable.

Proof

Apply Bodlaender's theorem.

1. Small tree decomposition: Courcelle
2. $2 \times 3k$ grid: No
3. $4k$-circus graph: Remove the tip and check for a FVS of size $k-1$.

Feedback Vertex Set

Theorem

Feedback Vertex Set is fixed parameter tractable.

Proof

Apply Bodlaender's theorem.

1. Small tree decomposition: Courcelle
2. $2 \times 3k$ grid: No
3. $4k$-circus graph: Remove the tip and check for a FVS of size $k-1$.

Feedback Vertex Set

Theorem

Feedback Vertex Set is fixed parameter tractable.

Proof

Apply Bodlaender's theorem.

1. Small tree decomposition: Courcelle
2. $2 \times 3k$ grid: No
3. $4k$-circus graph: Remove the tip and check for a FVS of size $k - 1$.

Feedback Vertex Set

Theorem

Feedback Vertex Set is fixed parameter tractable.

Proof

Apply Bodlaender's theorem.

1. Small tree decomposition: Courcelle
2. $2 \times 3k$ grid: No
3. $4k$-circus graph: Remove the tip and check for a FVS of size $k-1$.

Feedback Vertex Set

Theorem

Feedback Vertex Set is fixed parameter tractable.

Proof

Apply Bodlaender's theorem.

1. Small tree decomposition: Courcelle
2. $2 \times 3k$ grid: No
3. $4k$-circus graph: Remove the tip and check for a FVS of size $k - 1$.

Overview

Parameterized Complexity Theory

Classical complexity theory:

- Complexity classes P, NP, etc.
- Languages $L \in P$, $L \subseteq \Sigma^*$
- Framework insufficient for parameterized problems

Parameterized Complexity Theory

Definition

A parameterized problem over the alphabet Σ is a set of pairs (w, k), where $w \in \Sigma^*$ and $k \in \mathbf{N}$.

It is not allowed that there exists w and $k \neq k'$ with $(w, k) \in L$ and $(w, k') \in L$, if L is a parameterized problem.

The second condition states that k is a function of w.

Parameterized Complexity Theoery

We like to state parameterized problems as follows:

Input: A graph G and a number k

Parameter: k

Question: Does G contain a clique of size k as a subgraph?

Parameterized Complexity Theory

The parameter can be some arbitraty number, if it can be easily computed from the input.

Input: A graph G and a number k

Parameter: The diameter of G

Question: Does G contain a clique of size k as a subgraph?

Here it is easy to compute $(G, \Delta(G))$ from G in order to get formally a parameterized problem.

Parameterized Complexity Theory

One goal of complexity theory is to categorize problems into easy and hard ones.

For this purpose P and NP are best known.

Others are:

- NC and L
- AC^0 and NC^1
- $EXPTIME$ and $EXPSPACE$
- etc. etc.

Parameterized Complexity Theory

In parameterized complexity theory the easy problems can be found in the class FPT.

Definition

The class FPT contains all parameterized problems that are fixed parameter tractable.

Formally: $L \in FPT$, if there is an algorithm solving $(w, k) \in L$ in at most $f(k)|w|^c$ steps, where c is a constant and $f : \mathbf{N} \to \mathbf{N}$ an arbitrary function.

Parameterized Complexity Theory

A fundamental concept in complexity theory are reductions.

Important example: polynomial time many-one reductions:

$g\colon \Sigma^* \to \Sigma^*$ reduces the problem L_1 to L_2, if

1. $w \in L_1 \iff g(w) \in L_2$.
2. $g(w)$ can be computed in $|w|^{O(1)}$ steps.

Important property: If L_1 can be reduced to L_2 and $L_1 \notin P$, then $L_2 \notin P$.

"I can't find an efficient algorithm, but neither can all these famous people."

Important property: If L_1 can be reduced to L_2 and $L_1 \notin P$, then $L_2 \notin P$.

Parameterized Complexity Theory

Question: Is this reduction useful for parameterized problems?

1. $w \in L_1 \iff g(w) \in L_2$.
2. $g(w)$ can be computed in $|w|^{O(1)}$ steps.

Does the corresping property hold: It L_1 can be reduced to L_2 and $L_1 \notin FPT$, then $L_2 \notin FPT$.

Parameterized Complexity Theory

That corresponding property does not hold:

We can map (w, k) to $(w, |w|)$!

If we reduce a problem to itself like this, we have $f(|w|)|w|^c$ steps instead of $f(|w|)|w|^c$ steps to compute a solution.

It that way we can solve every computable problem.

A polynomial time reduction is not fine grained enough.

Parameterized Reductions

Definition

A parameterized problem $L_1 \subseteq \Sigma^*$ can be reduced to $L_2 \subseteq \Gamma^*$ by a parameterized reduction if

- $r, s \colon \mathbf{N} \to \mathbf{N}$ are computable functions,
- there is a function $g \colon \Sigma^* \times \mathbf{N} \to \Gamma^*$, $(w, k) \mapsto (w', k')$, that can be computed in $r(k)|w|^{O(1)}$ steps and $k' = s(k)$,
- $(w, k) \in L_1$ if and only if $g(w, k) \in L_2$.

Parameterized Reductions

Theorem

If $L_1 \notin FPT$ and there is a parameterized reduction from L_1 to L_2, then $L_2 \notin FPT$.

Proof

Assume $L_2 \in FPT$. We can computed $(w', k') = g(w, k)$ in $r(k)|w|^c$ steps such that $k' = s(k)$ and $|w'| \leq r(k)|w|^c$.

Then test whether $(w', k') \in L_2$ taking $f'(k')|w'|^d \leq f'(s(k))r(k)^d|w|^{cd}$ steps.

Because $(w, k) \in L_1 \iff (w', k') \in L_2$, we answered whether $(w, k) \in L_1$ holds and therefore $L_1 \in FPT$.

Parameterized Reductions

Look at some classical reductions:

- Vertex Cover ot Independent Set
- CNF-SAT to 3SAT (weighted)
- Clique to Independent Set

Classical reductions are usually not parameterized reductions.

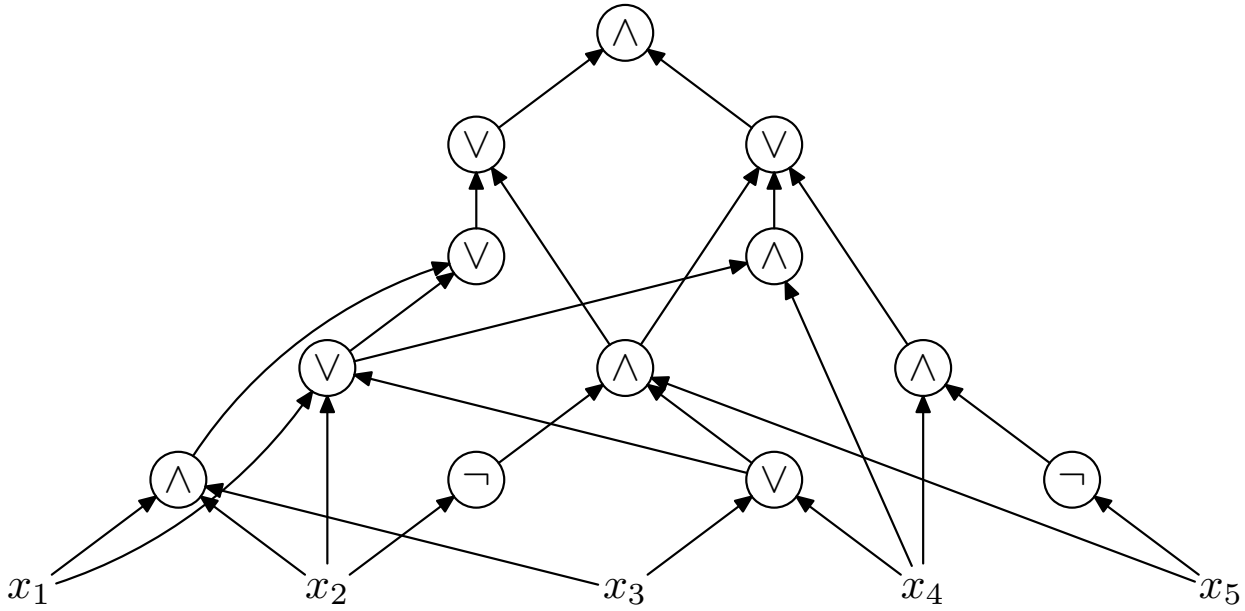
Parameterized Complexity Theory

Definition

A boolean circuit is an acyclic graph such that:

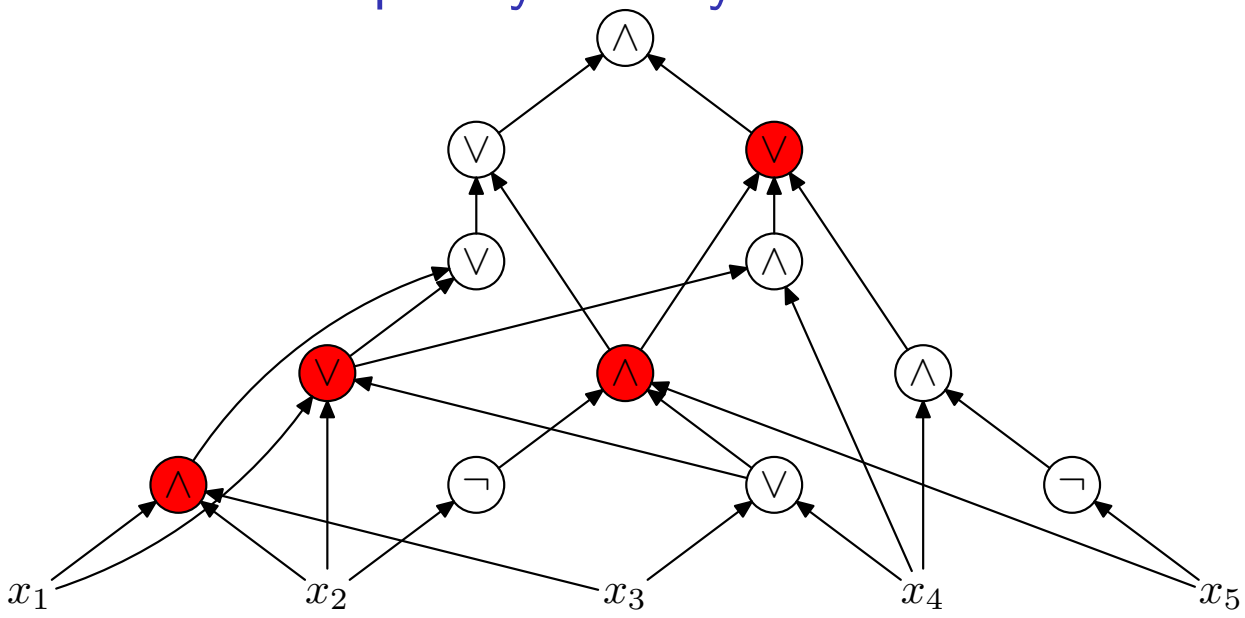
- There is exactly one vertex with outdegree 0, the Output.
- Every vertex with indegree 0 is an Input and labeled by x_i or $\neg x_i$.
- All other vertices are gates and labeled by $\wedge$, $\vee$ or $\neg$ (with indegree 1).

Parameterized Complexity Theory



To find out whether a boolean circuit has a satisfying assignment is *NP*-complete.

Parameterized Complexity Theory



Big gates have indegree > 2.

The height is the length of the longest path.

The weft is the maximal number of big gates on some path.

Parameterized Complexity Theory

Definition

Let $\mathcal{F}(t, h)$ be the set of all boolean circuits with height h and weft t.

Definition

The weighted satisfiability problem $L_{\mathcal{F}(t,h)}$:

Input: (G, k), where $G \in \mathcal{F}(t, h)$

Parameter: k

Question: Has G a satisfiying assingment of weight k

The weight of an assignment is the number of 1s.

Parameterized Complexity Theory

Definition

A parameterized problem is in the complexity class $W[t]$ if it can be reduced to $L_{\mathcal{F}(t,h)}$ for some h by a parameterized reduction.

Example

Independent Set is in $W[1]$.

Dominating Set is in $W[2]$.

Question: Why?

Parameterized Complexity Theory

Definition

A problem L is $W[t]$-hard, if every problem in $W[t]$ can be reduced to L by a parameterized reduction.

Definition

A problem is $W[t]$-complete, if it belongs to $W[t]$ and is $W[t]$-hard.

Parameterized Complexity Theory

Theorem

Let A be a $W[t]$-complete problem.

Assume that A can be reduced to B by a parameterized reduction and $B \in W[t]$.

Then B is also $W[t]$-complete.

Proof

We already assumed that $B \in W[t]$.

Since every problem in $W[t]$ can be reduced to A, the $W[t]$-hardness follows from the transitivity of parameterized reducibility.

Short Turing Machine Acceptance

We will see later that the following problem is $W[1]$-complete:

Definition

Short Turing Machine Acceptance:

Input: A non-deterministic Turing maching M, a word w, a number k.

Parameter: k

Question: Does M have an accepting path of length at most k on input w?

The class $W[1, s]$ and $W[1, 2]$

We consider the weighted satisfiability problem for very simple circuits.

Definition

Let $s > 1$ be a number. By $\mathcal{F}(s)$ we denote the family of all circuit whose output is an AND-gate that is connected to OR-gates with indegrees at most s. The OR-gates are directly connected to inputs (literals, i.e., variables or negated variables).

We define $W[1, s]$ as the class of all problems in $W[1]$ that can be reduced to $L_{\mathcal{F}(s)}$ by a parameterized reduction.

We will prove the following theorem:

Theorem

Short Turing Machine Acceptance $\in W[1,2]$

For this end we need a reduction from Short Turing Machine Acceptance to $L_{\mathcal{F}(2)}$.

We have to map M, w, k to a circuit and a number $k' = f(k)$ in such a way that there is a satisfying assignment of weight k' iff M accepts w in at most k steps.

We will prove the following theorem:

Theorem

Short Turing Machine Acceptance $\in W[1, 2]$

For this end we need a reduction from Short Turing Machine Acceptance to $L_{\mathcal{F}(2)}$.

We have to map M, w, k to a circuit and a number $k' = f(k)$ in such a way that there is a satisfying assignment of weight k' iff M accepts w in at most k steps.

We can enumerate all configurations of M and therefore identify them with the numbers $1, 2, \ldots$

A configuration consists of

- the position of the read/write-head,
- the state.

If i is a configuration and a is a symbol, then let $\delta(i, a) = (j, b)$ hold if M changes from configuration i into j when reading the symbol a overwriting it with b.

We model with the variable $C_{t,i,j,a,b}$ that M changes from configuration i to j in the tth step while reading an a and overwriting it with a b.

We can enumerate all configurations of M and therefore identify them with the numbers $1, 2, \ldots$

A configuration consists of

- the position of the read/write-head,
- the state.

If i is a configuration and a is a symbol, then let $\delta(i, a) = (j, b)$ hold if M changes from configuration i into j when reading the symbol a overwriting it with b.

We model with the variable $C_{t,i,j,a,b}$ that M changes from configuration i to j in the tth step while reading an a and overwriting it with a b.

We can enumerate all configurations of M and therefore identify them with the numbers $1, 2, \ldots$

A configuration consists of

- the position of the read/write-head,
- the state.

If i is a configuration and a is a symbol, then let $\delta(i, a) = (j, b)$ hold if M changes from configuration i into j when reading the symbol a overwriting it with b.

We model with the variable $C_{t,i,j,a,b}$ that M changes from configuration i to j in the tth step while reading an a and overwriting it with a b.

We can enumerate all configurations of M and therefore identify them with the numbers $1, 2, \ldots$

A configuration consists of

- the position of the read/write-head,
- the state.

If i is a configuration and a is a symbol, then let $\delta(i, a) = (j, b)$ hold if M changes from configuration i into j when reading the symbol a overwriting it with b.

We model with the variable $C_{t,i,j,a,b}$ that M changes from configuration i to j in the tth step while reading an a and overwriting it with a b.

The variable $M_{t,p,a,b}$ models that at the beginning of the tth step the symbol a can be found at the pth position of the tape and that it is overwritten with a b during this step.

Our next goal is to construct a formula whose satisfying assignments model a computation of the Turing maching M.

In particular there will be a satisfying assignment with $C_{t,i,j,a,b} = 1$ iff there is a computation where M goes from configuration i to j in its tth step reading an a and overwriting it with a b.

Every possible computation path should correspond to exactly one (weighted) satisfying assignment.

Moreover: There should exist only accepting computation of length k and satisfying assignments with weight $f(k)$.

The variable $M_{t,p,a,b}$ models that at the beginning of the tth step the symbol a can be found at the pth position of the tape and that it is overwritten with a b during this step.

Our next goal is to construct a formula whose satisfying assignments model a computation of the Turing maching M.

In particular there will be a satisfying assignment with $C_{t,i,j,a,b} = 1$ iff there is a computation where M goes from configuration i to j in its tth step reading an a and overwriting it with a b.

Every possible computation path should correspond to exactly one (weighted) satisfying assignment.

Moreover: There should exist only accepting computation of length k and satisfying assignments with weight $f(k)$.

We need a lot of constraints in order to make this model work.

We will express each constraint by an AND of ORs.

We define clauses in such a way that a wrong modelling automatically leads to a non-satisfying assignment.

In that way, satisfying assignments are those that do not overstep any rule.

Rule 1:

"Nothing exists twice because we have a computation path"

More precisely: In step t the machine M is in exactly one state, changes to exactly one other, reads exactly one symbol, and overwrites it by exactly one symbol.

How can we enforce Rule 1 by clauses?

$$C_{t,i,j,a,b} \rightarrow \overline{C_{t,i',j',a',b'}}$$

or, equivalently,

$$\overline{C_{t,i,j,a,b}} \vee \overline{C_{t,i',j',a',b'}}$$

for all $t, i, j, a, b, i', j', a', b'$ with $(i, j, a, b) \neq (i', j', a', b')$ and

$$M_{t,p,a,b} \rightarrow \overline{M_{t,p,a',b'}}$$

for all t, p, a, b, a', b' with $(a, b) \neq (a', b')$.

Rule 1:

"Nothing exists twice because we have a computation path"

More precisely: In step t the machine M is in exactly one state, changes to exactly one other, reads exactly one symbol, and overwrites it by exactly one symbol.

How can we enforce Rule 1 by clauses?

$$C_{t,i,j,a,b} \to \overline{C_{t,i',j',a',b'}}$$

or, equivalently,

$$\overline{C_{t,i,j,a,b}} \vee \overline{C_{t,i',j',a',b'}}$$

for all $t, i, j, a, b, i', j', a', b'$ with $(i, j, a, b) \neq (i', j', a', b')$ and

$$M_{t,p,a,b} \to \overline{M_{t,p,a',b'}}$$

for all t, p, a, b, a', b' with $(a, b) \neq (a', b')$.

Rule 1:

"Nothing exists twice because we have a computation path"

More precisely: In step t the machine M is in exactly one state, changes to exactly one other, reads exactly one symbol, and overwrites it by exactly one symbol.

How can we enforce Rule 1 by clauses?

Question:

Why only one possibility? We have nondeterministic TMs after all?

or, equivalen

for all $t, i, j, a, b, i', j', a', b'$ with $(i, j, a, b) \neq (i', j', a', b')$ and

$$M_{t,p,a,b} \rightarrow \overline{M_{t,p,a',b'}}$$

for all t, p, a, b, a', b' with $(a, b) \neq (a', b')$.

Rule 2:

"M and C must fit together."

More precisely: If we read a symbol, it has to be there beforehand. If we write a symbol, it must be there afterwards.

How can we enforce Rule 2 by clauses?

$$C_{t,i,j,a,b} \rightarrow M_{t,p(i),a,b}$$

for all t, i, j, a, b, where $p(i)$ is the position of the read/write head in configuration i.

Rule 2:

"M and C must fit together."

More precisely: If we read a symbol, it has to be there beforehand. If we write a symbol, it must be there afterwards.

How can we enforce Rule 2 by clauses?

$$C_{t,i,j,a,b} \rightarrow M_{t,p(i),a,b}$$

for all t, i, j, a, b, where $p(i)$ is the position of the read/write head in configuration i.

Rule 2:

"M and C must fit together."

More precise beforehand.
If we write a

How can we

Question:

Do we need the other direction, too?

"If there is a symbol on the tape, then this symbol is read."

for all t, i, j, a, b, where $p(i)$ is the position of the read/write head in configuration i.

Rule 3:

"Subsequent steps have to fit together."

More precisely: If one step ends with a configuration, the next step has to start with the same one. The tape content cannot change from step t to step $t+1$ at most places.

How can we enforce Rule 3 by clauses?

$$C_{t,i,j,a,b} \rightarrow \overline{C_{t+1,i',j',c,d}}$$

for all $t, i, j, i', j' a, b, c, d$ with $i' \neq j$ and

$$M_{t,p,a,b} \rightarrow \overline{M_{t+1,p,c,d}}$$

for all t, p, a, b, c, d with $b \neq c$.

Rule 3:

"Subsequent steps have to fit together."

More precisely: If one step ends with a configuration, the next step has to start with the same one. The tape content cannot change from step t to step $t+1$ at most places.

How can we enforce Rule 3 by clauses?

$$C_{t,i,j,a,b} \rightarrow \overline{C_{t+1,i',j',c,d}}$$

for all $t, i, j, i', j' a, b, c, d$ with $i' \neq j$ and

$$M_{t,p,a,b} \rightarrow \overline{M_{t+1,p,c,d}}$$

for all t, p, a, b, c, d with $b \neq c$.

Rule 4:

"The beginning and the end have to be correct. The computation path must be accepting."

$\rightarrow$ Exercise

Because all rules have to hold simultaneously we can combine them with a big AND.

This yields an $\mathcal{F}(2)$-formula as desired.

There is an accepting path of length k iff there is a satisfying assignment with weight k'.

Rule 4:

"The beginning and the end have to be correct. The computation path must be accepting."

$\rightarrow$ Exercise

Because all rules have to hold simultaneously we can combine them with a big AND.

This yields an $\mathcal{F}(2)$-formula as desired.

There is an accepting path of length k iff there is a satisfying assignment with weight k'.

Rule 4:

"The beginning and the end have to be correct. The computation path must be accepting."

→ Exercise

Because all rules have to hold simultaneously we can combine them v

Question:

How big is k'?

This y d.

There is an accepting path of length k iff there is a satisfying assignment with weight k'.

Remember:

We just proved the following.

Theorem

Short Turing Machine Acceptance $\in W[1, 2]$

The class antimonotone-$W[1, s]$

We consider the weighted satisfiability problem for very simple structured circuits.

Definition

Let $s > 1$ be a number. By antimonotone-$\mathcal{F}(s)$ we denote the family of all circuits whose output is an AND-gate connected to OR-gates with indegree at most s. The OR-gates are connected to negative literals only (negated variables).

Antimonotone-$W[1, s]$ is the class of all parameterized problems in $W[1]$ that can be reduced to $L_{\text{Antimonoton-}\mathcal{F}(s)}$ by a parameterized reduction.

Theorem

$L_{\text{Antimonoton-}\mathcal{F}(s)}$ can be reduced to Short Turing Machine Acceptance be a parameterized reduction.

Corollary

$\text{antimonotone-}W[1,s] \subseteq \text{antimonotone-}W[1,2]$

Proof

Construct a turing machine that works as follows:

1. Guess k variables onto the tape.
2. Visit all subsets of size s of them.
3. Verify for each subset that it does not cover a clause.

Our real goal is:

$W[1] \subseteq \text{antimonotone-}W[1,2]$

Theorem

$L_{\text{Antimonoton-}\mathcal{F}(s)}$ can be reduced to Short Turing Machine Acceptance be a parameterized reduction.

Corollary

antimonotone-$W[1, s] \subseteq$ antimonotone-$W[1, 2]$

Proof

Construct a turing machine that works as follows:

1. Guess k variables onto the tape.
2. Visit all subsets of size s of them.
3. Verify for each subset that it does not cover a clause.

Our real goal is:

$W[1] \subseteq$ antimonotone-$W[1, 2]$

Theorem

$L_{\text{Antimonoton-}\mathcal{F}(s)}$ can be reduced to Short Turing Machine Acceptance be a parameterized reduction.

Corollary

antimonotone-$W[1, s] \subseteq$ *antimonotone-*$W[1, 2]$

Proof

Construct a turing machine that works as follows:

1. Guess k variables onto the tape.
2. Visit all subsets of size s of them.
3. Verify for each subset that it does not cover a clause.

Our real goal is:

$W[1] \subseteq$ antimonotone-$W[1, 2]$

Theorem

$L_{\text{Antimonoton-}\mathcal{F}(s)}$ can be reduced to Short Turing Machine Acceptance be a parameterized reduction.

Corollary

antimonotone-$W[1,s] \subseteq$ antimonotone-$W[1,2]$

Proof

Construct a ollows:

Question:

What is the running time?

1. Guess k variables onto the tape.
2. Visit all subsets of size s of them.
3. Verify for each subset that it does not cover a clause.

Our real goal is:

$W[1] \subseteq$ antimonotone-$W[1,2]$

Theorem

$L_{\text{Antimonoton-}\mathcal{F}(s)}$ can be reduced to Short Turing Machine Acceptance be a parameterized reduction.

Corollary

antimonotone-$W[1, s] \subseteq$ *antimonotone-*$W[1, 2]$

Proof

Construct a turing machine that works as follows:

1. Guess k variables onto the tape.
2. Visit all subsets of size s of them.
3. Verify for each subset that it does not cover a clause.

Our real goal is:

$W[1] \subseteq$ antimonotone-$W[1, 2]$

The class $W[1, 1, s]$

Now we consider the weighted satisfiability problem for different, but still very simple circuits.

Definition

Let $s > 1$ be a number. By $\mathcal{F}(1, 1, s)$ we denote the family of all circuits whose output is an OR-gate connected to AND-gates that are connected to OR-gates with indegree at most s.

By $W[1, 1, s]$ we denote the class of problems in $W[1]$ that can be reduced to $L_{\mathcal{F}(1,1,s)}$ by a parameterized reduction.

Simplification of Weft-1-Circuits

Theorem

Consider a circuit of weft 1 and height h.

Then we can construct an equivalent circuit $\mathcal{F}(1, 1, s)$ in polynomial time, where s depends only on h.

Proof

- DNF und CNF
- de Morgan
- Combination of gates of same type
- Distributive law

Simplification of Weft-1-Circuits

Theorem

Consider a circuit of weft 1 and height h.

Then we can construct an equivalent circuit $\mathcal{F}(1, 1, s)$ in polynomial time, where s depends only on h.

Proof

- DNF und CNF
- de Morgan
- Combination of gates of same type
- Distributive law

$L_{\mathcal{F}(1,1,s)}$

Goal: Reduce $L_{\mathcal{F}(1,1,s)}$ to STMA.

Intermediate step:

Reduce $L_{\mathcal{F}(1,1,s)}$ to one TM M_i for all subcircuits below the output gate.

One TM guesses whicht M_i will be used for the simulation.

Still not known:

How to reduce $L_{\mathcal{F}(1,s)}$ to STMA.

$L_{\mathcal{F}(1,1,s)}$

Goal: Reduce $L_{\mathcal{F}(1,1,s)}$ to STMA.

Intermediate step:

Reduce $L_{\mathcal{F}(1,1,s)}$ to one TM M_i for all subcircuits below the output gate.

One TM guesses whicht M_i will be used for the simulation.

Still not known:

How to reduce $L_{\mathcal{F}(1,s)}$ to STMA.

Reduction of $L_{\mathcal{F}(1,s)}$ to STMA:

Construct a TM that guesses an assignment on the tape and then computes two numbers:

- $A =$ the number of clauses that are satisfied by negated variables.
- $M =$ the number of clauses that are satisfied only by positive literals.

Assignment is satisfying iff $A + M =$ number of clauses.

Reduction of $L_{\mathcal{F}(1,s)}$ to STMA:

Construct a TM that guesses an assignment on the tape and then computes two numbers:

- $A =$ the number of clauses that are satisfied by negated variables.
- $M =$ the number of clauses that are satisfied only by positive literals.

Assignment is satisfying iff $A + M =$ number of clauses.

Reduction of $L_{\mathcal{F}(1,s)}$ to STMA:

Construct a TM that guesses an assignment on the tape and then computes two numbers:

- $A =$ the number of clauses that are satisfied by negated variables.
- $M =$ the number of clauses that are satisfied only by positive literals.

Assignment is satisfying iff $A + M =$ number of clauses.

Reduction of $L_{\mathcal{F}(1,s)}$ to STMA:

Construct a TM that guesses an assignment on the tape and then computes two numbers:

- $A =$ the number of clauses that are satisfied by negated variables.
- $M =$ the number of clauses that are satisfied only by positive literals.

Assignment is satisfying iff $A + M =$ number of clauses.

A can be computed as in the antimonotone case.

How to compute M?

Let $M(S, T)$ be the number of clauses that have exactly all variables in T as their negative literals and have at least the variables in S as positive literals.

Then

$$M = \sum_{S,T \subseteq P, |S|, |T| \leq s} (-1)^{|S|+1} M(S, T),$$

if P is the set of variables on the tape.

A can be computed as in the antimonotone case.

How to compute M?

Let $M(S, T)$ be the number of clauses that have exactly all variables in T as their negative literals and have at least the variables in S as positive literals.

Then

$$M = \sum_{S,T \subseteq P, |S|, |T| \leq s} (-1)^{|S|+1} M(S, T),$$

if P is the set of variables on the tape.

A can be computed as in the antimonotone case.

How to compute M?

Let $M(S, T)$ be the number of clauses that have exactly all variables in T as their negative literals and have at least the variables in S as positive literals.

Then

$$M = \sum_{S,T \subseteq P, |S|, |T| \leq s} (-1)^{|S|+1} M(S, T),$$

if P is the set of variables on the tape.

Definition

Multicolored Clique is the following problem:

Input: A Graph G, nodes colored by k colors

Parameter: k

Question: Is there a k-clique with k colors?

Theorem

Multicolored Clique is $W[1]$-hard.

Definition

List Coloring (parameterized by treewidth) is the following problem:

Input: A Graph G, nodes have lists of colors

Parameter: treewidth of G

Question: Is there a node coloring with colors from the lists?

Theorem

tw–List Coloring is $W[1]$-hard.

Definition

Multicolored Grid is the following problem:

Input: A Graph G, nodes colored by $\{(i,j) \mid 1 \leq i,j \leq k\}$.

Parameter: k

Question: Is there a $k \times k$-grid whose node at coordinates (i,j) is colored with (i,j)?

Theorem

Multicolored Grid is $W[1]$-hard.

Theorem

List coloring is $W[1]$-hard with parameter treewidth even for planar graphs.

Definition

An OR-distillation algorithm for a problem L is an algorithm that transforms $(v_1, \ldots, v_t)$ into w with these properties:

1. runs in polynomial time
2. $w \in L$ iff some $v_i \in L$
3. $|w|$ polynomially bounded in $|v_i|$ for all i

For which problems L do distillation algorithms exist?

Theorem

If an OR-distillation algorithm for an NP-complete problem exists, then coNP $\subseteq$ NP/poly.

Definition

An OR-distillation algorithm for a problem L is an algorithm that transforms $(v_1, \ldots, v_t)$ into w with these properties:

1. runs in polynomial time
2. $w \in L$ iff some $v_i \in L$
3. $|w|$ polynomially bounded in $|v_i|$ for all i

For which problems L do distillation algorithms exist?

Theorem

If an OR-distillation algorithm for an NP-complete problem exists, then coNP $\subseteq$ NP/poly.

Definition

An OR-composition algorithm for a parameterized problem L is an algorithm that transforms $((v_1, k), \ldots, (v_t, k))$ into (w, k') with these properties:

1. runs in polynomial time
2. $(w, k') \in L$ iff some $(v_i, k) \in L$
3. k' polynomially bounded in k

Theorem

Let L be an NP-complete parameterized problem (where the parameter is encoded in unary as part of the input). If there is an OR-composition algorithm for L and L has a polynomial kernel, then there is also an OR-distillation algorithm for L.

Definition

An OR-composition algorithm for a parameterized problem L is an algorithm that transforms $((v_1, k), \ldots, (v_t, k))$ into (w, k') with these properties:

1. runs in polynomial time
2. $(w, k') \in L$ iff some $(v_i, k) \in L$
3. k' polynomially bounded in k

Theorem

Let L be an NP-complete parameterized problem (where the parameter is encoded in unary as part of the input). If there is an OR-composition algorithm for L and L has a polynomial kernel, then there is also an OR-distillation algorithm for L.

Theorem

If a parameterized problem has an OR-composition algorithm and a polynomial kernel, then coNP $\subseteq$ NP/poly.

The following problems have OR-composition algorithms:

- k-path
- k-cycle

Theorem

If a parameterized problem has an OR-composition algorithm and a polynomial kernel, then coNP $\subseteq$ NP/poly.

The following problems have OR-composition algorithms:

- k-path
- k-cycle

A similar framework exists for AND-composition and AND-distillation.

Theorem

If a parameterized problem has an AND-composition algorithm and a polynomial kernel, then coNP $\subseteq$ NP/poly.

The following problems have AND-composition algorithms:

- treewidth
- pathwidth

A similar framework exists for AND-composition and AND-distillation.

Theorem

If a parameterized problem has an AND-composition algorithm and a polynomial kernel, then coNP $\subseteq$ NP/poly.

The following problems have AND-composition algorithms:

- treewidth
- pathwidth

Definition

The k-leaf outbranching problem:

- Input: A directed graph G and a number k
- Parameter: k
- Question: Does G have a k-leaf outbranching.

An outbranching is a directed out-tree.

This problem has an OR-composition algorithm, hence no polynomial kernel.

Definition

The k-leaf outbranching problem:

- Input: A directed graph G and a number k
- Parameter: k
- Question: Does G have a k-leaf outbranching.

An outbranching is a directed out-tree.

This problem has an OR-composition algorithm, hence no polynomial kernel.

Definition

The rooted k-leaf outbranching problem:

- Input: A directed graph G, a node r, and a number k
- Parameter: k
- Question: Does G have a k-leaf outbranching with root r.

No easy to find OR-composition algorithm.

Indeed, there is a k^3 kernel for this problem.

(Proof: Very technical with five reduction rules.)

Definition

The rooted k-leaf outbranching problem:

- Input: A directed graph G, a node r, and a number k
- Parameter: k
- Question: Does G have a k-leaf outbranching with root r.

No easy to find OR-composition algorithm.

Indeed, there is a k^3 kernel for this problem.

(Proof: Very technical with five reduction rules.)

Definition

The rooted k-leaf outbranching problem:

- Input: A directed graph G, a node r, and a number k
- Parameter: k
- Question: Does G have a k-leaf outbranching with root r.

No easy to find OR-composition algorithm.

Indeed, there is a k^3 kernel for this problem.

(Proof: Very technical with five reduction rules.)

Can we "reduce" the k-leaf outbranching problem to the rooted k-leaf outbranching problem?

Yes and No. Depends on what "reduce" exactly means.

We can take a k-leaf outbranching problem and reduce it to n instances of rooted k-leaf outbranching.

This is similar to a kernel and is called a "Turing kernel."

Can we "reduce" the k-leaf outbranching problem to the rooted k-leaf outbranching problem?

Yes and No. Depends on what "reduce" exactly means.

We can take a k-leaf outbranching problem and reduce it to n instances of rooted k-leaf outbranching.

This is similar to a kernel and is called a "Turing kernel."

Can we "reduce" the k-leaf outbranching problem to the rooted k-leaf outbranching problem?

Yes and No. Depends on what "reduce" exactly means.

We can take a k-leaf outbranching problem and reduce it to n instances of rooted k-leaf outbranching.

This is similar to a kernel and is called a "Turing kernel."

The Exponential Time Hypothesis

We will use this simple form of the Exponential Time Hypothesis (ETH):

There is a constant $\alpha > 0$ such that no algorithm can solve 3-SAT in at most $2^{\alpha n}(n+m)^{O(1)}$ time.

In particular this implies:

There is no algorithm that solves 3-SAT in $2^{o(n)}(n+m)^{O(1)}$.

The ETH is a complexity theoretic assumption (like $P \neq NP$).

$P \neq NP$ follows from ETH, but not necessarily the other way around.

The Exponential Time Hypothesis

We will use this simple form of the Exponential Time Hypothesis (ETH):

There is a constant $\alpha > 0$ such that no algorithm can solve 3-SAT in at most $2^{\alpha n}(n+m)^{O(1)}$ time.

In particular this implies:

There is no algorithm that solves 3-SAT in $2^{o(n)}(n+m)^{O(1)}$.

The ETH is a complexity theoretic assumption (like $P \neq NP$).

$P \neq NP$ follows from ETH, but not necessarily the other way around.

The Exponential Time Hypothesis

We will use this simple form of the Exponential Time Hypothesis (ETH):

There is a constant $\alpha > 0$ such that no algorithm can solve 3-SAT in at most $2^{\alpha n}(n+m)^{O(1)}$ time.

In particular this implies:

There is no algorithm that solves 3-SAT in $2^{o(n)}(n+m)^{O(1)}$.

The ETH is a complexity theoretic assumption (like $P \neq NP$).

$P \neq NP$ follows from ETH, but not necessarily the other way around.

Subexponential time lower bounds

Assume that we can solve Indepented Set in $2^{o(n)}$ steps (where n is the number of vertices).

How fast can we then solve 3-SAT by reducing it to an IS instance and solve this instance with the above subexponential time solver?

Let ϕ be a 3-SAT instance with n variables m clauses.

We can reduce it to an IS instance with $3m$ vertices.

This can be solved in $2^{o(3m)} = 2^{o(m)}$ time.

It does not contradict the ETH.

Subexponential time lower bounds

Assume that we can solve Indepented Set in $2^{o(n)}$ steps (where n is the number of vertices).

How fast can we then solve 3-SAT by reducing it to an IS instance and solve this instance with the above subexponential time solver?

Let ϕ be a 3-SAT instance with n variables m clauses.

We can reduce it to an IS instance with $3m$ vertices.

This can be solved in $2^{o(3m)} = 2^{o(m)}$ time.

It does not contradict the ETH.

Subexponential time lower bounds

Assume that we can solve Indepented Set in $2^{o(n)}$ steps (where n is the number of vertices).

How fast can we then solve 3-SAT by reducing it to an IS instance and solve this instance with the above subexponential time solver?

Let ϕ be a 3-SAT instance with n variables m clauses.

We can reduce it to an IS instance with $3m$ vertices.

This can be solved in $2^{o(3m)} = 2^{o(m)}$ time.

It does not contradict the ETH.

The Sparsification Lemma

Theorem

For $\epsilon > 0$ and an integer $r > 0$ there is a constant c such that:

1. *For every r-CNF formula ϕ with n variables there is a disjunction ψ of at most $2^{\epsilon n}$ many r-CNF formulas in which every variable occurs in at most c clauses.*
2. *ϕ is satisfiable iff ψ is satisfiable.*
3. *ψ can be computed in $2^{\epsilon n} n^{O(1)}$ time.*

Subexponential time lower bounds

Assume that we can solve Indepented Set in $2^{o(n)}$ steps (where n is the number of vertices).

How fast can we then solve 3-SAT by reducing it to an IS instance and solve this instance with the above subexponential time solver?

Let ϕ be a 3-SAT instance with n variables m clauses.

Use the sparsification lemma to get formulas ψ_i.

The length of each ψ_i is only $O(n)$.

Turn each ψ_i into an IS instance with $O(n)$ vertices.

Solve them in $2^{o(n)}$ steps.

Lower bounds on parameterized problems

If we can reduce 3-SAT to a parameterized problem L such that the parameter is bounded by $O(n+m)$, then L cannot be solved in time $2^{o(k)}poly(n)$ (under ETH).

Proof: Assume otherwise. Then we can solve 3-SAT in time $2^{o(n+m)}$, which contradicts ETH:

First transform a 3-SAT instance I into an L-instance (x,k). Then $|x| = poly(|I|)$ and $k = O(n+m)$. This takes polynomial time.

Then solve $(x,k) \in L$ in time

$$2^{o(k)}poly(|x|) = 2^{o(n+m)}poly(n+m) = 2^{o(n+m)}.$$

Corollary: Vertex Cover and Feedback Vertex Set (and many other problems) cannot be solved in time $2^{o(k)}poly(n)$ under ETH.

Planar 3-SAT

The incidence graph of a 3-SAT formula has a node for each clause and a node for each variable. There is an edge if the variable occurs in a clause.

Planar 3-SAT consists of all satisfiable 3-SAT instances whose incidence graph is planar.

We can reduce in polynomial time a 3-SAT with n variables and m clauses to a Planar 3-SAT instance with $O((n+m)^2)$ clauses and variables.

Proof idea: Replace each crossing by a planar crossover gadget. There are at most nm crossings.

Planar 3-SAT

We cannot solve Planar 3-SAT in time $2^{o(\sqrt{n})}$ under ETH.

Proof: Assume otherwise. Take a 3-SAT instance with n variables and $O(n)$ clauses and transform it into a Planar 3-SAT instance with $O(n^2)$ variables. Then solve it in $2^{(o\sqrt{n^2})} = 2^{o(n)}$ time.

Contradiction.

Corollary: *We cannot solve Planar Vertex Cover and Planar Dominating Set in time $2^{o(\sqrt{k})}poly(n)$ under ETH.*

An artificial problem

The problem $k \times k$-clique:

Input: Graph G with $V(G) = \{1, \ldots, k\}^2$

Parameter: k

Question: Is there a k-clique with one vertex from "each row"?

Lemma

$k \times k$-clique cannot be solved in $2^{o(k \log k)}$ under ETH.

Suppose otherwise. Then we can solve 3-coloring in $2^{o(n)}$.

Proof idea: Given a graph G with n nodes.

Let k be the smallest number with $3^{n/k+1} \leq k$.

(Then $k \log k = O(n)$ and $n/k = O(\log n)$.)

Evenly partition vertices of G into $X_1, \ldots, X_k$.

Construct graph with proper 3-colorings of X_i's as vertices and an edge between "compatible" colorings.

There is a special k-clique iff G is 3-colorable.

Lower bound can be transferred to closest string:

Under ETH closest string cannot be solved in $2^{o(m \log m)} = m^{o(m)}$.

Lower bounds for hard problems

k-clique (and k-independent set) cannot be solved in $f(k)n^{o(k)}$ steps for any computable f.

Proof idea: Assume otherwise and let, e.g., $f(k) = 2^k$.

Then choose $k = \log n$ (or, in general $k = f^{-1}(n)$.

Split a graph G with n vertices into k groups of almost same size.

Build a new graph H whose vertices are valid 3-colorings of the groups. There is an edge between two vertices if their colorings are compatible.

The size of H is at most $N = k3^{n/k}$. H has a k-clique iff G is 3-colorable.

Find such a clique in time $N^{o(k)} = (k3^{n/k})^{o(k)} = 2^{o(n)}$.

We can therefore find out whether G is 3-colorable in time $2^{o(n)}$. Contradiction.

The Strong Exponential Time Hypothesis (SETH)

Let δ_r be the infimum of all δ'_r for which an algorithm exists that solves r-SAT in time $O(2^{\delta'_r n})$.

ETH: $\delta_3 > 0$

SETH: $\lim_{r\to\infty} \delta_r = 1$

SETH implies ETH. (why?)

ETH implies $\mathrm{W}[1] \neq \mathrm{FPT}$. (why?)

The faith into SETH is smaller than into ETH.

There are fewer results for SETH than for ETH.

Lower bound for algorithms on tree decompositions

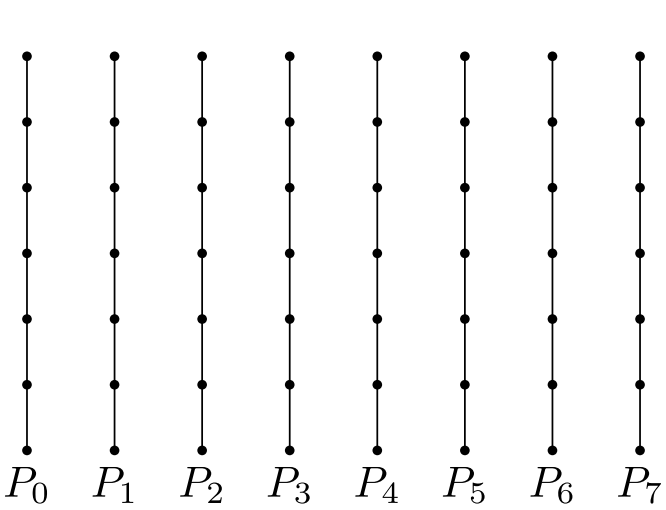
We have seen that Independent Set can be solved in time $2^k n^{O(1)}$ if the input is a tree decomposition of width k.

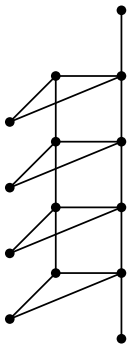
Under SETH we cannot solve this problem in time $(2-\epsilon)^k n^{O(1)}$ *for any* $\epsilon > 0$.

Proof idea: For a given CNF-SAT formula ϕ with n variables and m clauses construct a graph G with path-width $n+3$ and size $O(n^3 m)$.

G has an independent set of a given size iff ϕ is satifiable.

If we can find a maximal independent set in time $(2-\epsilon)^{n+3}|G|^{O(1)}$, then we solve CNF-SAT in time $(2-\epsilon)^n|\phi|^{O(1)}$ and SETH fails.





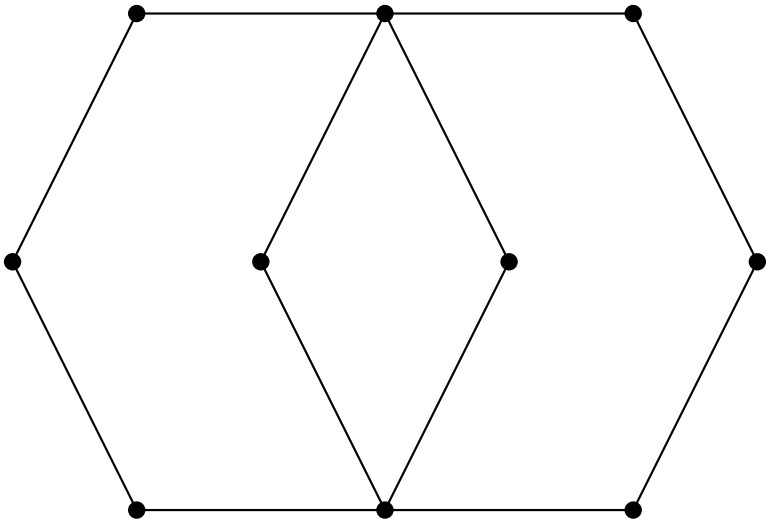
Connect gadget for every clause.

One connection has to go to an empty vertex.

Repeat $n + 1$ times to avoid cheating.

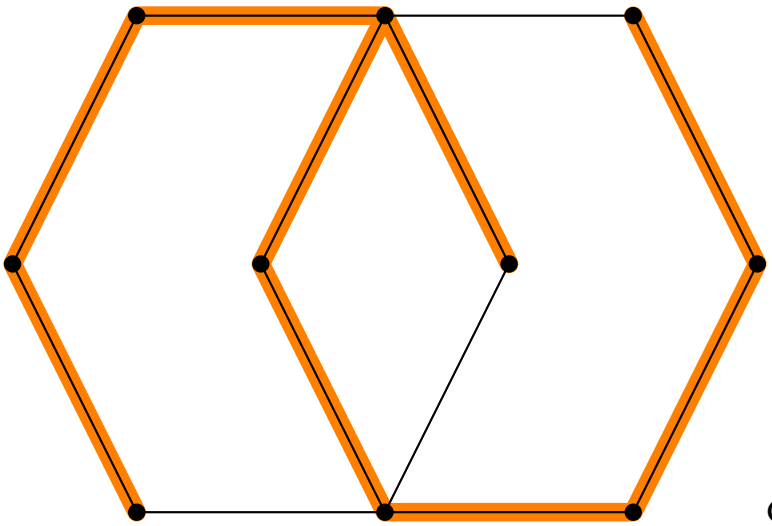
Overview

Spanning trees



1. Minimum weight spanning tree $\rightarrow$ polynomial time
2. Maximum leaf spanning tree $\rightarrow$ NP-complete

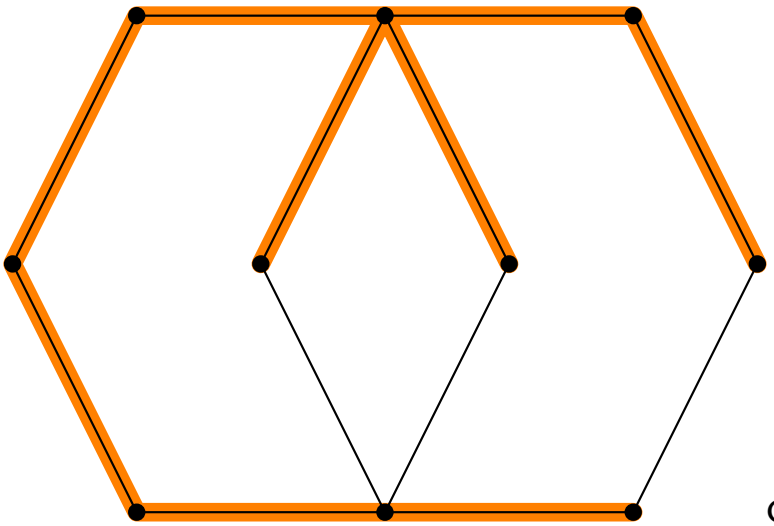
Spanning trees



optimal?

1. Minimum weight spanning tree $\rightarrow$ polynomial time
2. Maximum leaf spanning tree $\rightarrow$ NP-complete

Spanning trees



optimal!

1. Minimum weight spanning tree $\rightarrow$ polynomial time
2. Maximum leaf spanning tree $\rightarrow$ NP-complete

Maximum Leaf Spanning Trees

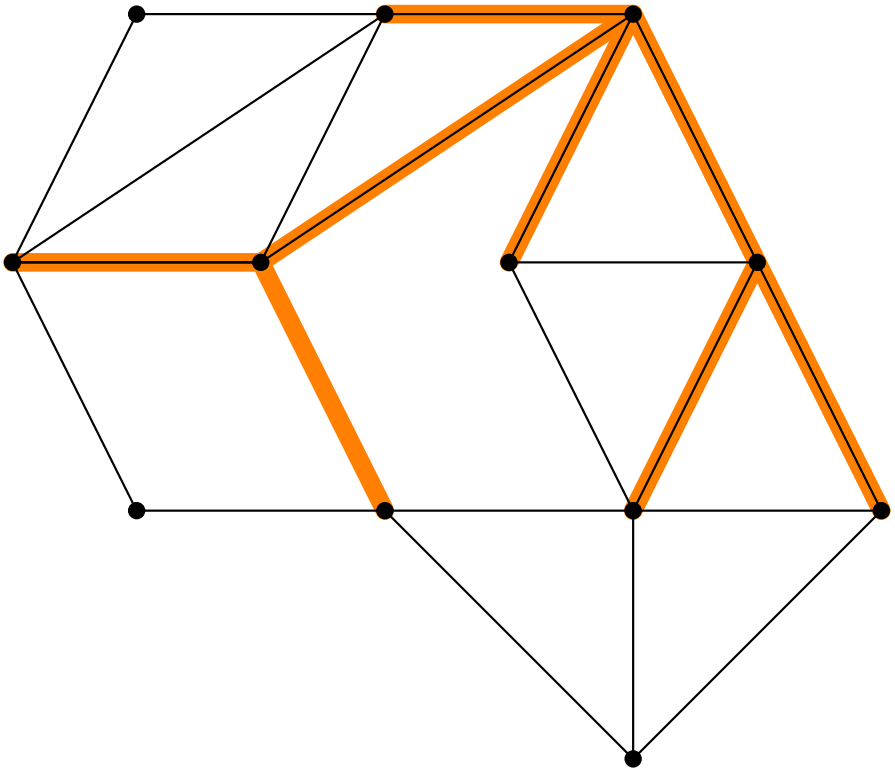
We consider this problem:

- Input: An undirected graph G and a number k
- Question: Does G contain a spanning tree with at least k leaves?

Applications: Operations research, network design

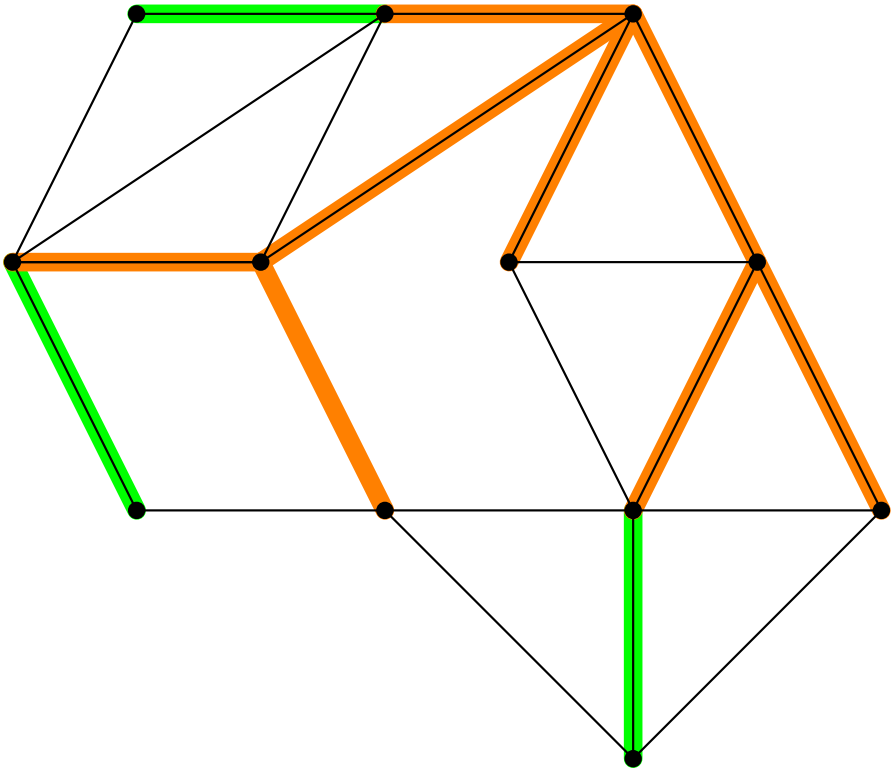
A simpler problem

How to turn a k-leaf tree into a k-leaf spanning tree:



A simpler problem

How to turn a k-leaf tree into a k-leaf spanning tree:



Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Known Results

APX-hard	Galbiati, Maffioli, Morzenti, 1994
2-approximation	Solis-Oba, 1998
3-approximation	Lu & Ravi, 1998
1.5-approximation (cubic)	Bonsma & Zickfeld, 2008
$O((17k)!\,(n+m))$	Bodlaender, 1993
$(2k)^{4k} n^{O(1)}$	Downey & Fellows, 1995
$O(14.23^k + n + m)$	Fellows, McCartin, Rosamond, Steege, 2000
$O(9.49^k k^3 + n^3)$	Bonsma, Brueggemann, Woeginger, 2003
$O(8.12^k k^3 + n^3)$	Estivill-Castro, Fellows, Langston, Rosamond, 2005
$O^*(1.94^n)$	Fomin, Grandoni, Kratsch, 2006
$6.75^k k^{O(1)} + n^{O(1)}$	Bonsma & Zickfeld, 2008
$4^k k^2 + n^{O(1)}$	Much simpler algorithm!

Directed Graphs

Directed Maximum Leaf Out-Tree (DMLOT) problem:

- Input: A directed graph G and a number k
- Question: Does G contain an out-tree with at least k leaves?

Directed Maximum Leaf Spanning Tree (DMLST) problem:

- Input: A directed graph G and a number k
- Question: Does G contain an out-tree with at least k leaves that spans all nodes?

Directed Graphs

Directed Maximum Leaf Out-Tree (DMLOT) problem:

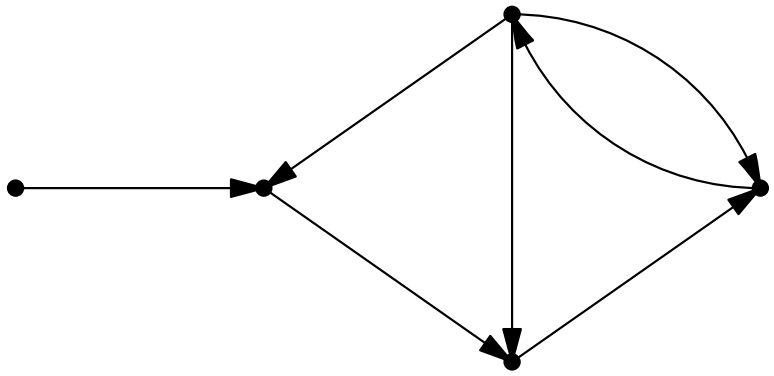
- Input: A directed graph G and a number k
- Question: Does G contain an out-tree with at least k leaves?

Directed Maximum Leaf Spanning Tree (DMLST) problem:

- Input: A directed graph G and a number k
- Question: Does G contain an out-tree with at least k leaves that spans all nodes?

Example

Open for a long time: DMLOT and DMLST in FPT?



Best directed out-tree: 3 leaves

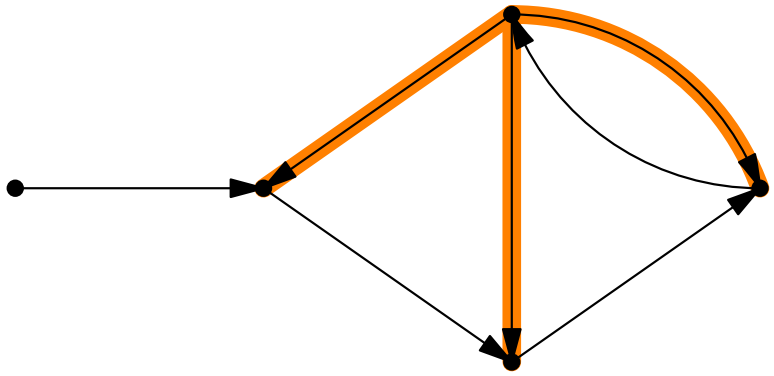
Best directed spanning tree: 1 leaf

DMLOT $\neq$ DMLST

We cannot extend an out-tree into a spanning tree!

Example

Open for a long time: DMLOT and DMLST in FPT?



Best directed out-tree: 3 leaves

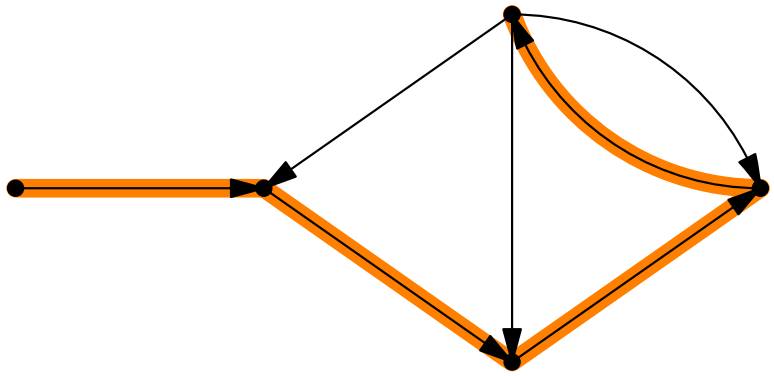
Best directed spanning tree: 1 leaf

$$\text{DMLOT} \neq \text{DMLST}$$

We cannot extend an out-tree into a spanning tree!

Example

Open for a long time: DMLOT and DMLST in FPT?



Best directed out-tree: 3 leaves

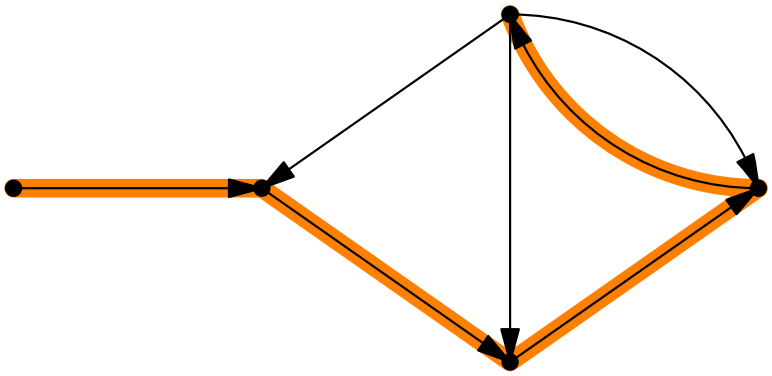
Best directed spanning tree: 1 leaf

DMLOT $\neq$ DMLST

We cannot extend an out-tree into a spanning tree!

Example

Open for a long time: DMLOT and DMLST in FPT?



Best directed out-tree: 3 leaves

Best directed spanning tree: 1 leaf

$$\text{DMLOT} \neq \text{DMLST}$$

We cannot extend an out-tree into a spanning tree!

Known results — directed graphs

Alon, Fomin, Gutin, Krivelevich, Saurabh, ICALP 2007

Theorem

G has either a k-leaf out-tree or its pathwidth is bounded by $2k^2$.

We call this a win-win scenario.

$\rightarrow$ solve DMLOT in time $c^{k^3 \log k} n^{O(1)}$.

Known results — directed graphs

Alon, Fomin, Gutin, Krivelevich, Saurabh, ICALP 2007

Theorem

G has either a k-leaf out-tree or its pathwidth is bounded by $2k^2$.

We call this a win-win scenario.

$\rightarrow$ solve DMLOT in time $c^{k^3 \log k} n^{O(1)}$.

Known results — directed graphs

Alon, Fomin, Gutin, Krivelevich, Saurabh, FST&TCS 2007

Some improvements by the same authors:

DMLOT in $c^{k^2 \log k} n^{O(1)}$ time

DMLOT in $c^{k \log k} n^{O(1)}$ time for acyclic graphs

(Improved bounds on the pathwidth.)

Results — directed graphs

Bonsma & Dorn, 2007:

DMLST in $c^{k^3 \log k} n^{O(1)}$ time

Bonsma & Dorn, 2008:

DMLST and DMLOT in $c^{k \log k} n^{O(1)}$

Now:

DMLST and DMLOT in $O(4^k nm)$

Results — directed graphs

Bonsma & Dorn, 2007:

DMLST in $c^{k^3 \log k} n^{O(1)}$ time

Bonsma & Dorn, 2008:

DMLST and DMLOT in $c^{k \log k} n^{O(1)}$

Now:

DMLST and DMLOT in $O(4^k nm)$

Results — directed graphs

Bonsma & Dorn, 2007:

DMLST in $c^{k^3 \log k} n^{O(1)}$ time

Bonsma & Dorn, 2008:

DMLST and DMLOT in $c^{k \log k} n^{O(1)}$

Now:

DMLST and DMLOT in $O(4^k nm)$

Results — directed graphs

Bonsma & Dorn, 2007:

DMLST in $c^{k^3 \log k} n^{O(1)}$ time

Bonsma & Dorn, 2008:

DMLST and DMLOT in $c^{k \log k} n^{O(1)}$

Now:

DMLST and DMLOT in $O(4^k nm)$

A simple algorithm to find k-leaf trees

Idea: Start at some node and grow a tree.

Run the following algorithms on all nodes v:

- mark v blue.
- Repeat:
  Choose a blue leaf u.

  (a) Mark it red
  OR
  (b) Connect u's outside neighbors to u and mark them blue

  if the tree has $\geq k$ leaves, then answer **YES**
  if there is no blue leaf, answer **NO**

(Outside neighbor: Neighbor that is not yet in the tree)

A simple algorithm to find k-leaf trees

Idea: Start at some node and grow a tree.

Run the following algorithms on all nodes v:

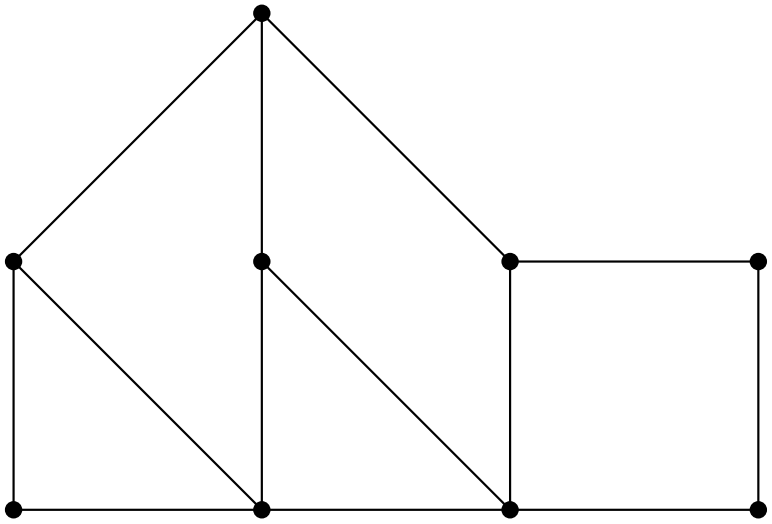
- mark v blue.
- Repeat:
  Choose a blue leaf u.

  (a) Mark it red
  OR
  (b) Connect u's outside neighbors to u and mark them blue
  (if $\deg(u) = 1$ follow the path)

  if the tree has $\geq k$ leaves, then answer **YES**
  if there is no blue leaf, answer **NO**

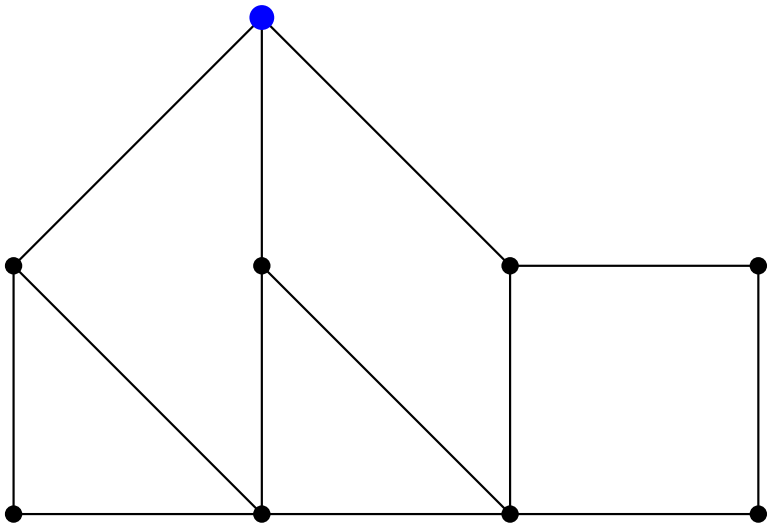
(Outside neighbor: Neighbor that is not yet in the tree)

Example



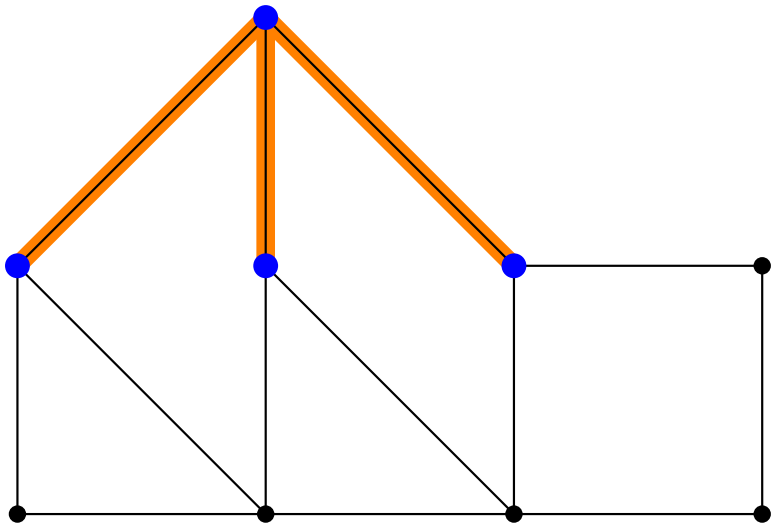
- We grow a tree.
- A blue leaf can be expanded.
- A red leaf remains a leaf.

Example



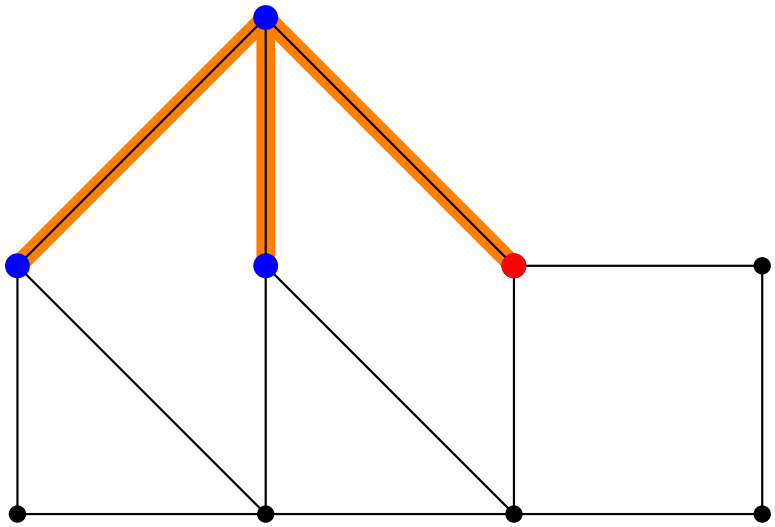
- We grow a tree.
- A blue leaf can be expanded.
- A red leaf remains a leaf.

Example



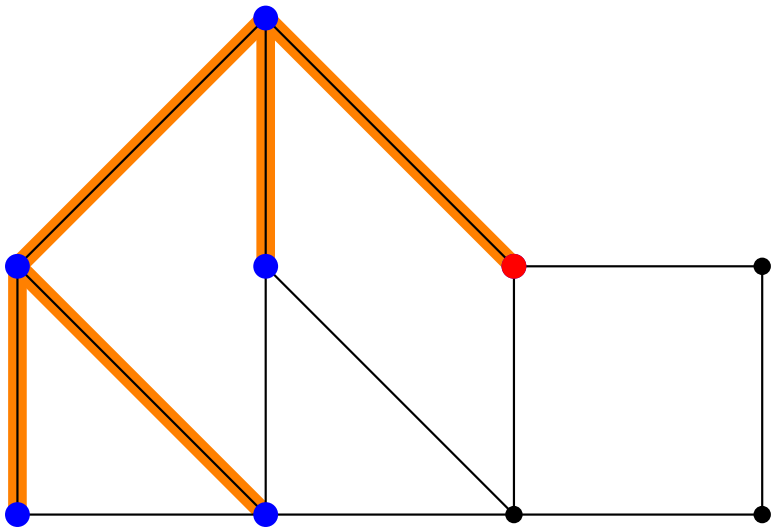
- We grow a tree.
- A blue leaf can be expanded.
- A red leaf remains a leaf.

Example



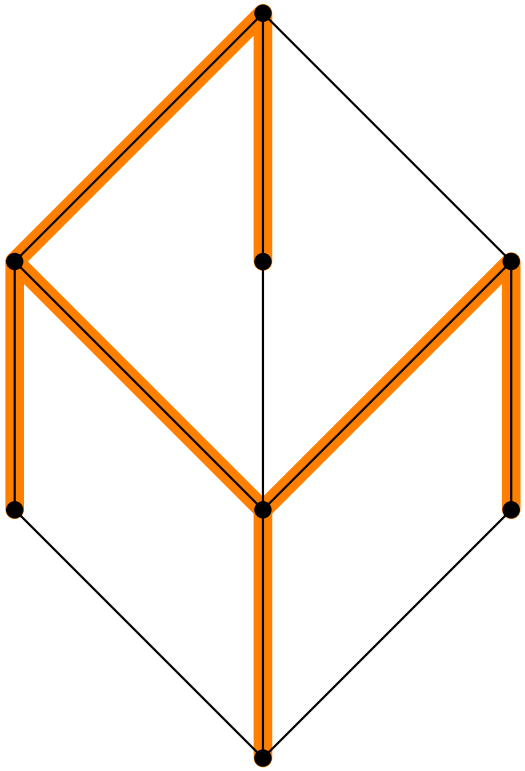
- We grow a tree.
- A blue leaf can be expanded.
- A red leaf remains a leaf.

Example



- We grow a tree.
- A blue leaf can be expanded.
- A red leaf remains a leaf.

We can't grow every tree



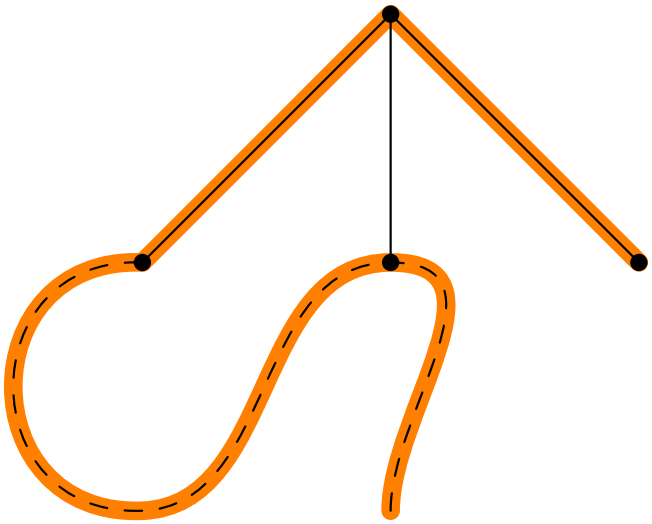
Is the algorithm correct?

Correctness

Theorem

If there is a k-leaf tree, the algorithm finds some k-leaf tree.

Proof: Modify a k-leaf spanning tree.



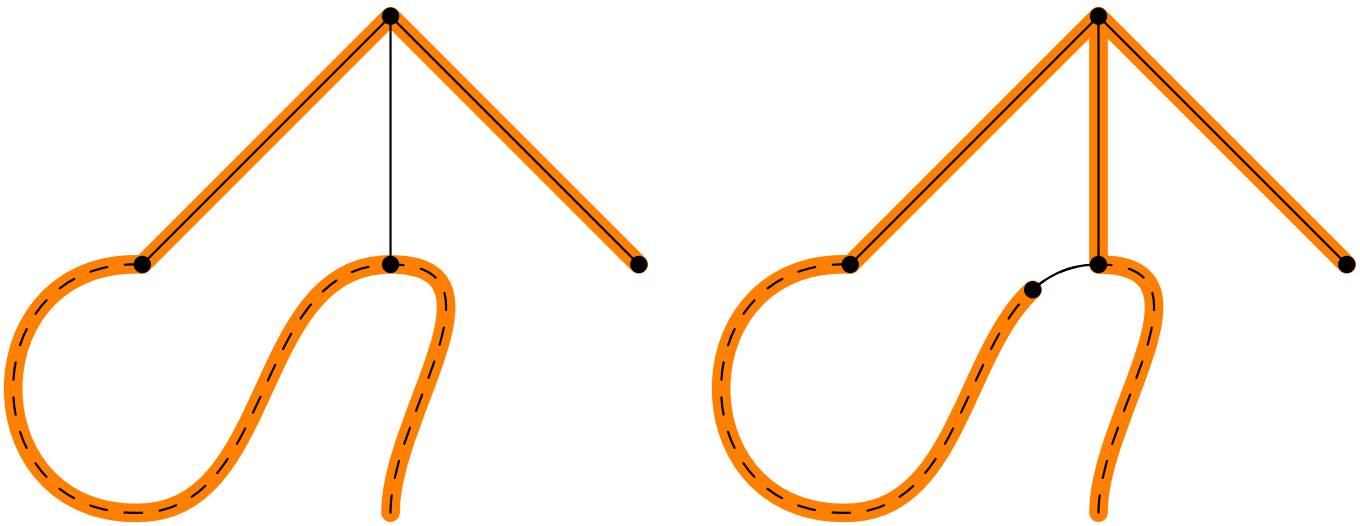
This theorem holds for directed graphs, too $\Rightarrow$ DMLOT

Correctness

Theorem

If there is a k-leaf tree, the algorithm finds some k-leaf tree.

Proof: Modify a k-leaf spanning tree.



This theorem holds for directed graphs, too $\Rightarrow$ DMLOT.

A very useful theorem

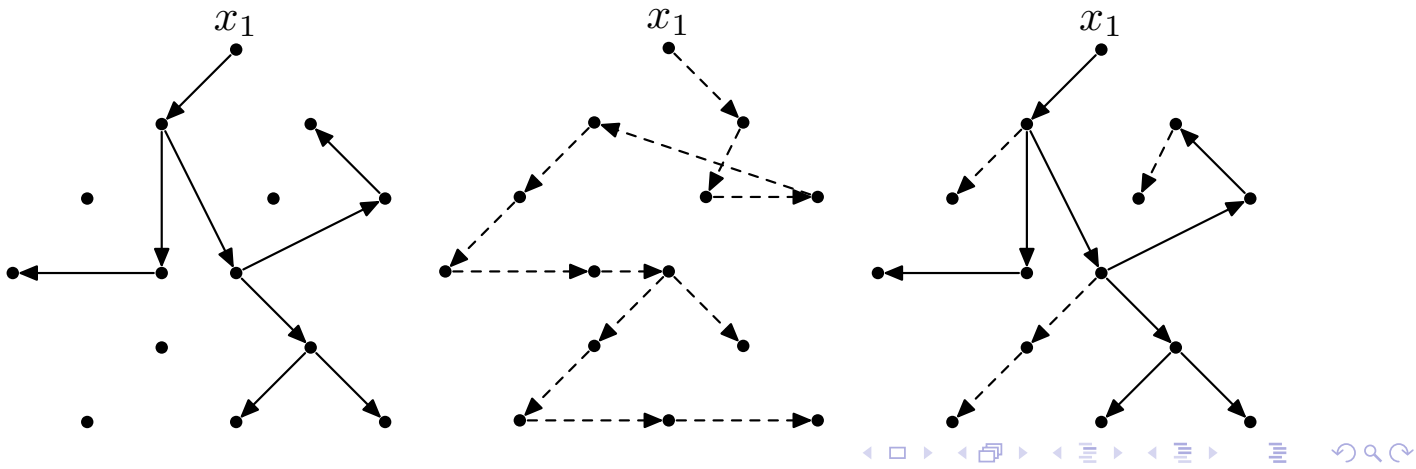
Theorem

Let G contain a directed spanning tree with root r.

Then every out-tree with root r can be extended into a spanning tree.

Proof

Use the spanning tree to extend the out-tree.



Running time (FPT)

In every step one of the following happens:

- No recursive branch. Tree grows. Number of red and blue leaves does not change.
- A blue leaf becomes a red leaf.
- The number of blue leaves is increased.

Let r be the number of red leaves and b be the number of blue leaves.

Then the function $2r + b$ grows in each recursive call.

If $2r + b \geq 2k$, the algorithm terminates.

The recursion depth is at most $2k$ and there are at most 2^{2k} recursive calls.

Running time (FPT)

In every step one of the following happens:

- No recursive branch. Tree grows. Number of red and blue leaves does not change.
- A blue leaf becomes a red leaf.
- The number of blue leaves is increased.

Let r be the number of red leaves and b be the number of blue leaves.

Then the function $2r + b$ grows in each recursive call.

If $2r + b \geq 2k$, the algorithm terminates.

The recursion depth is at most $2k$ and there are at most 2^{2k} recursive calls.

Running time (FPT)

In every step one of the following happens:

- No recursive branch. Tree grows. Number of red and blue leaves does not change.
- A blue leaf becomes a red leaf.
- The number of blue leaves is increased.

Let r be the number of red leaves and b be the number of blue leaves.

Then the function $2r + b$ grows in each recursive call.

If $2r + b \geq 2k$, the algorithm terminates.

The recursion depth is at most $2k$ and there are at most 2^{2k} recursive calls.

Running time (FPT)

In every step one of the following happens:

- No recursive branch. Tree grows. Number of red and blue leaves does not change.
- A blue leaf becomes a red leaf.
- The number of blue leaves is increased.

Let r be the number of red leaves and b be the number of blue leaves.

Then the function $2r + b$ grows in each recursive call.

If $2r + b \geq 2k$, the algorithm terminates.

The recursion depth is at most $2k$ and there are at most 2^{2k} recursive calls.

Running time (FPT)

In every step one of the following happens:

- No recursive branch. Tree grows. Number of red and blue leaves does not change.
- A blue leaf becomes a red leaf.
- The number of blue leaves is increased.

Let r be the number of red leaves and b be the number of blue leaves.

Then the function $2r + b$ grows in each recursive call.

If $2r + b \geq 2k$, the algorithm terminates.

The recursion depth is at most $2k$ and there are at most 2^{2k} recursive calls.

Kernelization on sparse graph classes

- Framework for planar graphs
  Guo and Niedermeier: *Linear problem kernels for NP-hard problems on planar graphs*
- Meta-result for graphs of bounded genus
  Bodlaender, Fomin, Lokshtanov, Penninkx, Saurabh and Thilikos: *(Meta) Kernelization*
- Meta-result for graphs excluding a fixed graph as a minor
  Fomin, Lokshtanov, Saurabh and Thilikos: *Bidimensionality and kernels*
- Here: Meta-result for graphs excluding a fixed graph as a topological minor

FPT algorithms for $\mathcal{F}$-*Deletion*

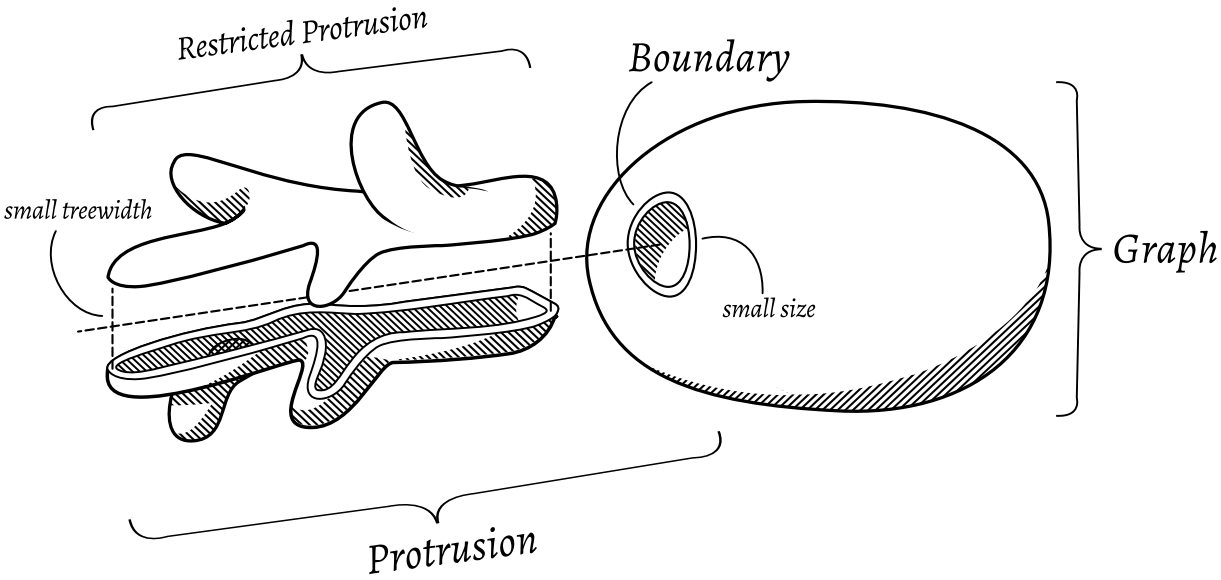
The $\mathcal{F}$-Deletion problem:

Input: A graph G, an integer k

Question: Is there a set $X \subseteq V(G)$ of size at most k such that $G - X$ contains no graph from $\mathcal{F}$ as a minor?

- Many special results, e.g. $\mathcal{F} = \{K_4\}$
- $\mathcal{F}$ contains a planar graph: FPT by Robertson-Seymour
  Fellows and Langston: Nonconstructive tools for proving polynomial-time decidability
- $2^{O(k \log k)} n^2$-Algorithm for *Planar-$\mathcal{F}$-Deletion*, later improved to $2^{O(k)} n^2$ if $\mathcal{F}$ contains only connected graphs
  Fomin, Lokshtanov, Misra and Saurabh: Nearly optimal FPT algorithms for Planar-$\mathcal{F}$-Deletion / Planar-$\mathcal{F}$-Deletion: Approximation and Optimal FPT Algorithms
- Here: a $2^{O(k)} n^2$ algorithm for *Planar-$\mathcal{F}$-Deletion* even if $\mathcal{F}$ contains disconnected graphs

Protrusion

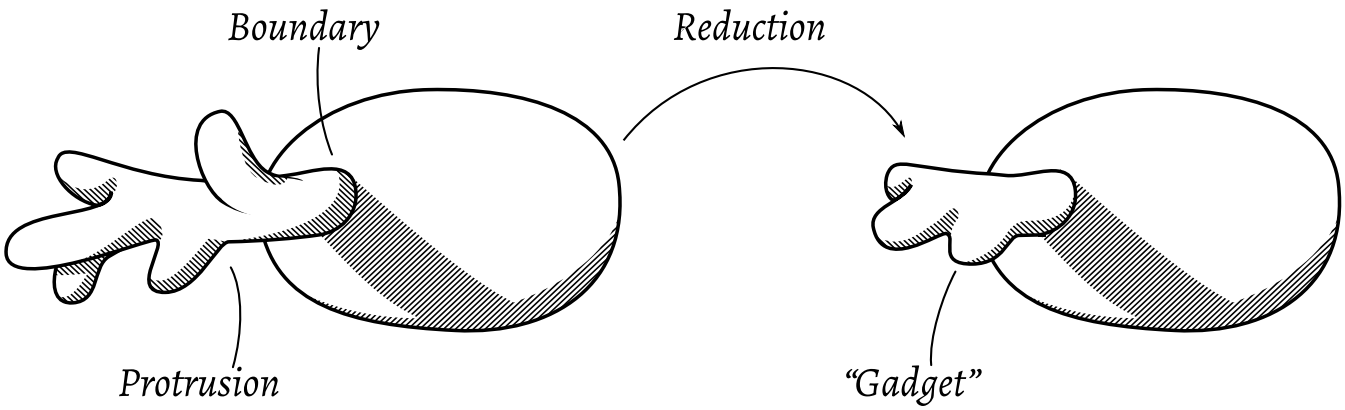


Definition

$X \subseteq V(G)$ is a *t-protrusion* if

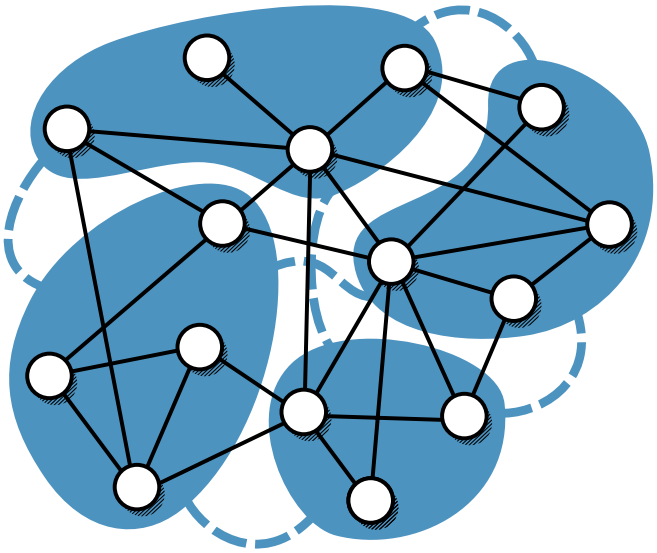
1. $|\partial(X)| = |N(X) \setminus X| \leq t$ (small boundary)
2. $\mathbf{tw}(G[X]) \leq t$ (small treewidth)

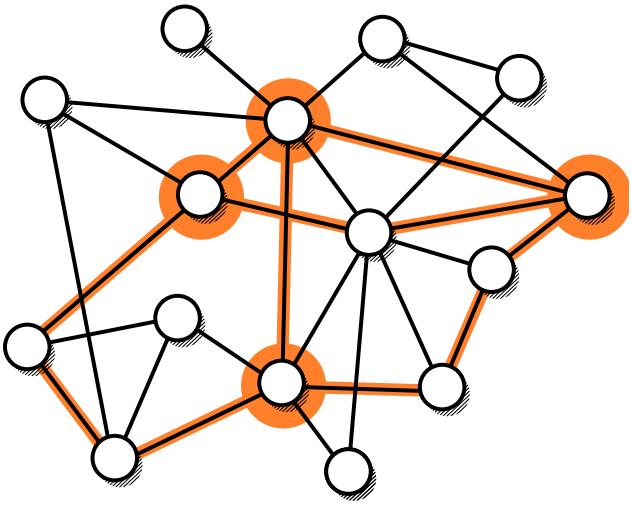
Protrusion replacement



- We want to replace a large protrusion by something smaller
- Possible if problem has *finite integer index*
- Recursive structure of graphs of small treewidth (i.e. protrusion) helps
- Lots of technicalities omitted...

Minors, top-minors



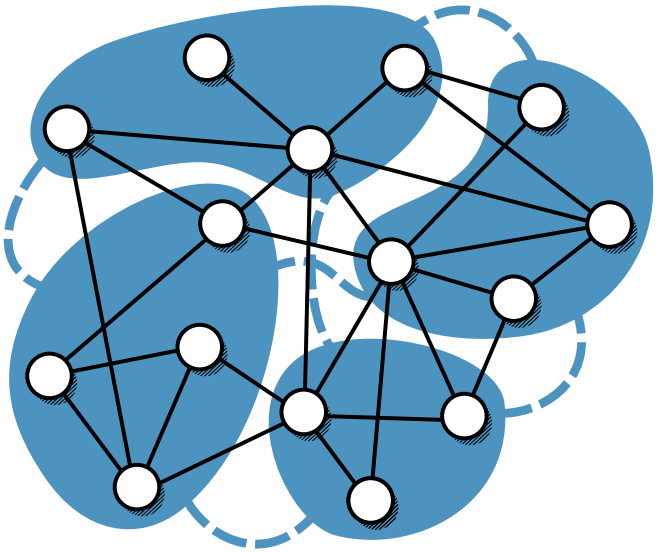


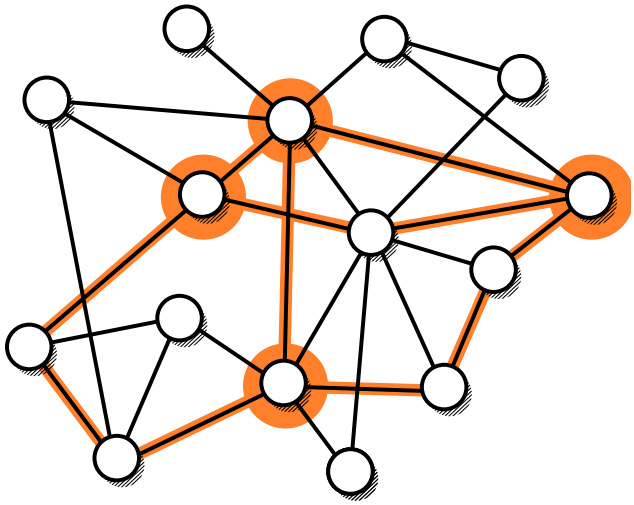
Graphs excluding a fixed Minor/Top-Minor:

- d-degenerate (d depends on the excluded graph)
- closed under taking minors/top-minors

$\Rightarrow$ every minor/top-minor *also* d-degenerate

Minors, top-minors



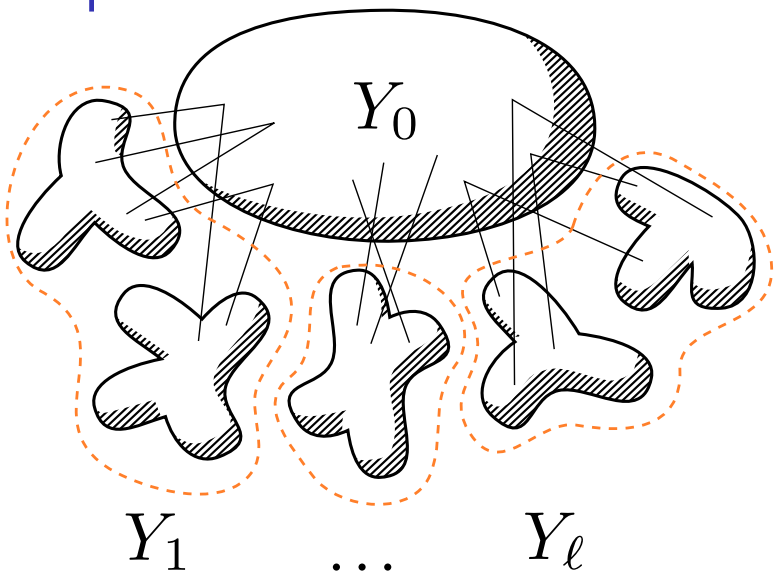


Graphs excluding a fixed Minor/Top-Minor:

- d-degenerate (d depends on the excluded graph)
- closed under taking minors/top-minors

$\Rightarrow$ every minor/top-minor *also* d-degenerate

Protrusion decomposition



(α, t)-*Protrusion decomposition* is a partition $V = Y_0 \uplus Y_1 \uplus \cdots \uplus Y_\ell$ such that:

1. for $1 \leq i \leq \ell$, $N(Y_i) \subseteq Y_0$
2. $\ell \leq \alpha$ and $|Y_0| \leq \alpha$
3. for $1 \leq i \leq \ell$, $Y_i \cup N(Y_i)$ is a t-protrusion

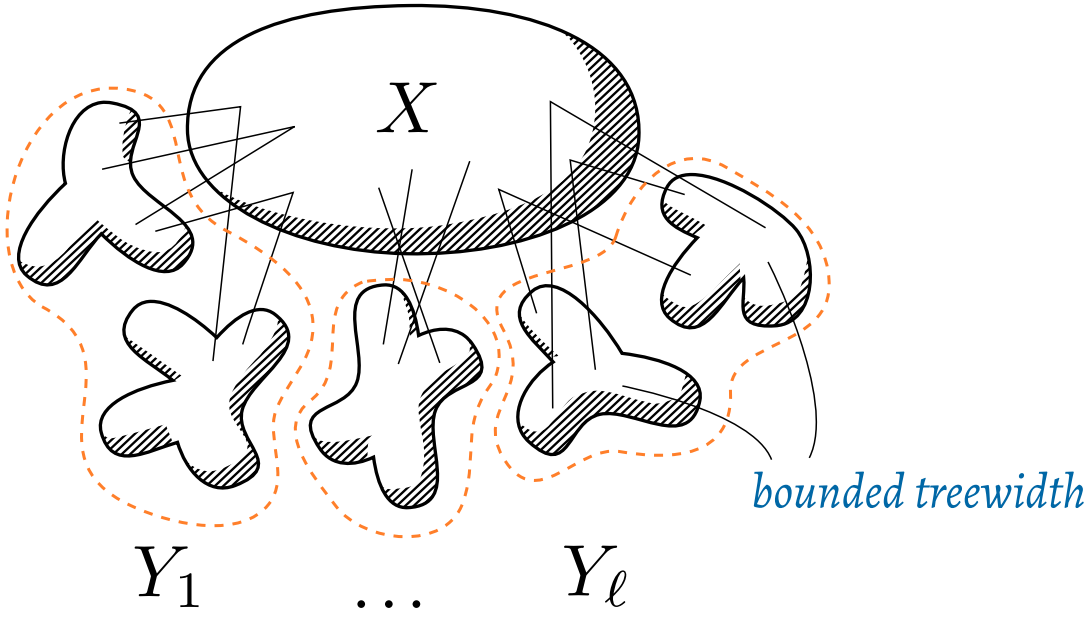
...in H-topological minor free graphs

Lemma

Let G exclude H as a topological minor and let $X \subseteq V(G)$ be such that $\mathbf{tw}(G - X) \leq t$. Then G has a $(O(|X|), 2t + |H|)$-protrusion decomposition.

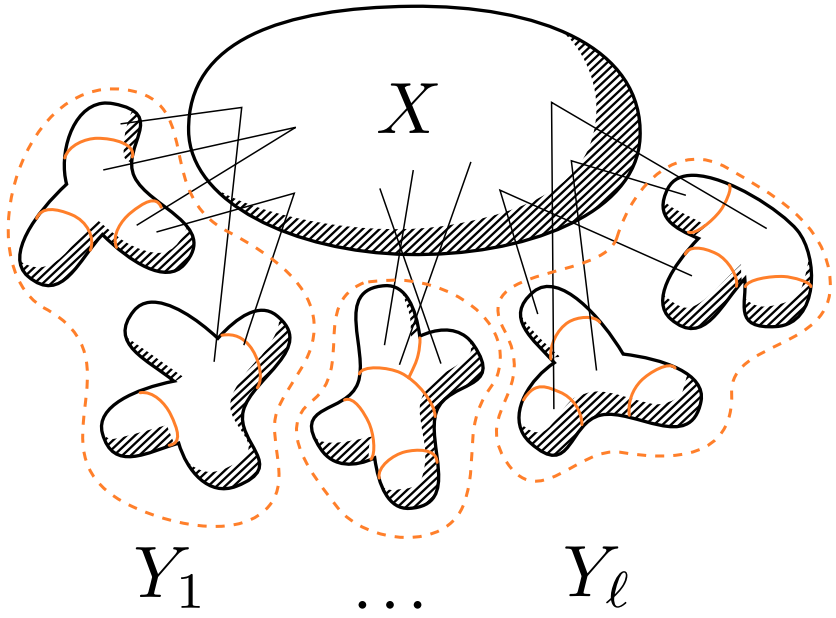
- Can be computed in linear time if X is given
- X is called a *treewidth-t modulator*

Proof sketch



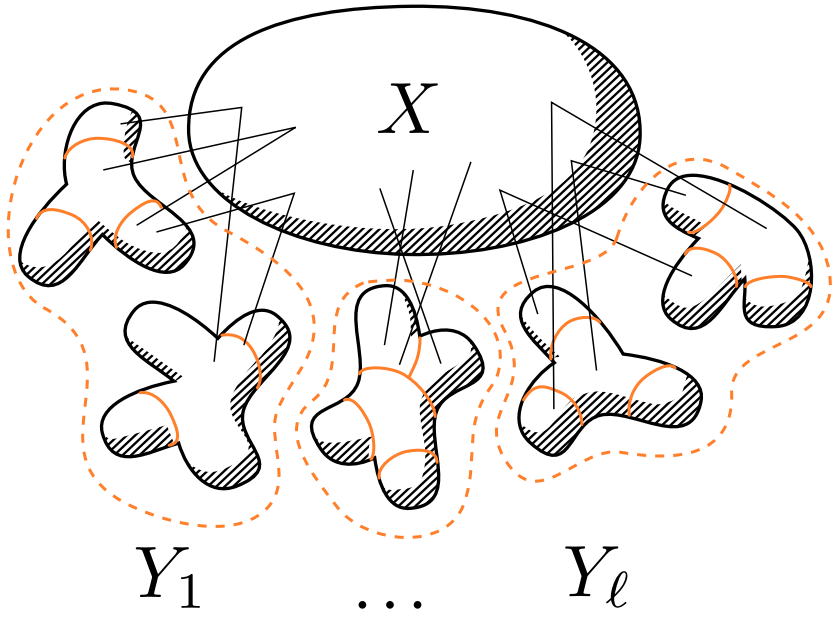
- Given X such that $\mathbf{tw}(G - X) \leq t$
- Group components of $G - X$ by respective neighbourhood in X and obtain $Y_1, \ldots, Y_\ell$

Proof sketch



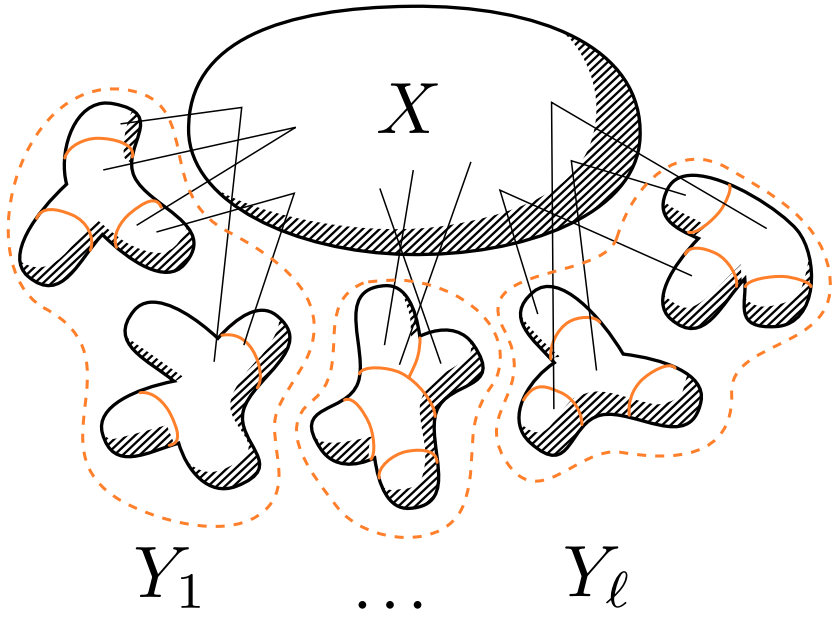
- From bottom up, mark bags whose subtree induces component with more than $|H|$ neighbours in X
- Number of such bags at most linear $|X|$: otherwise we can construct $K_{|H|}$ and thus H as a top. minor
- Mark LCA bags also

Proof sketch



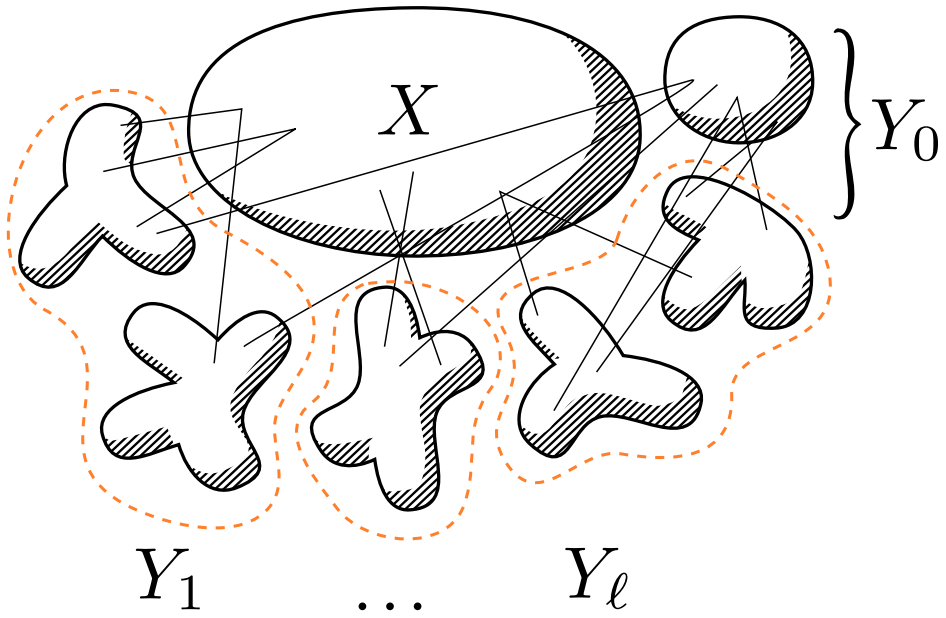
- From bottom up, mark bags whose subtree induces component with more than $|H|$ neighbours in X
- Number of such bags at most linear $|X|$: otherwise we can construct $K_{|H|}$ and thus H as a top. minor
- Mark LCA bags also

Proof sketch



- From bottom up, mark bags whose subtree induces component with more than $|H|$ neighbours in X
- Number of such bags at most linear $|X|$: otherwise we can construct $K_{|H|}$ and thus H as a top. minor
- Mark LCA bags also

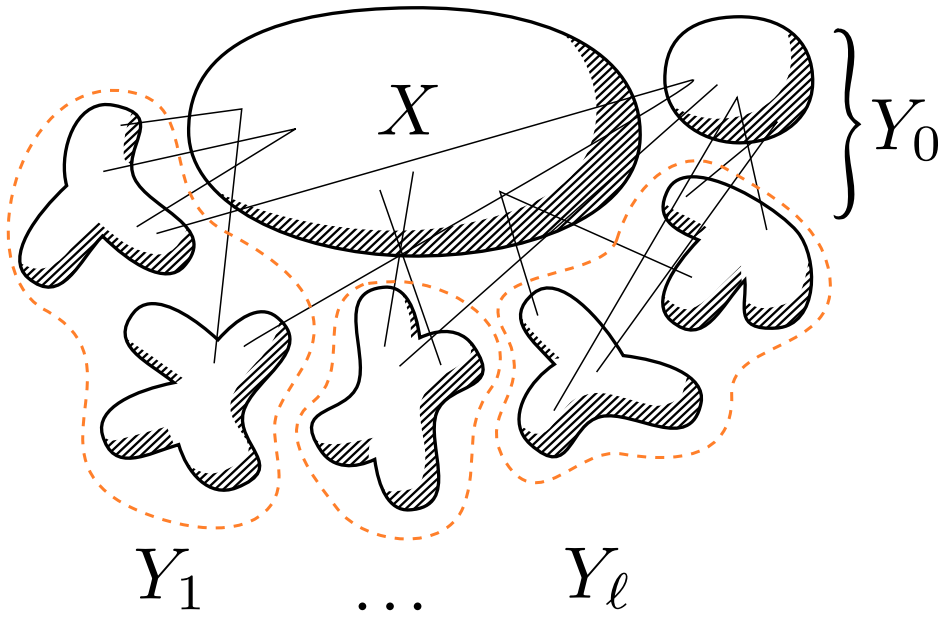
Proof sketch



- Add content of bags to X to obtain Y_0, by previous observations $|Y_0| = O(|X|)$
- LCA marking ensures that now $|N(Y_i)| \leq 2t + |H|$ for $1 \leq i \leq \ell$

$\Rightarrow$ Each Y_i now a $(2t + |H|)$-protrusion!

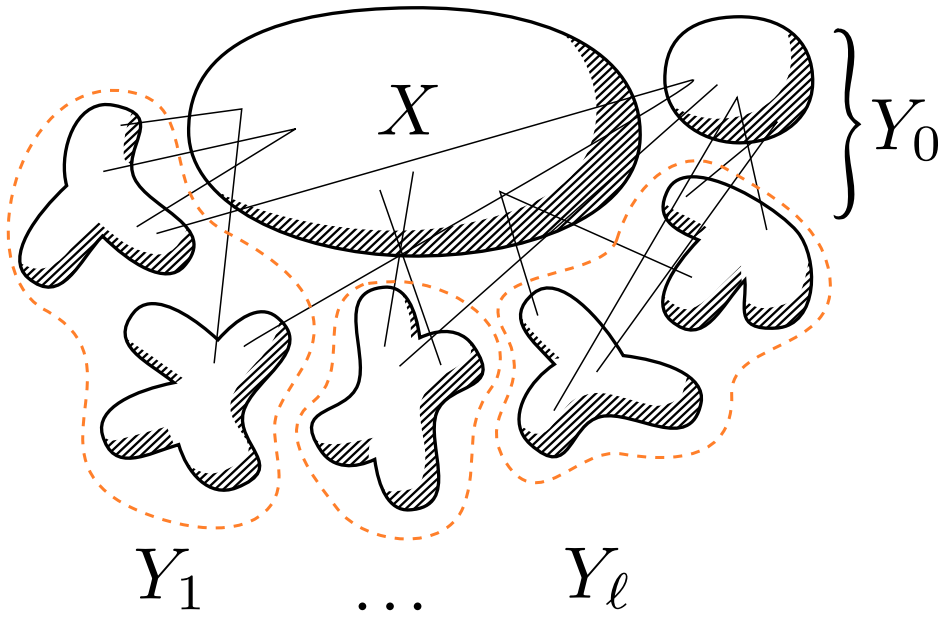
Proof sketch



- Add content of bags to X to obtain Y_0, by previous observations $|Y_0| = O(|X|)$
- LCA marking ensures that now $|N(Y_i)| \leq 2t + |H|$ for $1 \leq i \leq \ell$

$\Rightarrow$ Each Y_i now a $(2t + |H|)$-protrusion!

Proof sketch



- Add content of bags to X to obtain Y_0, by previous observations $|Y_0| = O(|X|)$
- LCA marking ensures that now $|N(Y_i)| \leq 2t + |H|$ for $1 \leq i \leq \ell$

$\Rightarrow$ Each Y_i now a $(2t + |H|)$-protrusion!

The theorem

Theorem

Fix a graph H. Let Π be a parameterized graph problem on the class of H-topological-minor-free graphs that is treewidth-bounding[a] and has finite integer index[b]. Then Π admits a linear kernel.

a) A parameterized graph problem is *treewidth-bounding* if every yes-instance contains a $O(k)$-sized treewidth-t-modulator for some fixed t.

b) Also required by all previous results.

Vertex Feedback Vertex Set, *Chordal Vertex Deletion*, *(Proper) Interval Vertex Deletion*, *Cograph Vertex Deletion*, *Edge Dominating Set*, *Connected Vertex Cover*, ...

The theorem

Theorem

Fix a graph H. Let Π be a parameterized graph problem on the class of H-topological-minor-free graphs that is treewidth-bounding[a] and has finite integer index[b]. Then Π admits a linear kernel.

a) A parameterized graph problem is *treewidth-bounding* if every yes-instance contains a $O(k)$-sized treewidth-t-modulator for some fixed t

b) Also required by all previous results

- Holds for e.g. *Feedback Vertex Set*, *Chordal Vertex Deletion*, *(Proper) Interval Vertex Deletion*, *Cograph Vertex Deletion*, *Edge Dominating Set*, *Connected Vertex Cover*

The theorem

Theorem

Fix a graph H. Let Π be a parameterized graph problem on the class of H-topological-minor-free graphs that is treewidth-bounding[a] and has finite integer index[b]. Then Π admits a linear kernel.

a) A parameterized graph problem is *treewidth-bounding* if every yes-instance contains a $O(k)$-sized treewidth-t-modulator for some fixed t

b) Also required by all previous results

- Holds for e.g. *Feedback Vertex Set*, *Chordal Vertex Deletion*, *(Proper) Interval Vertex Deletion*, *Cograph Vertex Deletion*, *Edge Dominating Set*, *Connected Vertex Cover*

The theorem

Theorem

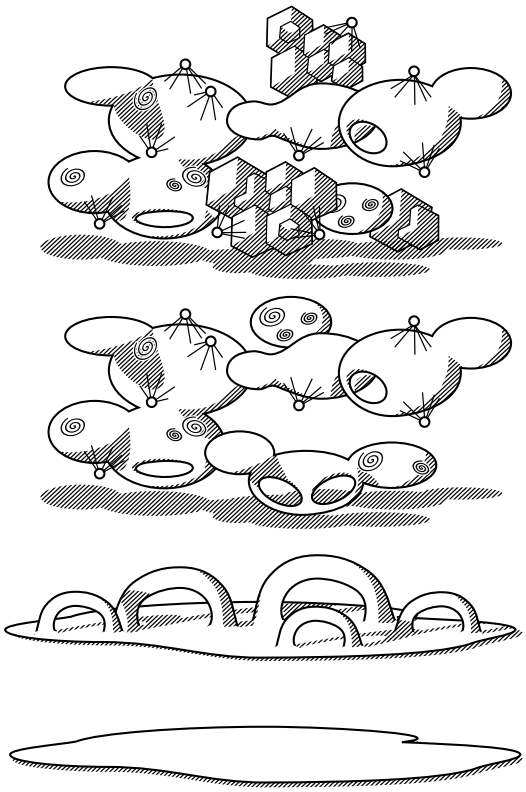
Fix a graph H. Let Π be a parameterized graph problem on the class of H-topological-minor-free graphs that is treewidth-bounding[a] and has finite integer index[b]. Then Π admits a linear kernel.

a) A parameterized graph problem is *treewidth-bounding* if every yes-instance contains a $O(k)$-sized treewidth-t-modulator for some fixed t

b) Also required by all previous results

- Holds for e.g. *Feedback Vertex Set*, *Chordal Vertex Deletion*, *(Proper) Interval Vertex Deletion*, *Cograph Vertex Deletion*, *Edge Dominating Set*, *Connected Vertex Cover*

Treewidth-bounding?



H-Topological-Minor-Free — *Treewidth-bounding*

$\cup$

H-Minor-Free — *Bidimensional + separation property*

$\cup$

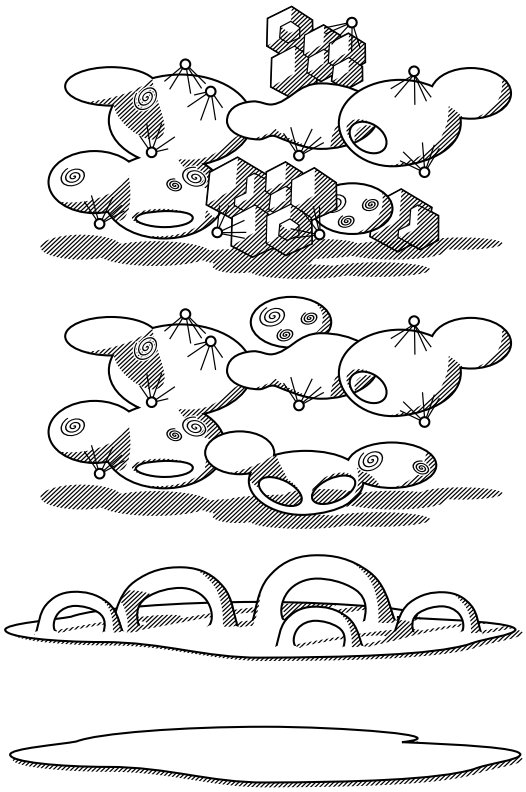
Bounded Genus — *Quasi-compact*

$\cup$

Planar — *"Distance-property"*

Ealier properties imply treewidth-bounding!

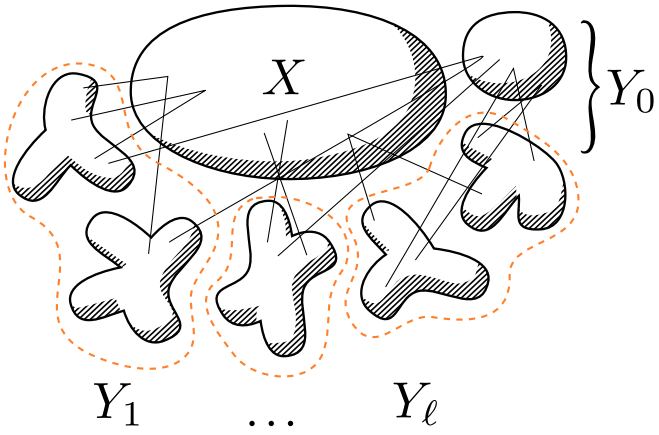
Treewidth-bounding?



H-Topological-Minor-Free	*Treewidth-bounding*
∪	
H-Minor-Free	*Bidimensional* + *separation property*
∪	
Bounded Genus	*Quasi-compact*
∪	
Planar	*"Distance-property"*

Ealier properties imply treewidth-bounding!

Proof idea



- Problem is treewidth-bounding: there exists a treewidth-t-modulator (if it is a yes-instance)
- Exhaustively reduce all $(2t + |H|)$-protrusions in polynomial time

$\Rightarrow$ Every such protrusion has now constant size

- There exists a $(O(|X|), 2t + |H|)$-protusion-decomposition: only need to bound number of clusters ℓ (idea: too many would create an $|H|$ as a topological minor)

The theorem

Planar-$\mathcal{F}$-Deletion:

Input: A graph G, an integer k

Problem: Is there a set $X \subseteq V(G)$ of size at most k such that $G - X$ contains no graph from $\mathcal{F}$ as a minor?

Theorem

Let $\mathcal{F}$ be a fixed finite family of graphs containing at least one planar graph. There exists an algorithm to solve Planar-$\mathcal{F}$-Deletion in time $2^{O(k)} \cdot n^2$.

Considerations

- No finite state property, because $\mathcal{F}$ can contain disconnected graphs

$\Rightarrow$ Protrusion reduction not possible!

- As $\mathcal{F}$ contains a planar graph, a solution X will fullfill $\mathbf{tw}(G - X) \leq t$ for some constant t

$\Rightarrow$ Use iterative compression to have solution X' that works as a treewdith modulator

- But: We are working on general graphs! Bounds for H-(topological)-minor-free graphs do not apply!

Algorithm outline

From iterative compression: got solution X, $|X| \leq k+1$ and want disjoint solution $\tilde{X}$, $|\tilde{X}| \leq k$.

- Given X, obtain $(|X|, t)$-protrusion-decomposition $Y_0 \uplus Y_1 \uplus \cdots \uplus Y_\ell$, where t depends on $\mathcal{F}$
- Guess intersections I of $\tilde{X}$ with Y_0 (in time $2^{O(k)}$)
- New solution $\tilde{X}$ can intersect at most $\leq k$ clusters

$\Rightarrow$ Disregarding those $\leq k$ clusters, $G - I$ is H-minor-free!

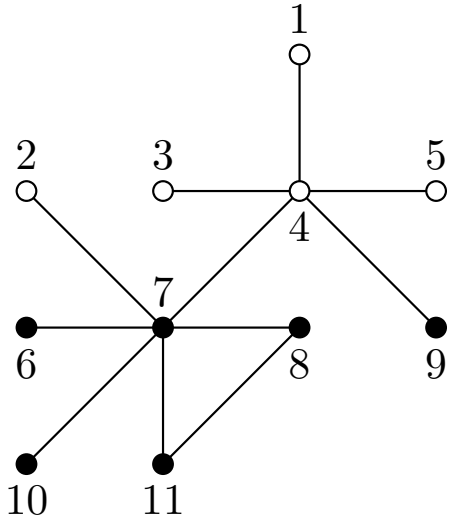
$\Rightarrow$ $\ell = O(k)$ or we have a no-instance

Using a the finite state property of solutions sets inside the protrusions we can enumerate all necessary vertex sets in $2^{O(k)}$ time (quite technical)

Adding Recursion to Color-Coding

Idea:

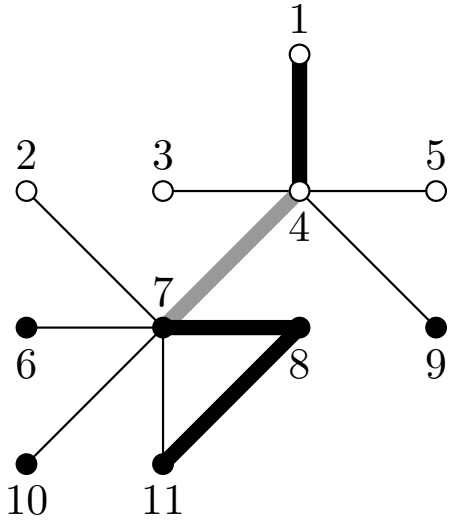
1. Randomly color G in black and white.
2. Recursively check for a black $\lceil k/2 \rceil$-node path and a white $\lfloor k/2 \rfloor$-node path that combine to form a k-node path in G.



Adding Recursion to Color-Coding

Idea:

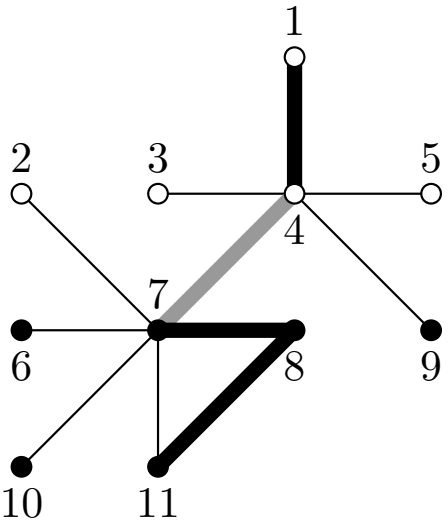
1. Randomly color G in black and white.
2. Recursively check for a black $\lceil k/2 \rceil$-node path and a white $\lfloor k/2 \rfloor$-node path that combine to form a k-node path in G.



The Algorithm for Longest Path

Crucial details:

1. Try $3 \cdot 2^k$ colorings *in each call*.
2. Return *all* the $(u, v) \in V^2$ with $u \xrightarrow{k} v$ that were found.



Combine black (u, x) and white (y, v) into new (u, v) if $\{x, y\} \in E$

Error Probability

Sources of error:

1. Bad coloring: $= 1 - 2^{-k}$
2. Good coloring, error in recursion: $\leq 2^{-k} \cdot 2 \cdot p_{\lceil k/2 \rceil}$

p_k: Pr[algorithm misses needed (u, v) with $u \xrightarrow{k} v$]

Due to the $3 \cdot 2^k$ iterations, $p_k \leq \left(1 - 2^{-k} + 2^{-k+1} p_{\lceil k/2 \rceil}\right)^{3 \cdot 2^k}$.

Proof that $p_k \leq 1/4$: $p_1 = 0$, and by induction

$$\left(1 - 2^{-k} + 2^{-k+1} p_{\lceil k/2 \rceil}\right)^{3 \cdot 2^k} \leq \left(1 - 2^{-k-1}\right)^{\frac{3}{2} \cdot 2^{k+1}} \leq e^{-\frac{3}{2}} < \frac{1}{4}.$$

Error Probability

Sources of error:

1. Bad coloring: $= 1 - 2^{-k}$
2. Good coloring, error in recursion: $\leq 2^{-k} \cdot 2 \cdot p_{\lceil k/2 \rceil}$

p_k: Pr[algorithm misses needed (u, v) with $u \xrightarrow{k} v$]

Due to the $3 \cdot 2^k$ iterations, $p_k \leq \left(1 - 2^{-k} + 2^{-k+1} p_{\lceil k/2 \rceil}\right)^{3 \cdot 2^k}$.

Proof that $p_k \leq 1/4$: $p_1 = 0$, and by induction

$$\left(1 - 2^{-k} + 2^{-k+1} p_{\lceil k/2 \rceil}\right)^{3 \cdot 2^k} \leq \left(1 - 2^{-k-1}\right)^{\frac{3}{2} \cdot 2^{k+1}} \leq e^{-\frac{3}{2}} < \frac{1}{4}.$$

Error Probability

Sources of error:

1. Bad coloring: $= 1 - 2^{-k}$
2. Good coloring, error in recursion: $\leq 2^{-k} \cdot 2 \cdot p_{\lceil k/2 \rceil}$

p_k: Pr[algorithm misses needed (u, v) with $u \xrightarrow{k} v$]

Due to the $3 \cdot 2^k$ iterations, $p_k \leq \left(1 - 2^{-k} + 2^{-k+1} p_{\lceil k/2 \rceil}\right)^{3 \cdot 2^k}$.

Proof that $p_k \leq 1/4$: $p_1 = 0$, and by induction

$$\left(1 - 2^{-k} + 2^{-k+1} p_{\lceil k/2 \rceil}\right)^{3 \cdot 2^k} \leq \left(1 - 2^{-k-1}\right)^{\frac{3}{2} \cdot 2^{k+1}} \leq e^{-\frac{3}{2}} < \frac{1}{4}.$$

Total Running Time

Number of recursive calls:

$$T_k \leq 3 \cdot 2^k (T_{\lceil k/2 \rceil} + T_{\lfloor k/2 \rfloor}) \leq 3 \cdot 2^{k+1} T_{\lceil k/2 \rceil}$$

Observe that

$$k + \lceil k/2 \rceil + \lceil \lceil k/2 \rceil /2 \rceil + \cdots + 1 \leq 2k + \log k.$$

Total running time:

$$O(3^{\log k} 2^{2k+2\log k}) = O(k^{\log 3} k^2 4^k) = O^*(4^k)$$

Total Running Time

Number of recursive calls:

$$T_k \leq 3 \cdot 2^k (T_{\lceil k/2 \rceil} + T_{\lfloor k/2 \rfloor}) \leq 3 \cdot 2^{k+1} T_{\lceil k/2 \rceil}$$

Observe that

$$k + \lceil k/2 \rceil + \lceil \lceil k/2 \rceil /2 \rceil + \cdots + 1 \leq 2k + \log k.$$

Total running time:

$$O(3^{\log k} 2^{2k+2\log k}) = O(k^{\log 3} k^2 4^k) = O^*(4^k)$$

Total Running Time

Number of recursive calls:

$$T_k \leq 3 \cdot 2^k (T_{\lceil k/2 \rceil} + T_{\lfloor k/2 \rfloor}) \leq 3 \cdot 2^{k+1} T_{\lceil k/2 \rceil}$$

Observe that

$$k + \lceil k/2 \rceil + \lceil \lceil k/2 \rceil /2 \rceil + \cdots + 1 \leq 2k + \log k.$$

Total running time:

$$O(3^{\log k} 2^{2k+2 \log k}) = O(k^{\log 3} k^2 4^k) = O^*(4^k)$$