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.

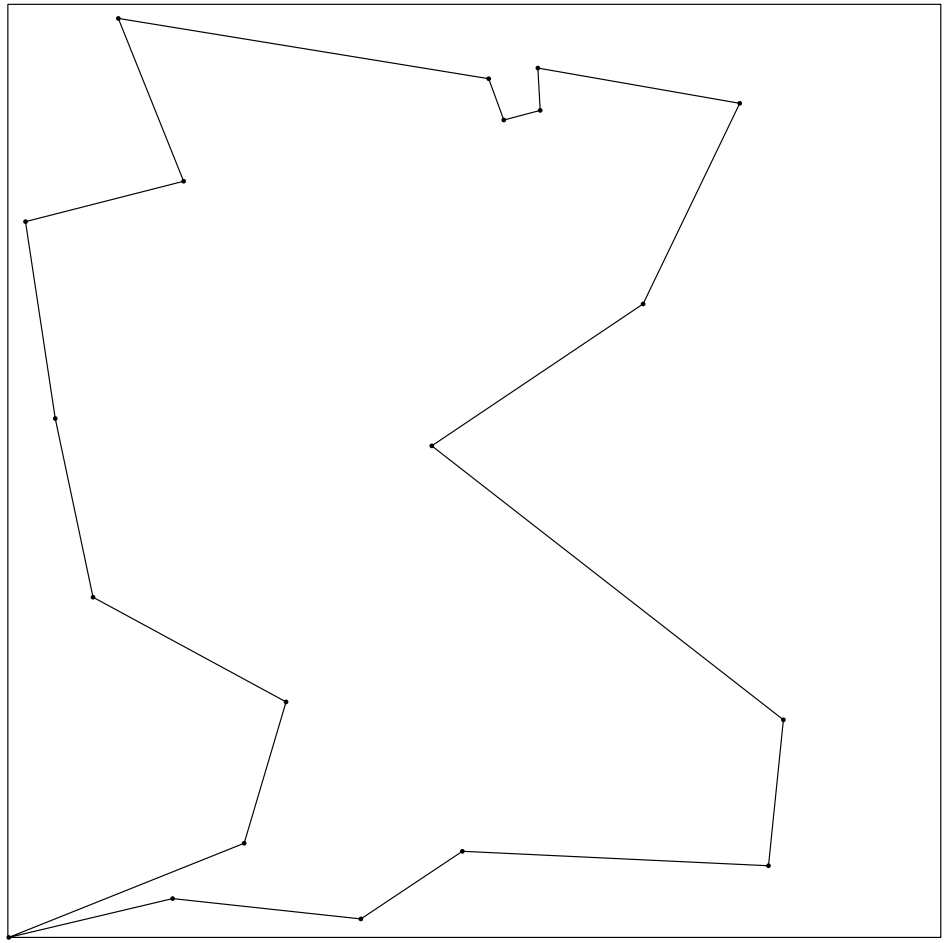
The existence of a successful computation of a Turingmachine can be reduced to the existence of a satisfying assignment for a circuit.

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.

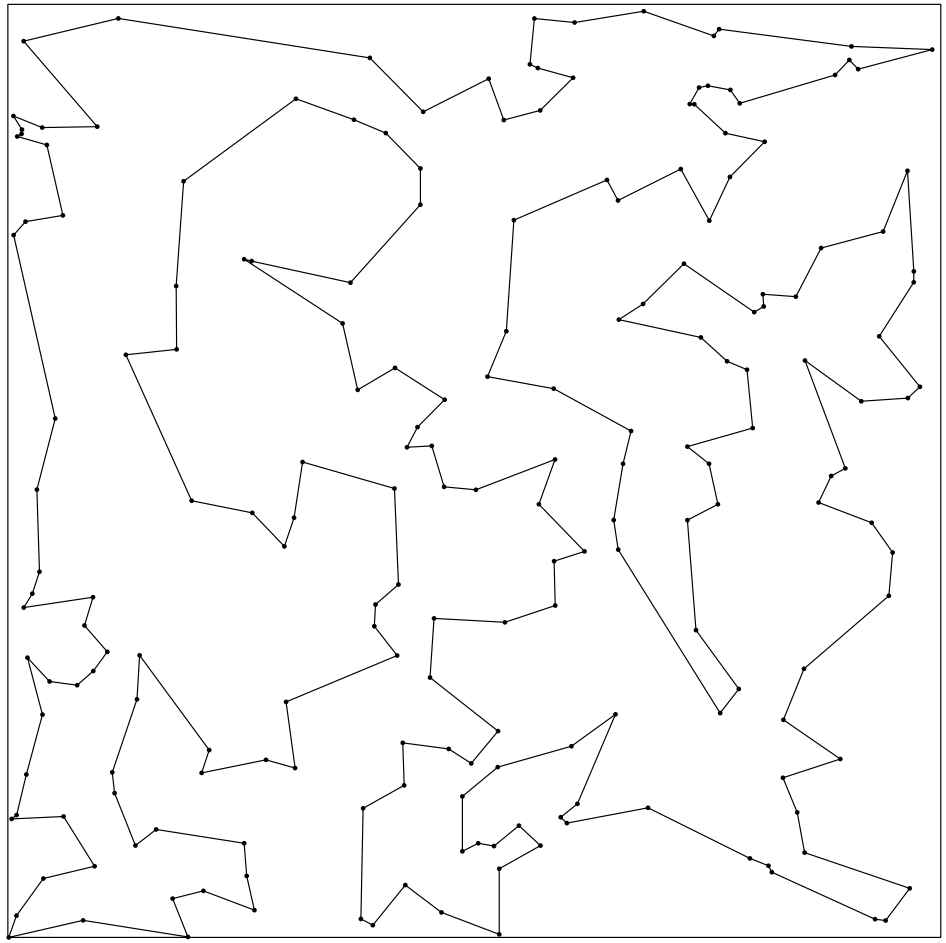
But most formulas are not constructed in this way!

Example: TSP



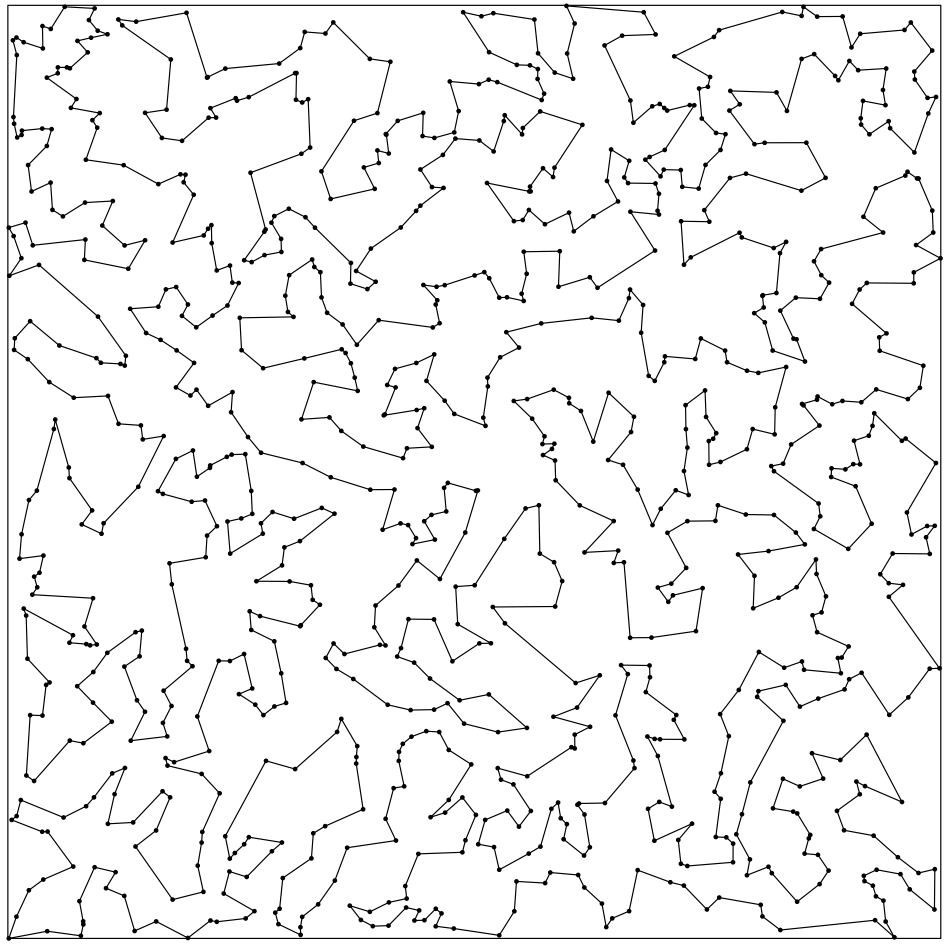
51.050611

Example: TSP



435.489475

Example: TSP



2107.739054

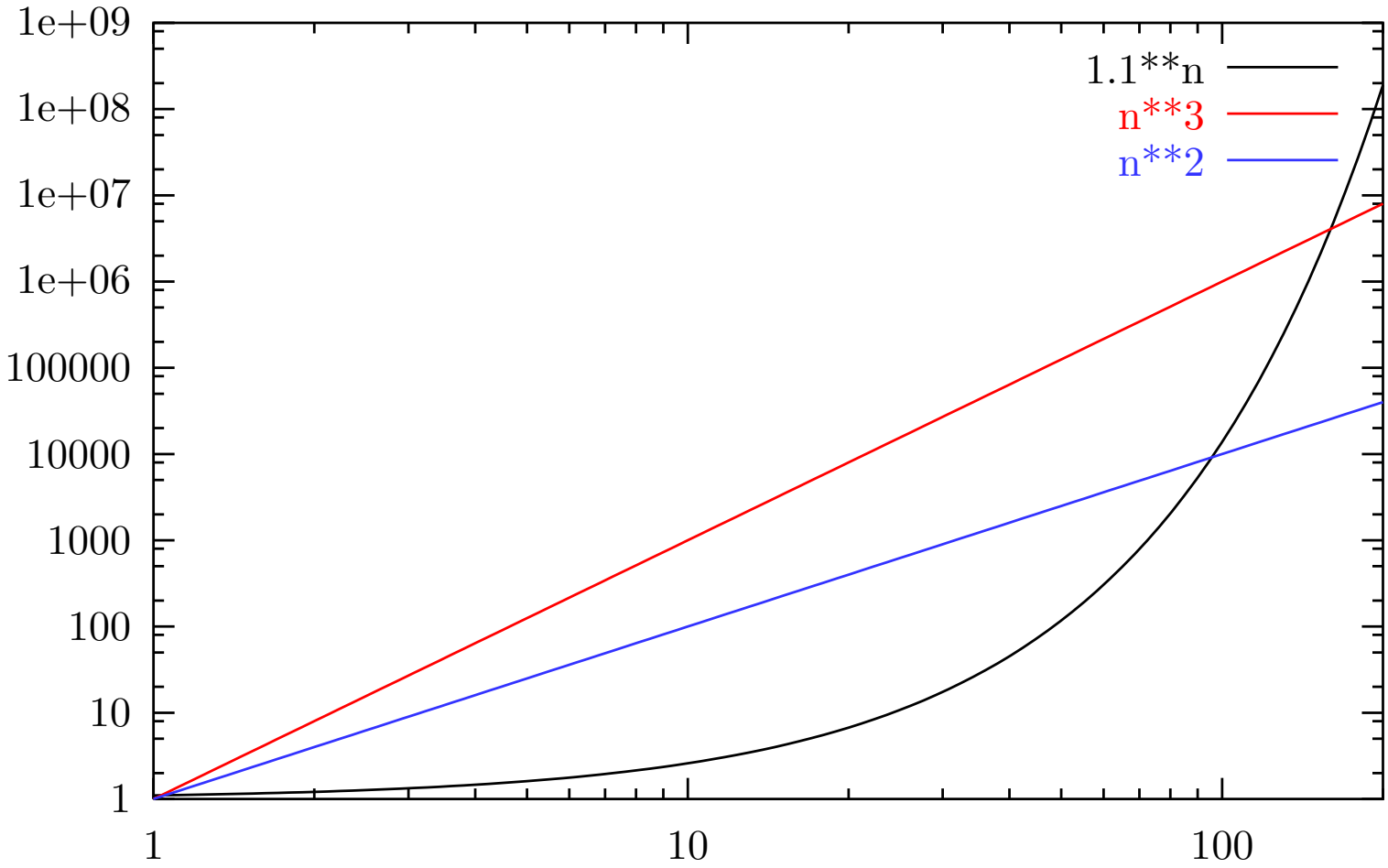
Running Times

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

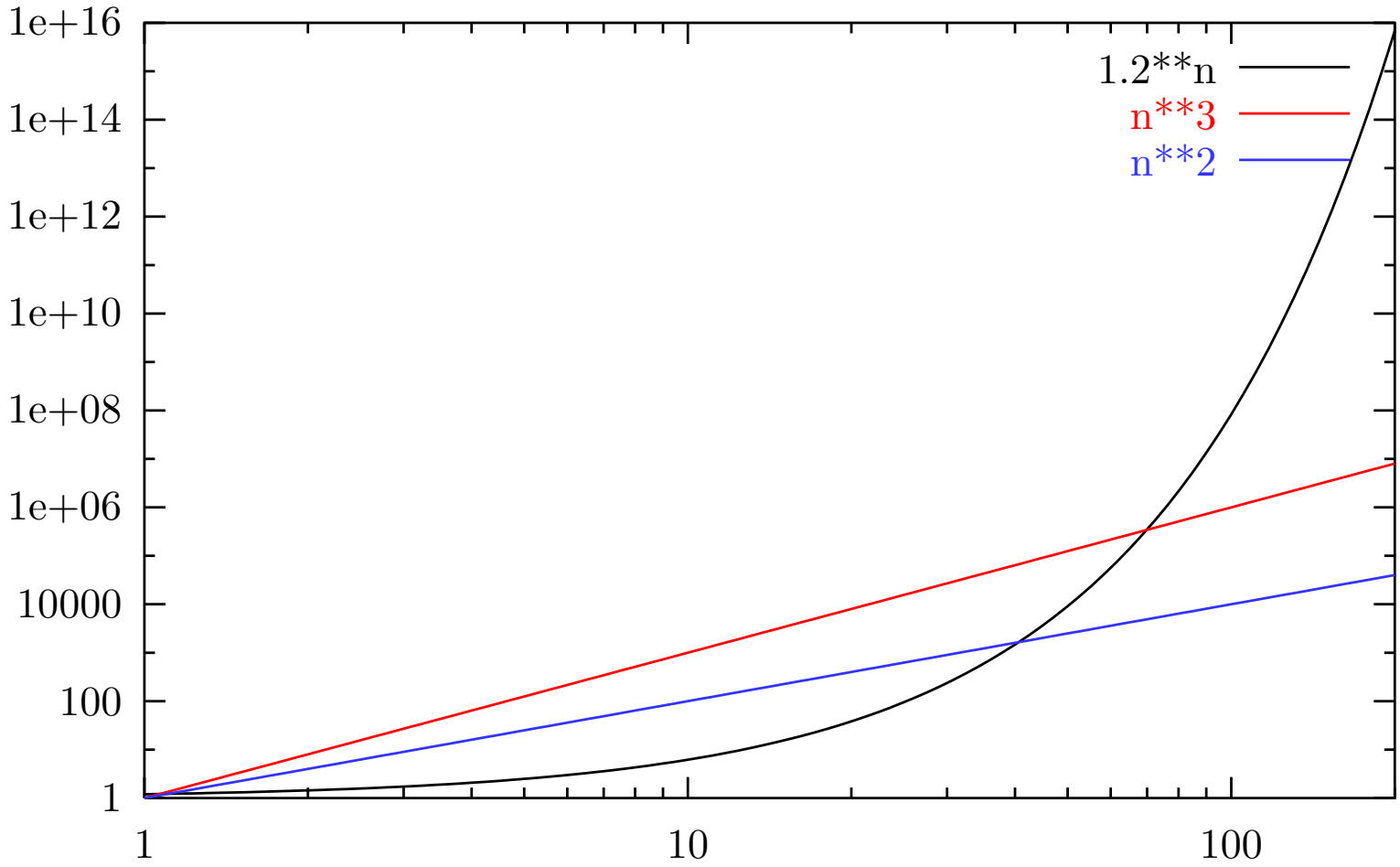
- Greedy-Algorithmen
- Divide-and-Conquer
- Dynamic Programming

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

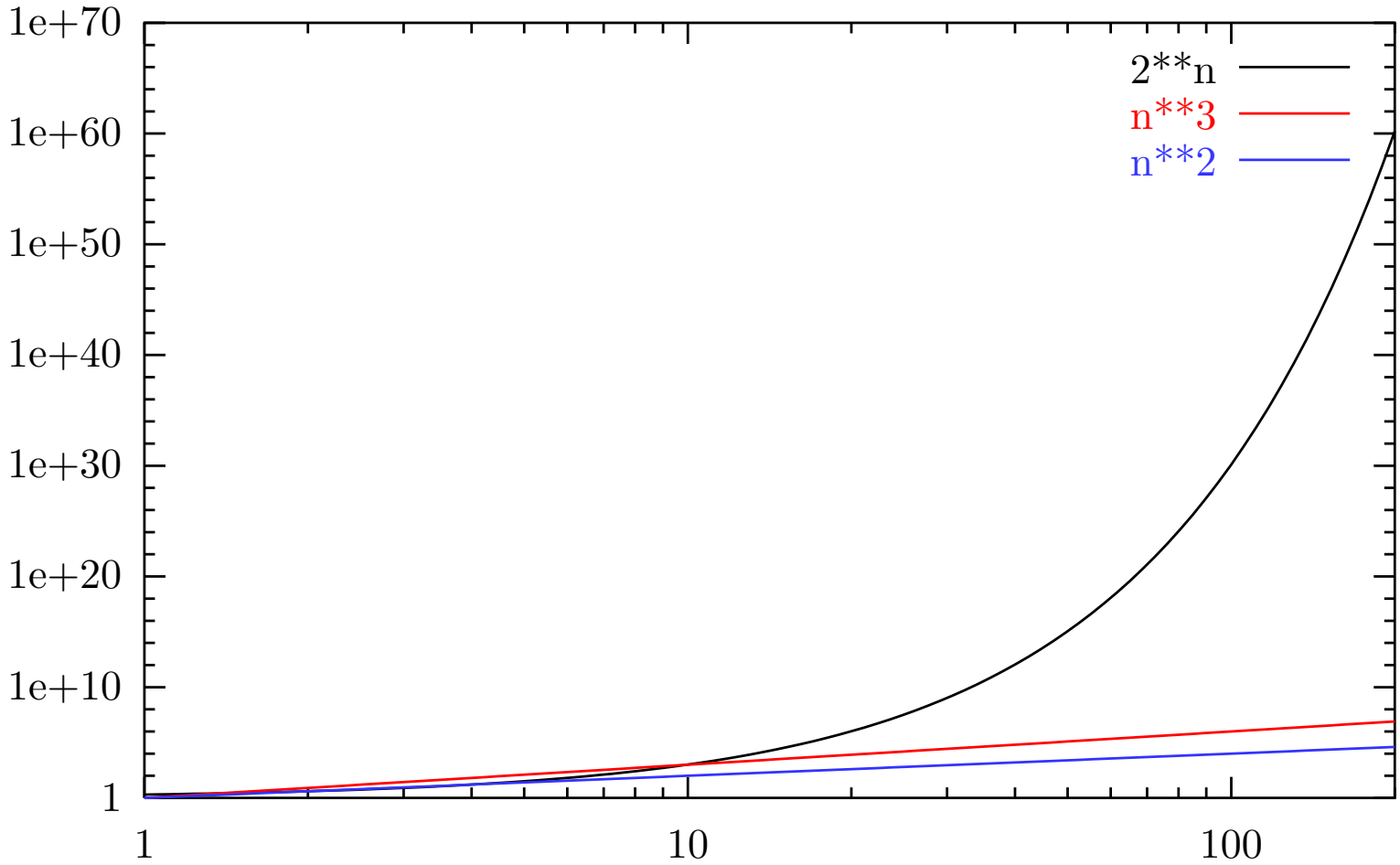
Comparing Running Times



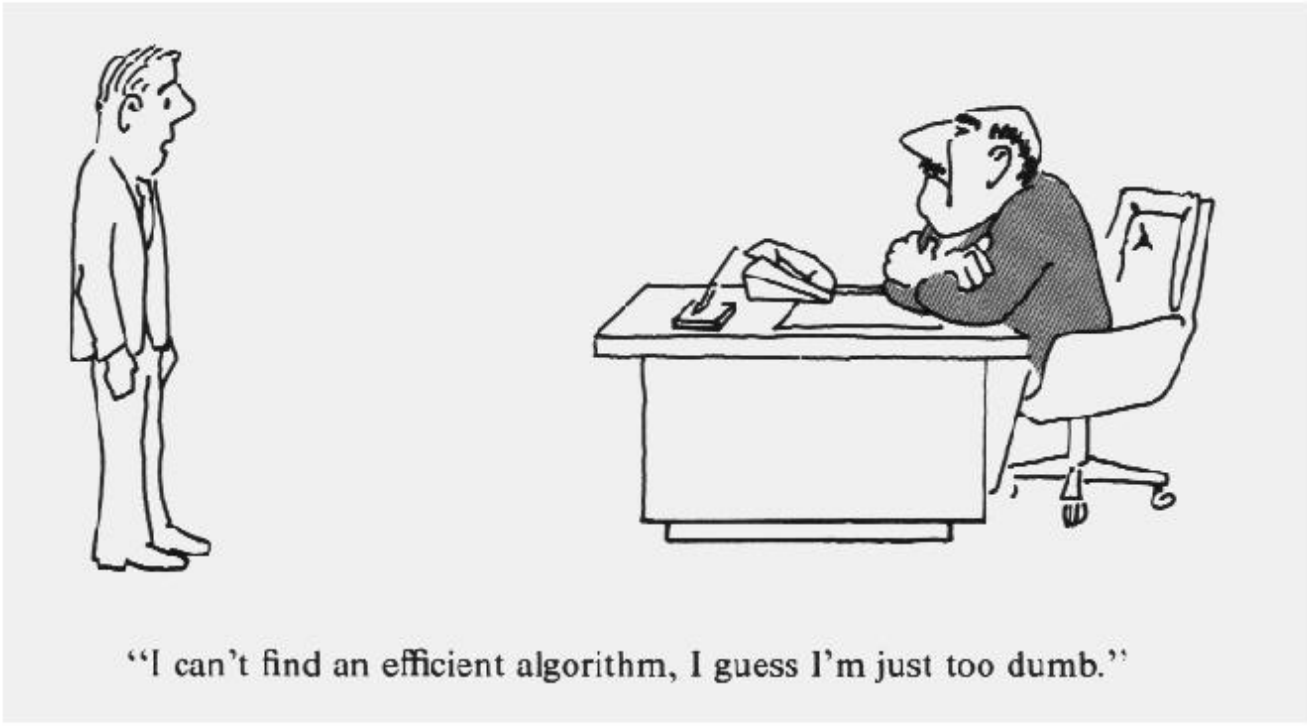
Comparing Running Times



Comparing Running Times



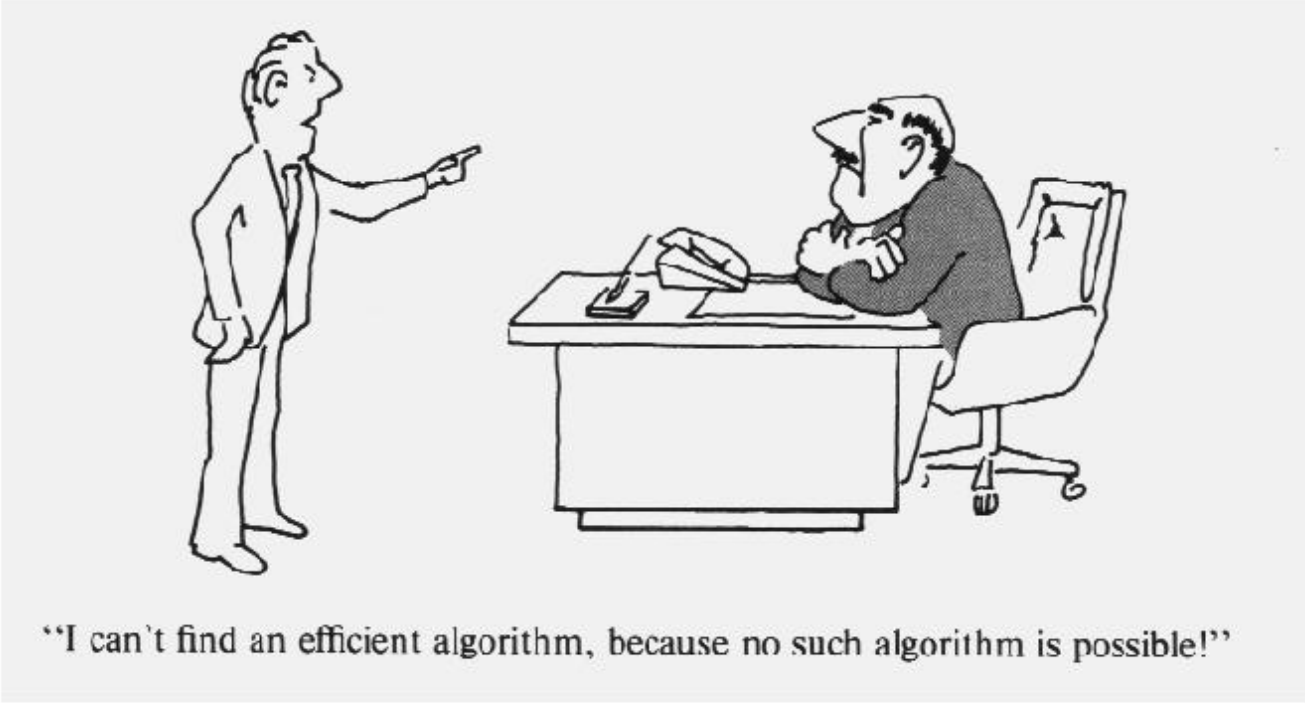
NP-Completeness as an Excuse



"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

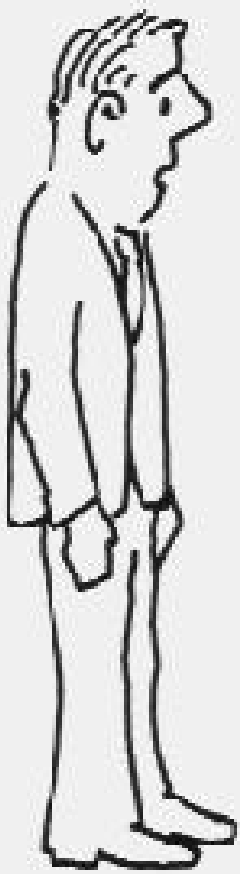
"I can't find an efficient algorithm, but neither can all these famous people."

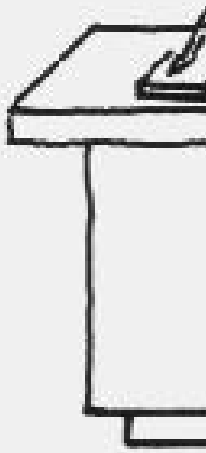
Garey and Johnson. Computers and Intractability.

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!*





"I can't find an efficient algorithm, I

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.

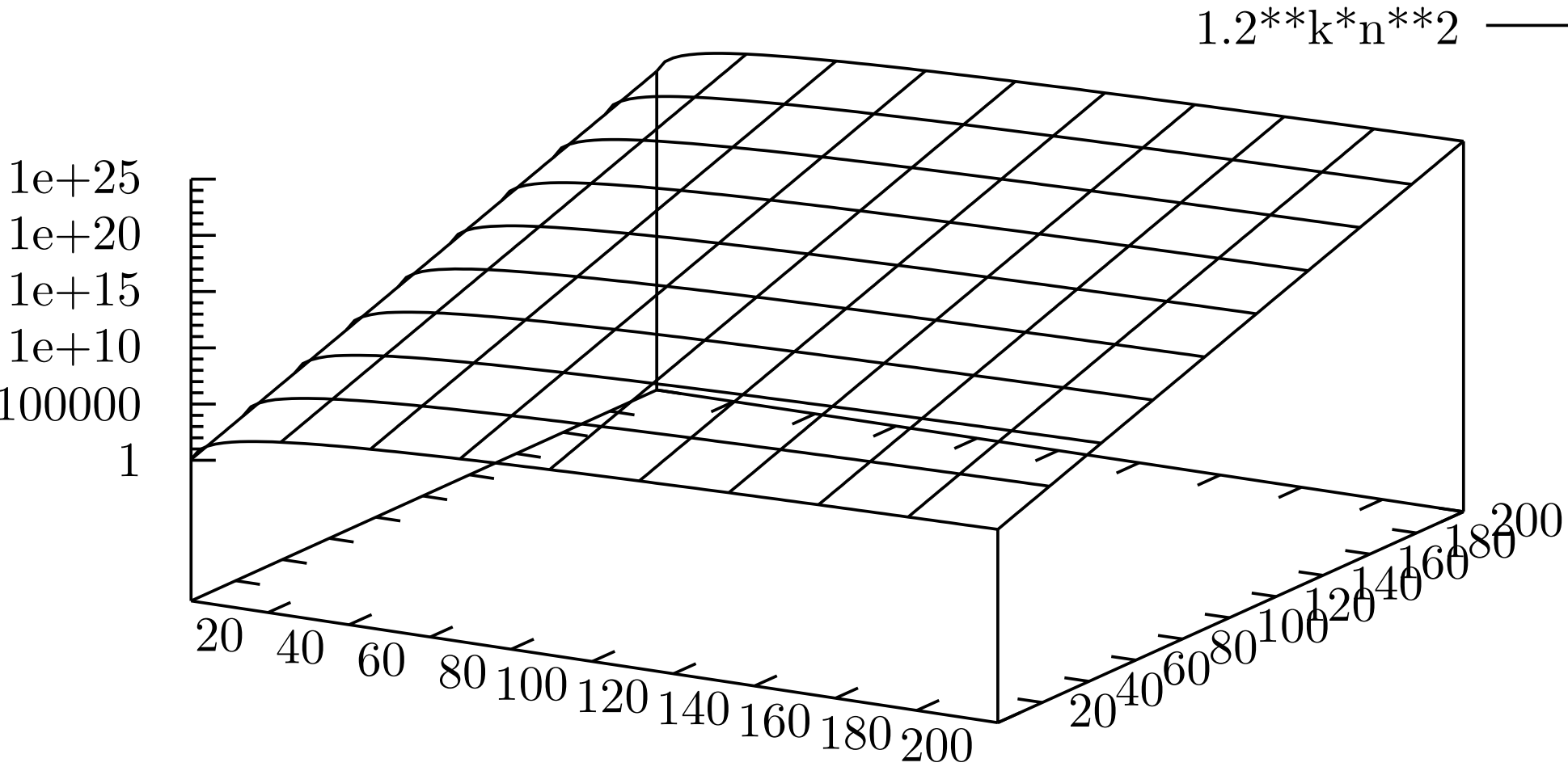
Let n be the size of some instance and k the corresponding parameter.

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

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

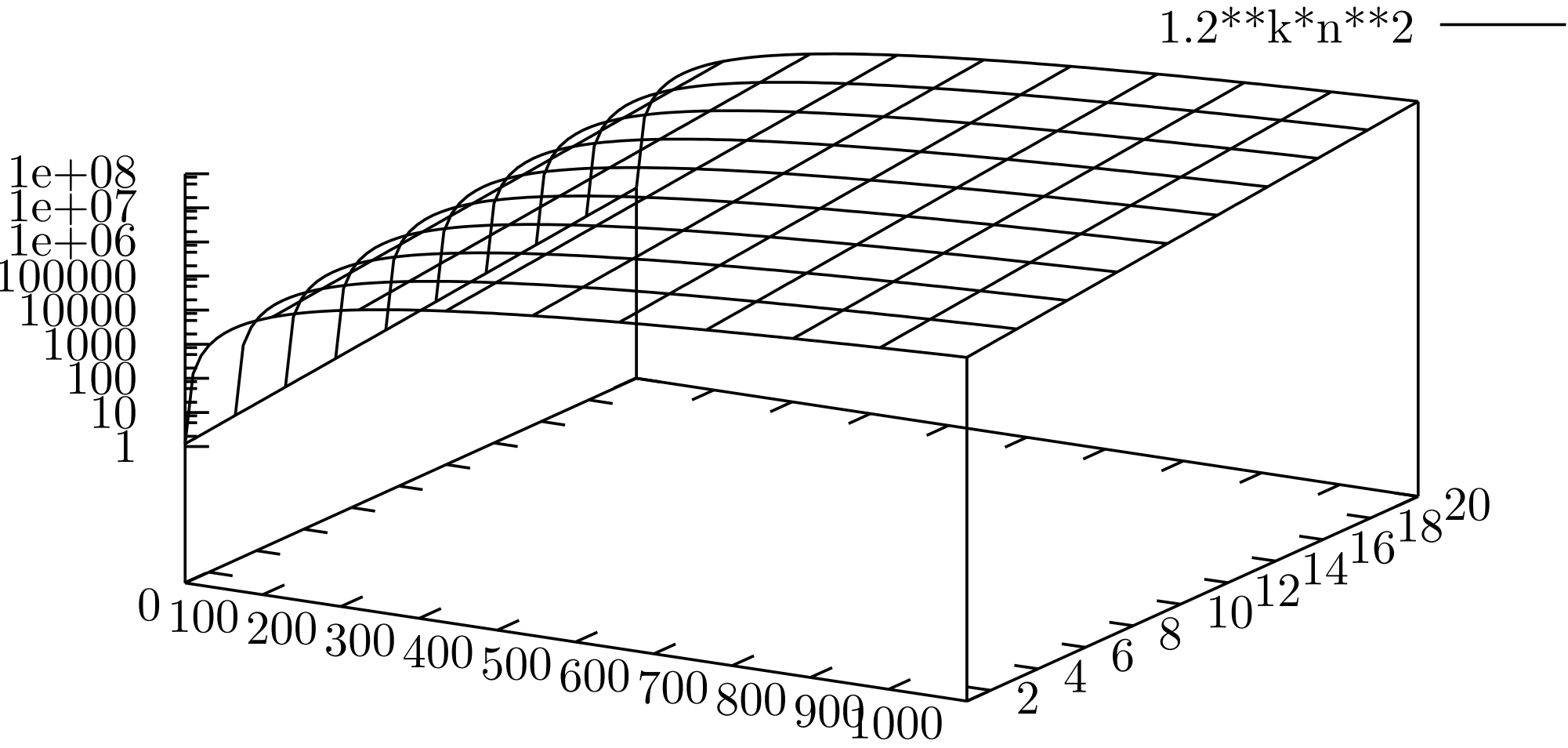
Here c is a constant and f an arbitrary function.

Running Time of a Parameterized Algorithm



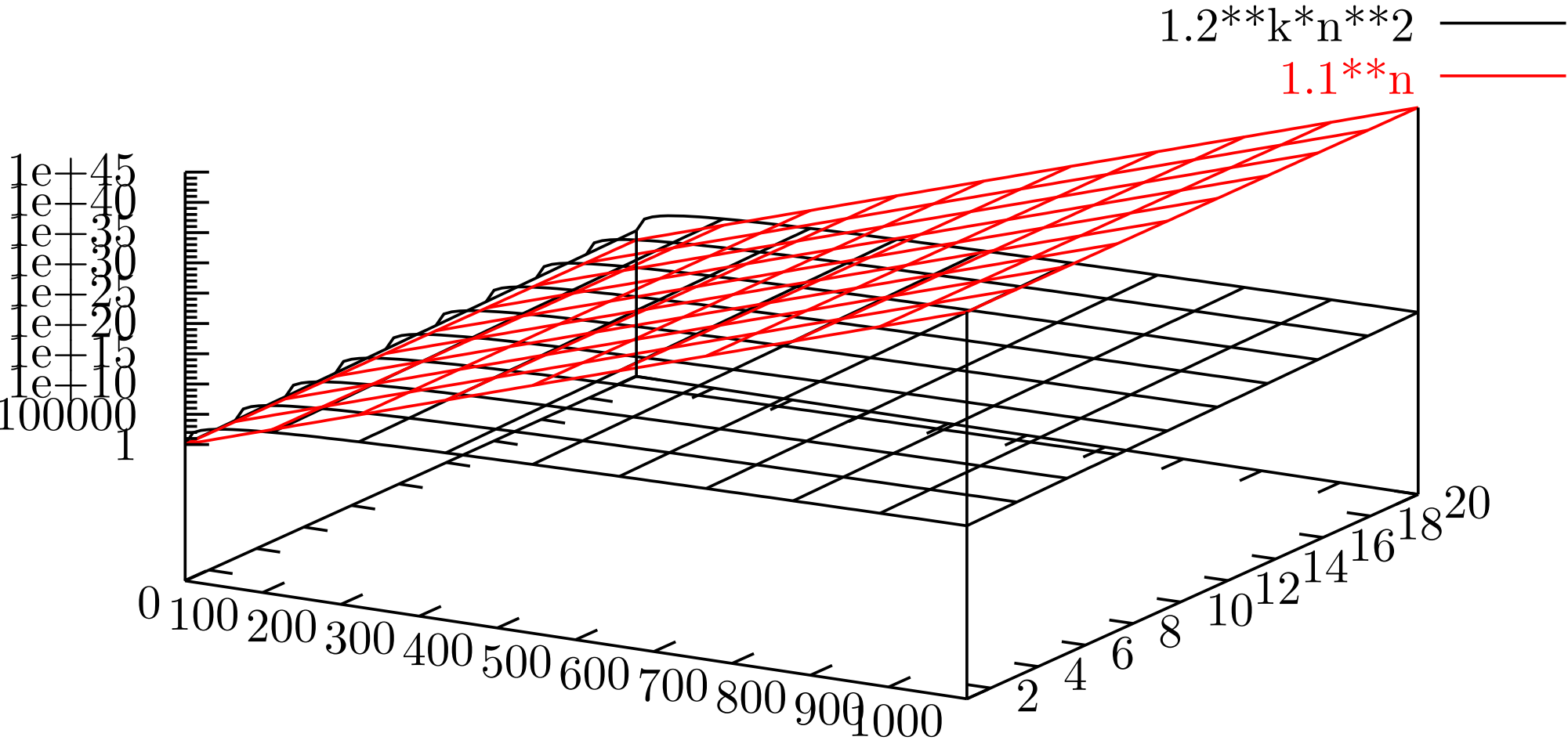
The running time is $1.2^k n^2$. The parameter is between 1 and n.

Running Time of a Parameterized Algorithm



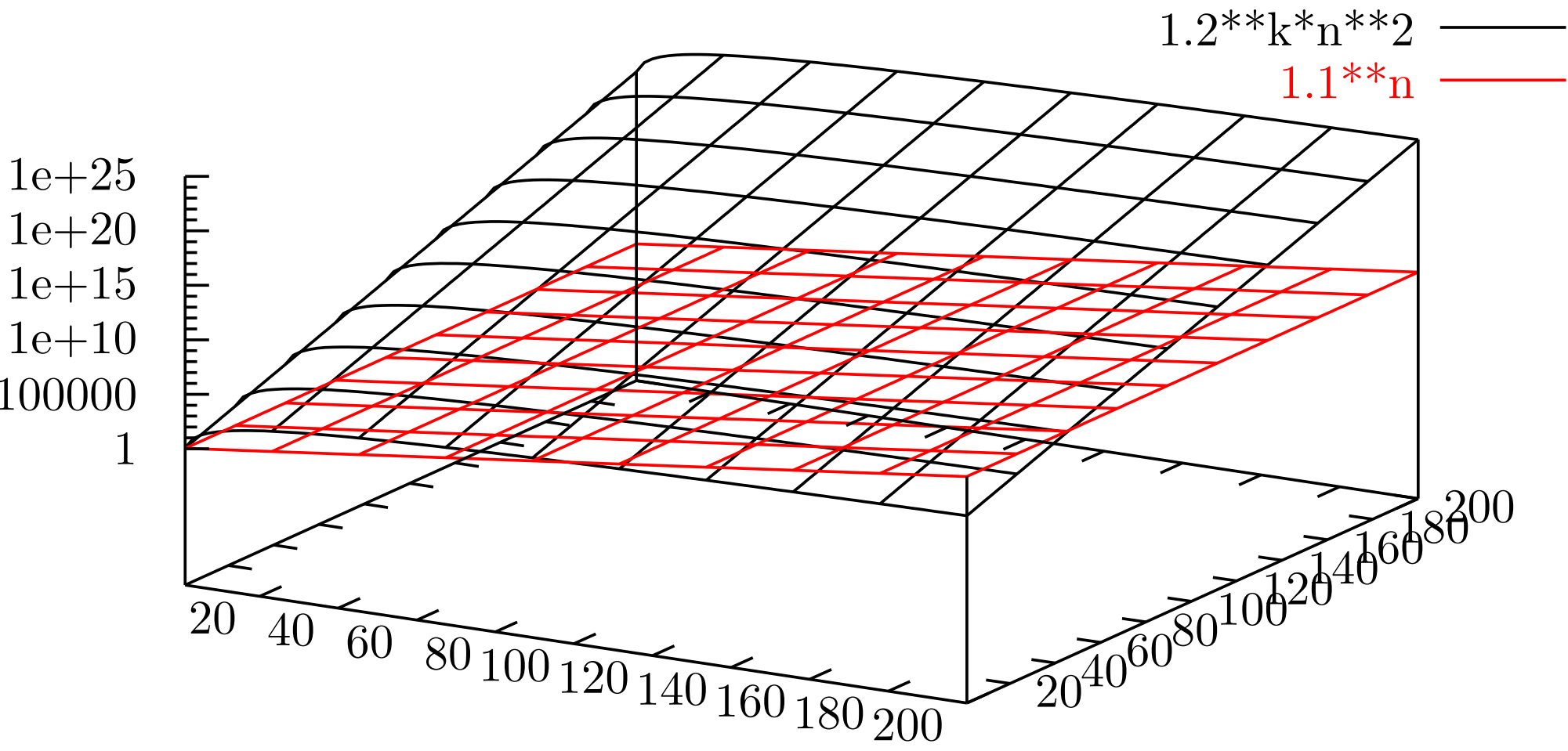
The running time is $1.2^k n^2$. The parameter is small.

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

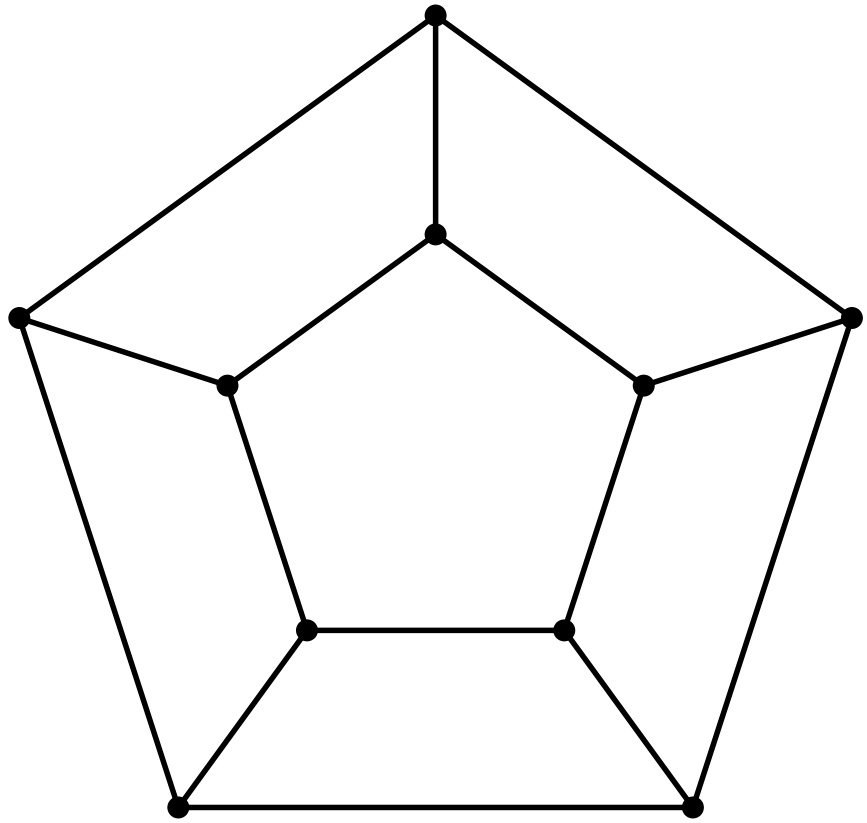
Input: A graph $G = (V, E)$.

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

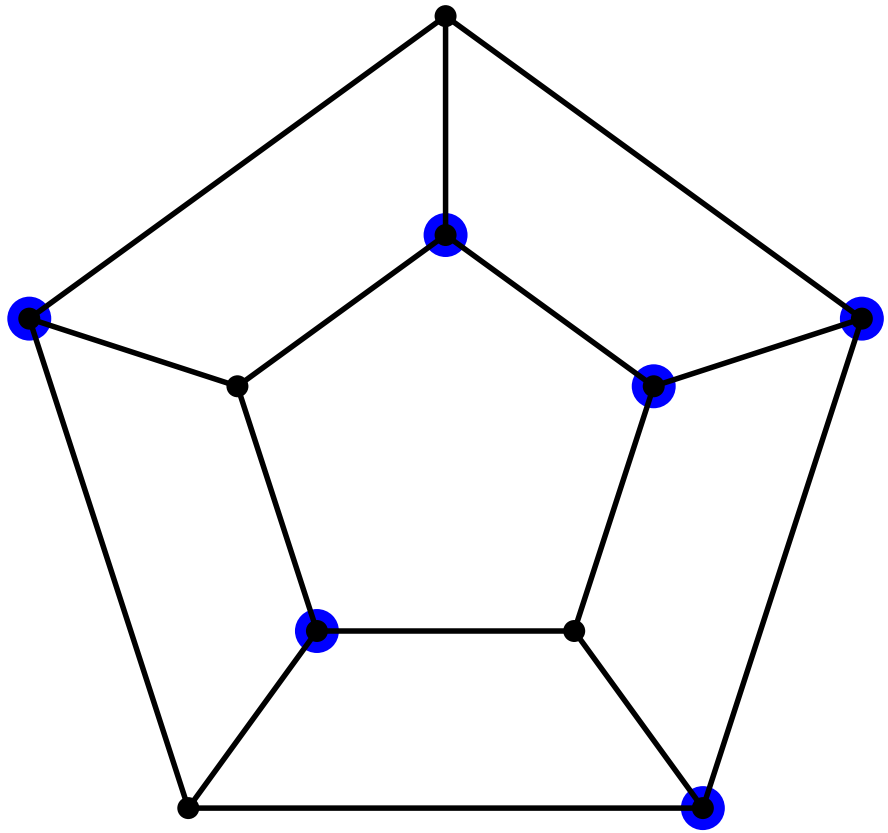
Definition

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.

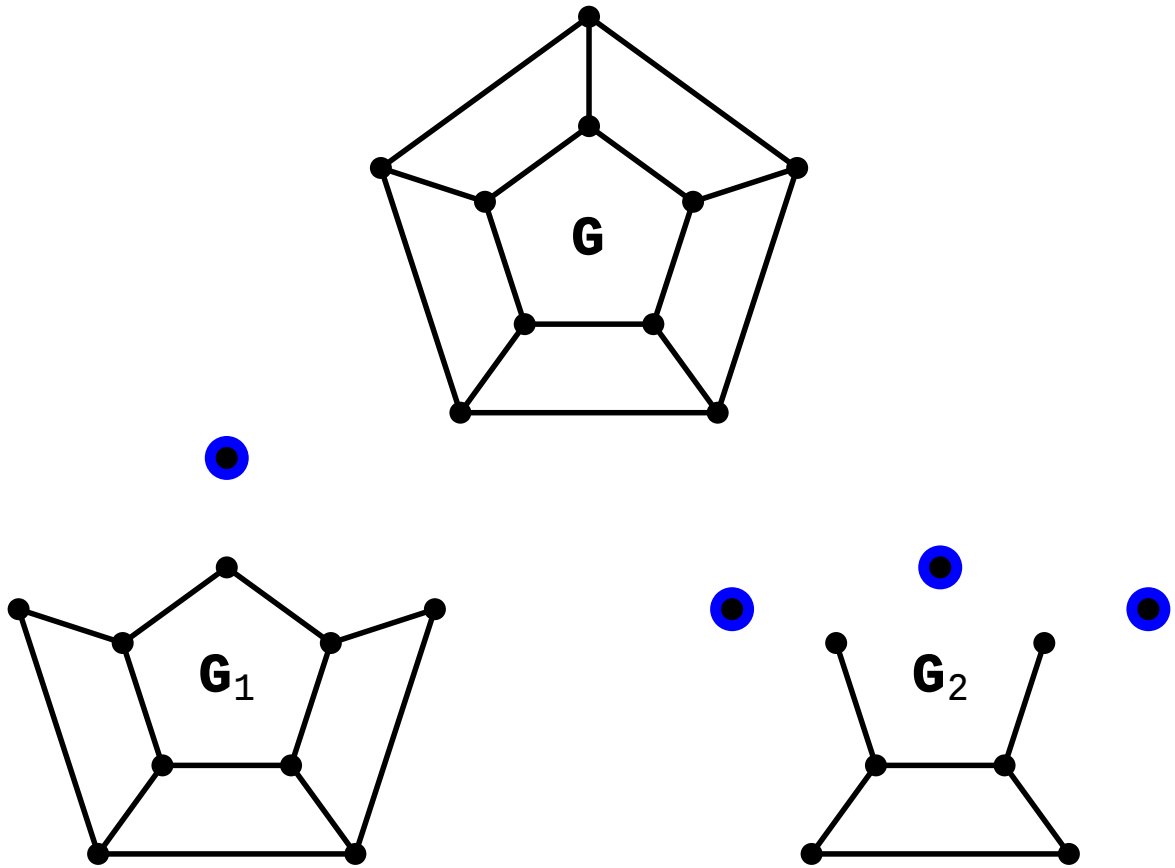
Backtracking

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)

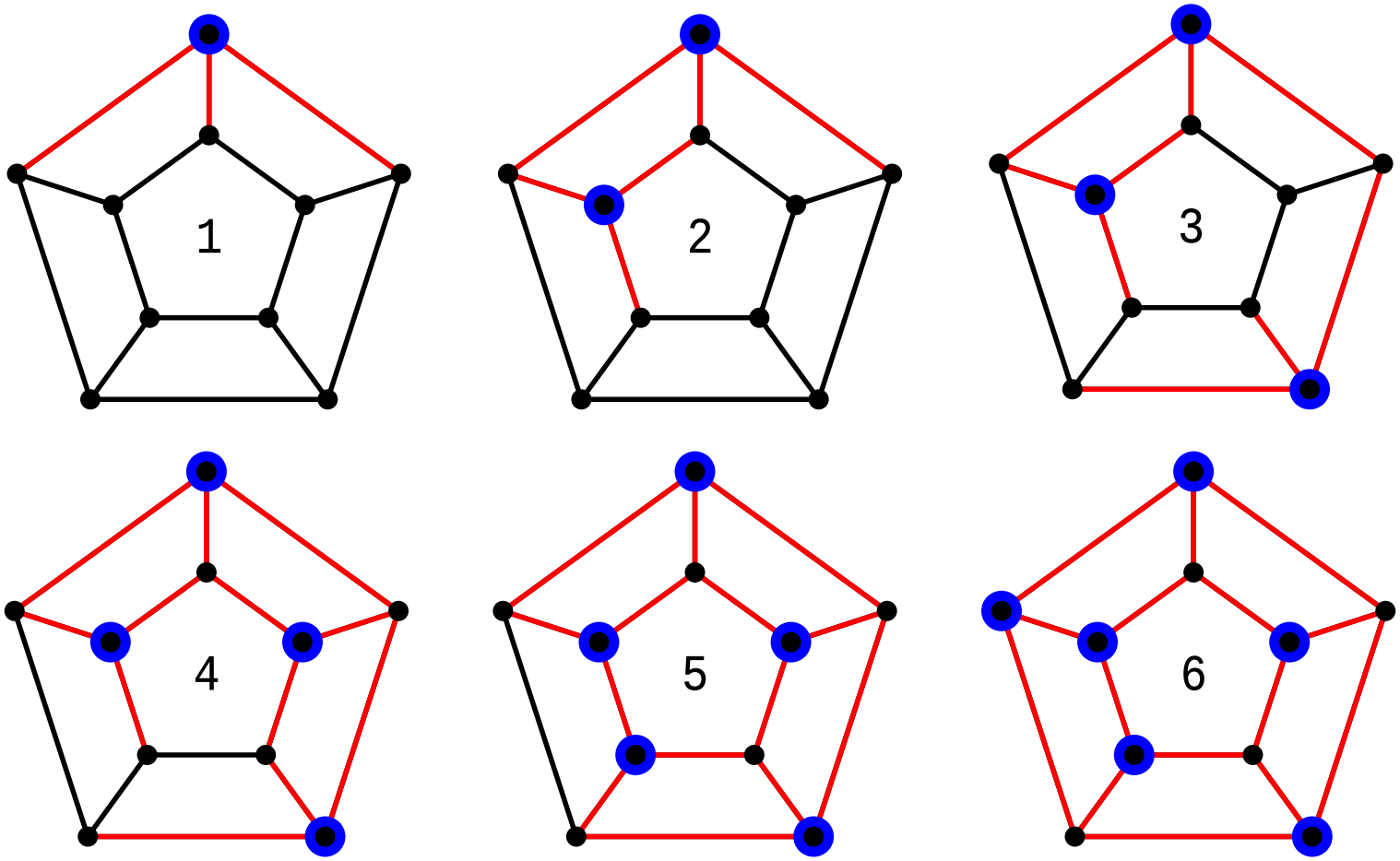
Input: $G = (V, E)$

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

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

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 retur
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$(s', l - 1)$. In 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

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

- 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.

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 answered only recently.)

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*

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

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.