Computational complexity

lecture 11

Derandomization

End of previous lecture: we have presented two „practical” methods of derandomization:

1) <u>The method of conditional expected values</u>
(we invariably ensure that the expected value of a good result is high)

2) <u>The method of pairwise-independent variables</u>
(out of $log(n)$ random bits we produce n pseudorandom bits, for which our algorithm performs as for completely random bits)

Derandomization

End of previous lecture: we have presented two „practical” methods of derandomization:

1) The method of conditional expected values
   (we invariably ensure that the expected value of a good result is high)

2) The method of pairwise-independent variables
   (out of $log(n)$ random bits we produce n pseudorandom bits, for which our algorithm performs as for completely random bits)

Now more theoretically: how one can try to derandomize an arbitrary randomized algorithm.

- We want to generalize the second method − having only $log(n)$ random bits, we want to generate n pseudorandom bits, such that no polynomial algorithm can distinguish them from completely random bits

Derandomization

Having only $log(n)$ random bits, we want to generate n pseudo-random bits, such that no polynomial algorithm can distinguish them from completely random bits.

More precisely:

- a generator – a function $G:\{0,1\}^{log(n)} \to \{0,1\}^n$ computable in time polynomial in n
- a generator is ε-pseudorandom, if for every family of functions $D:\{0,1\}^n \to \{0,1\}$ from the class **P/poly** we have the property that for every large enough n:

$$Pr_{x \in \{0,1\}^{log(n)}}[D(G(x))=1]-Pr_{y \in \{0,1\}^n}[D(y)=1] \leq \varepsilon$$

Theorem (Yao 1982)

If a (1/10)-pseudorandom generator exists, then **BPP**=**P**.

Derandomization

<u>Theorem</u> (Yao 1982)

If a (1/10)-pseudorandom generator exists, then **BPP**=**P**.

<u>Proof</u>

Take a machine M, which **BPP**-recognizes some language L in time $p(n)$. Given an input word w of length n:

- generate, consecutively, all sequences of bits x of length $log(p(n))$
- for each of them compute $G(x)$
- simulate M on the word w with bits $G(x)$
- accept if at least half of computations have accepted

Derandomization

<u>Theorem</u> (Yao 1982)

If a (1/10)-pseudorandom generator exists, then **BPP**=**P**.

<u>Proof</u>

Take a machine M, which **BPP**-recognizes some language L in time $p(n)$. Given an input word w of length n:

- generate, consecutively, all sequences of bits x of length $log(p(n))$
- for each of them compute $G(x)$
- simulate M on the word w with bits $G(x)$
- accept if at least half of computations have accepted

<u>Correctness proof:</u>

The amount of “yes” results equals: $a_w = Pr_{x\in\{0,1\}^{log(p(n))}}[M(w,G(x))=1]$

For a random algorithm this is: $b_w = Pr_{y\in\{0,1\}^{p(n)}}[M(w,y)=1]$

If for every (long enough) w it holds that $|a_w - b_w| \leq 1/10$, then OK:

for $w \in L$ it is $b_w \geq 3/4$, i.e., $a_w > 1/2$, similarly for $w \notin L$

(a finite number of short inputs can be handled “manually”)

Derandomization

<u>Correctness proof:</u>

The amount of "yes" results equals: $a_w = Pr_{x \in \{0,1\}^{\log(p(n))}}[M(w,G(x))=1]$

For a random algorithm this is: $b_w = Pr_{y \in \{0,1\}^{p(n)}}[M(w,y)=1]$

If for every (long enough) w it holds that $|a_w - b_w| \leq 1/10$, then OK:

for $w \in L$ it is $b_w \geq 3/4$, i.e., $a_w > 1/2$, similarly for $w \notin L$

(a finite number of short inputs can be handled "manually")

If the difference is $\geq 1/10$ for arbitrarily long words w_n, then we consider $D(y)=M(w_n, y)$ (it is in **P/poly**) and we obtain a contradiction with the definition of a pseudorandom generator – for every (large enough) n it should hold that:

$$Pr_{x \in \{0,1\}^{\log(n)}}[D(G(x))=1] - Pr_{y \in \{0,1\}^n}[D(y)=1] \leq 1/10$$

Derandomization

Theorem (Yao 1982)

If a (1/10)-pseudorandom generator exists, then **BPP**=**P**.

Formerly, people supposed that such generators rather do not exist.

But in the course of time, it turned out that they exist under weaker and weaker assumptions. The strongest result of this form is:

Theorem (Impagliazzo-Wigderson 1998)

If SAT cannot be solved by a (not necessarily uniform) family of circuits of size smaller than $2^{\varepsilon n}$, then **BPP**=**P**

The proof is difficult.

Notice an interesting phenomenon: out of hardness of one problem (SAT) it is possible to deduce that other problems (those from **BPP**) are easy.

Fixed-parameter tractability

Idea:

- sometimes the reason for hardness lies not in the length of the input, but in some its parameter
- while fixing the parameter, we can sometimes obtain a polynomial algorithm
- sometimes even the exponent in this polynomial does not depend on the parameter

Such problems are called FPT (fixed-parameter tractable)

More formally:

- a *parameter* – a function from input words to natural numbers
- a problem is *fixed-parameter tractable* with respect to a para-meter k, if it has complexity $f(k)\cdot n^c$ (important: the exponent does not depend on k)

Fixed-parameter tractability

Formally:

- a *parameter* – a function from input words to natural numbers
- a problem is *fixed-parameter tractable* with respect to a parameter k, if it has complexity $f(k) \cdot n^c$ (important: the exponent does not depend on k)

<u>Example</u>:

SAT is FPT with respect to the number of variables k

algorithm: check all valuations

complexity: $2^k n$

Fixed-parameter tractability

Example 2:

VERTEX-COVER is FPT with respect to the size of the maximal cover.

- Algorithm: A full backtracking – for every edge (not having any of its ends in the cover yet), consecutively, decide which of its ends should be taken to the cover.
- This backtracking has depth k; in each step we choose one of two nodes, so the complexity is $2^k n$.
- Attention: One can consider another backtracking (a more natural one) – for every node decide whether it should be taken to the cover or not. This is not an FPT algorithm: it considers all k-element sets of nodes, so k goes to the exponent here.

Fixed-parameter tractability

Example 3:

k-colorability of a graph is not FPT with respect to the number of colors (unless **P=NP**).

If it was FPT, then it would be possible to solve the **NP**-complete problem of 3-colorability in time $f(3)\cdot n^c$, i.e., in polynomial time

But there is another parameter, under which k-colorability of a graph is FPT.

This is *treewidth.*

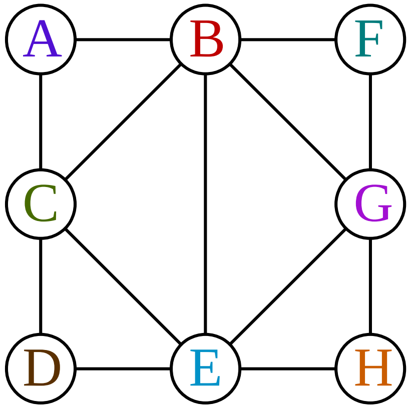
Treewidth

Intuitively: treewidth is small when the graph is similar to a tree.

Formally: we consider a tree decomposition.

A *tree decomposition* of a graph (V,E) consists of a tree T, in which nodes (called *bags*) are labeled by subsets of V, such that:

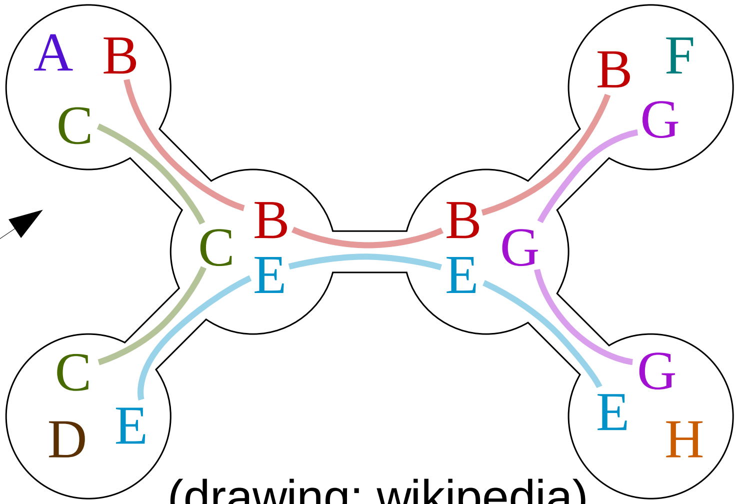
- for every $v \in V$, the bags of T to which v belongs form a connected (and nonempty) subtree
- for every edge $(u,v) \in E$ there is a bag X in T such that $u,v \in X$



The decomposition is not unique.

Treewidth = maximal size of a bag in a decomposition, minus 1.

Treewidth = 2



(drawing: wikipedia)

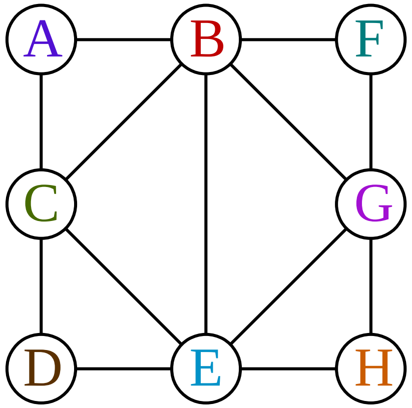
Treewidth

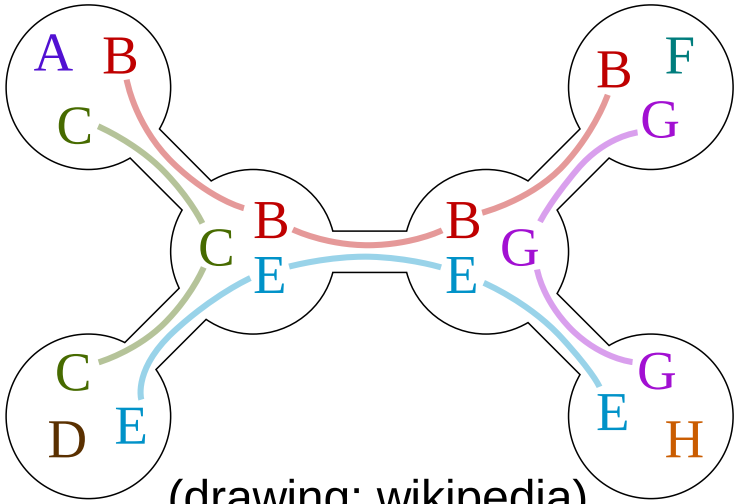
k-colorability of a graph having treewidth s

Algorithm: dynamic programming over the tree decomposition. We compute which k-colorings of a bag in the decomposition (out of k^{s+1} possibilities) can be extended to a k-coloring of the whole subtree. We ensure that:

- if some node belongs to neighboring bags, then it should have the same color in both bags (this is enough, since the subtree containing a node is connected), and
- if two nodes in a bag are neighbors, then they should have different colors (this is enough, since every edge "belongs" to some bag).

Running time: $f(s) \cdot n$





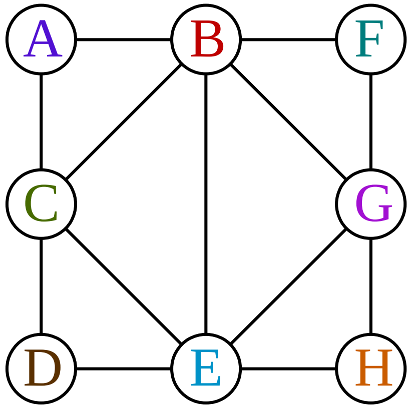
(drawing: wikipedia)

Treewidth

k-colorability of a graph having treewidth s

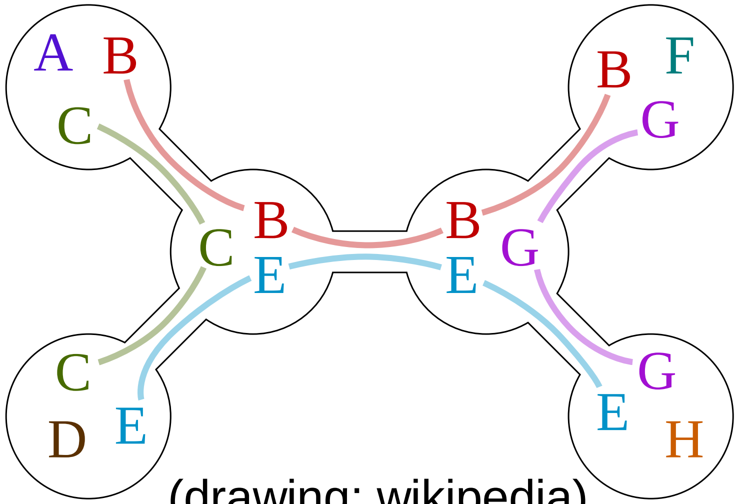
Algorithm: dynamic programming over the tree decomposition. We compute which k-colorings of a bag in the decomposition (out of k^{s+1} possibilities) can be extended to a k-coloring of the whole subtree.

Running time: $f(s) \cdot n$



Additional problem: one has to find an optimal tree decomposition, before running this algorithm

- computing the treewidth is **NP**-hard
- but for a fixed treewidth s, a decompo-sition can be found in time $f(s) \cdot n$
- a decomposition of width slightly worse than the optimal one can be found in polynomial time



(drawing: wikipedia)

Treewidth

A dynamic programming over the tree decomposition of a graph, like for k-colorability, can be applied to many other problems.

There is even a general theorem (“metatheorem”):

Theorem (Courcelle 1990):

Every property of graphs expressible in the MSO logic can be decided in time $f(s)\cdot n$, where s is the treewidth.

Treewidth

A dynamic programming over the tree decomposition of a graph, like for k-colorability, can be applied to many other problems.
There is even a general theorem ("metatheorem"):

Theorem (Courcelle 1990):

Every property of graphs expressible in the MSO logic can be decided in time $f(s)\cdot n$, where s is the treewidth.

- in this logic, we allow quantification over sets of nodes, and over sets of edges
- for most properties, it is easy to express them in this logic
- in this way we easily obtain an FPT algorithm, but the function $f(s)$ is fast-growing. In order to obtain a practical algorithm, it is necessary to concentrate on a particular problem
- another version of this theorem: the MSO logic without quantifi-cation over sets of edges (i.e., a weaker logic), but instead of treewidth we have "cliquewidth" (which can be smaller)

Treewidth

An example application of treewidth:

Graphs of control flow in structural programs (without GOTO) have treewidth at most 6 (Thorup 1998)

This helps e.g. in an optimal allocation of registers during compilation of programs

Fixed-parameter tractability

More on hardness:

k-colorability of a graph is not FPT with respect to the number of colors (unless **P**=**NP**).
If it was FPT, then it would be possible to solve the **NP**-complete problem of 3-colorability in time $f(3)\cdot n^c$, i.e., in polynomial time

This example is very unusual. There are many problems, which for a fixed value of a parameter are polynomial, but are not FTP (i.e., they can be solved in time $n^{f(k)}$, but not in $f(k)\cdot n^c$)

There exist multiple reductions between such problems (similarly to reductions between **NP**-complete problems), thus either all are are FPT, or none of them (we rather believe that none of them).
In this context, it makes sense to consider only reductions which do not change the value of the parameter.

In this context, one considers classes called **W**[k] for all $k\geq 0$ (we skip a definition)
W[0] contains FPT problems
it is assumed that problems from **W**[1] are not FPT
problems from **W**[2] are even harder (can be reduced to **W**[1], but not vice-versa), etc.

Approximation

- Approximation is considered for hard optimization problems: find the smallest vertex cover, the greatest clique, etc.
- One often says that these problems are e.g. **NP**-complete. What does it mean precisely? These are not decision problems. But optimization problems can be considered in a decision variant, and then they can be **NP**-complete, e.g.
  - ➔ for a given graph, and a number k, is there a clique of size k?
  - ➔ or: for a given graph, a number k, and a set of nodes, is there a clique of size k containing these nodes?
    (this corresponds to searching for the clique, not only its size)
- The problem in the optimization version can be easily reduced to many calls of the problem in the decision version.

Approximation

- Approximation: looking for approximate solutions for (hard) optimization problems
  - ➔ We want an algorithm that outputs some solution, which is maybe not optimal, but not much worse than the optimal solution

Approximation

- Approximation: looking for approximate solutions for (hard) optimization problems

- An approximation algorithm has error ε when:
  - ➔ for maximization problems:

    $$(\text{optimum-solution})/\text{optimum} \le \varepsilon$$

    that is: $\text{solution} \ge \text{optimum}\cdot(1-\varepsilon)$
  - ➔ for minimization problems:

    $$(\text{solution-optimum})/\text{solution} \le \varepsilon$$

    that is: $\text{solution} \le \text{optimum}/(1-\varepsilon)$

Approximation

- Approximation: looking for approximate solutions for (hard) optimization problems
- An approximation algorithm has error ε when:
  - ➔ for maximization problems:

    $$(\text{optimum-solution})/\text{optimum} \leq \varepsilon$$

    that is: $\text{solution} \geq \text{optimum}\cdot(1-\varepsilon)$
  - ➔ for minimization problems:

    $$(\text{solution-optimum})/\text{solution} \leq \varepsilon$$

    that is: $\text{solution} \leq \text{optimum}/(1-\varepsilon)$
- The infimum of all ε for which there is a polynomial algorithm, is called *approximation factor*
  - ➔ factor = 1 – effective approximation impossible
  - ➔ factor = 0 – we say that there exists a PTAS (polynomial time approximation scheme); if the exponent in the algorithm does not depend on ε, we say that there exists a strong PTAS

Example: VERTEX-COVER

Vertex cover: a set of nodes containing at least one end of every edge (we want to find a smallest one)

A simple idea: in a loop – take a node of the maximal degree, take it to the cover, and remove it from the graph, together with all its neighbors.

Does it approximate well the minimal cover?

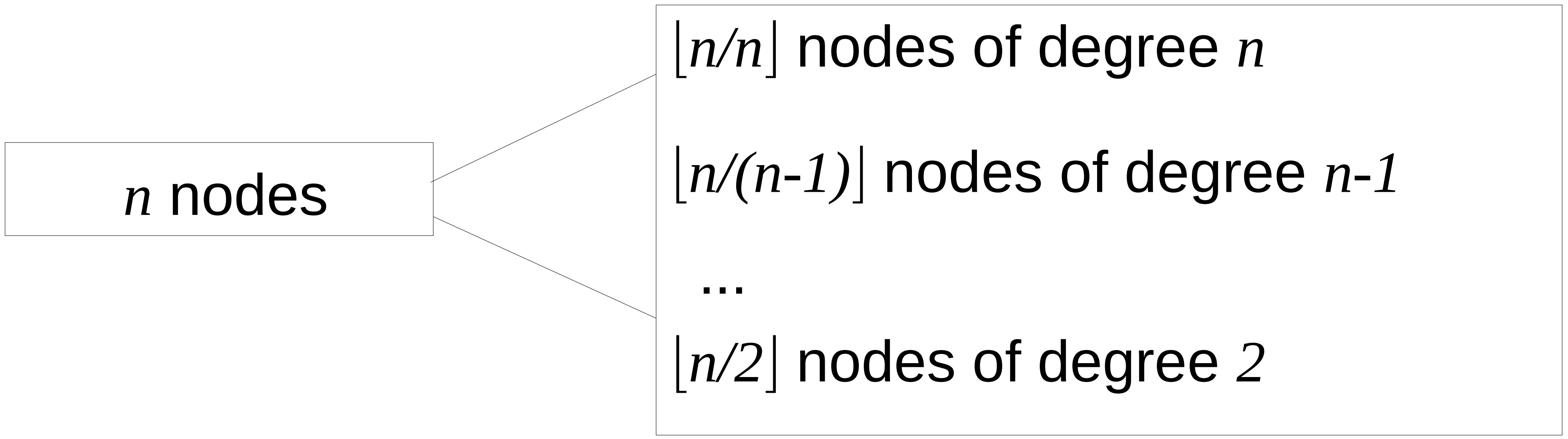
Example: VERTEX-COVER

Vertex cover: a set of nodes containing at least one end of every edge (we want to find a smallest one)

A simple idea: in a loop – take a node of the maximal degree, take it to the cover, and remove it from the graph, together with all its neighbors.

Does it approximate well the minimal cover?

The solution can be $log(n)$ times worse than the minimal one, e.g. for such a bipartite graph:


n nodes
$\lfloor n/n \rfloor$ nodes of degree n
$\lfloor n/(n-1) \rfloor$ nodes of degree n-1
...
$\lfloor n/2 \rfloor$ nodes of degree 2

the algorithm chooses all the nodes on the right, instead of nodes on the left

Example: VERTEX-COVER

But there is an approximation of VERTEX-COVER with factor 1/2:

In a loop – add both ends of some edge to the cover, and remove them from the graph, together with their neighbors

Every cover contains at least one of these ends, thus our cover is at most twice larger (error 1/2)

A known hypothesis: there is no better algorithm