Parameterized Algorithms

Dániel Marx
Roohani Sharma
Barış Can Esmer (tutorials)

Lecture #1
April 11, 2023
Modalities

- Lectures every Tuesday (10:15-12:00)
- Exercises sheets handed out every \( \approx 2 \) weeks, needed to be submitted in \( \approx 1 \) week
- Tutorials to discuss the exercises (dates to be discussed)
- At least 50% of all points on exercises needed to be admitted to exam.
- Oral exams
- Deadline for unregistering from the course: first oral exam
Prerequisites: algorithms

- Worst-case analysis: guaranteed running time $T(n)$ for every input of size $n$.
- Big-O notation: hiding constant factors + ignoring small inputs.
- Two main classes:
  - Polynomial time ($O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, ...)
    - **Example:** Quick Sort, Matrix multiplication, Perfect Matching
  - Exponential time ($2^n$, $2^{\sqrt{n}}$, $n!$, $n^2$, $n^3$, ...)
    - **Example:** brute force search, dynamic programming for TSP
Prerequisites: graphs

- Graph $G$ with set of vertices $V(G)$ and set of edges $E(G)$.
- Directed graphs: each edge $\overrightarrow{uv}$ has an orientation.
- Basic graph-theoretic terms: degree of a vertex, connectedness, (induced) subgraphs, planar graphs, proper coloring of the vertices of a graph, clique, independent set, matching.

Classic algorithmic problems on graphs: connectivity, shortest paths, perfect matching, maximum flow, minimum $s-t$ cut, ...
Prerequisites: graph classes

Some important classes:

- Regular graphs: every vertex has the same degree
- Trees, forests
- Planar graphs
- Intersection graphs (e.g., interval graphs)
Prerequisites: optimization

- **Decision problems**: return a yes-no answer
- **Search problems**: return a solution
- **Optimization problems**: return the best solution
  - feasible solution
  - cost function
  - goal is to minimize/maximize cost

Classic optimization problems: shortest path, maximum flow, linear programming, bin packing, knapsack, . . .

Turning an optimization problem into a decision problem:
 “Is there a solution with cost at least/at most $k$?”
A brief review:

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

Can we say anything nontrivial about NP-hard problems?
Parameterized problems

Main idea

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

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

Main idea

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

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

What can be the parameter $k$?

- The size $k$ of the solution we are looking for.
- The maximum degree $\Delta$ of the input graph.
- The dimension $d$ of the point set in the input.
- The length $L$ of the strings in the input.
- The length $\ell$ of clauses in the input Boolean formula.
- ...
Parameterized complexity

Problem: Vertex Cover
Input: Graph $G$, integer $k$
Question: Is it possible to cover the edges with $k$ vertices?

Complexity: NP-complete

Problem: Independent Set
Input: Graph $G$, integer $k$
Question: Is it possible to find $k$ independent vertices?

Complexity: NP-complete
Parameterized complexity

Problem: 
Input: 
Question: 

<table>
<thead>
<tr>
<th><strong>Vertex Cover</strong></th>
<th><strong>Independent Set</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph $G$, integer $k$</td>
<td>Graph $G$, integer $k$</td>
</tr>
<tr>
<td>Is it possible to cover the edges with $k$ vertices?</td>
<td>Is it possible to find $k$ independent vertices?</td>
</tr>
</tbody>
</table>

Complexity: NP-complete

Brute force: $O(n^k)$ possibilities
Parameterized complexity

**Problem:**

**Input:** Graph $G$, integer $k$

**Question:** Is it possible to cover the edges with $k$ vertices? Is it possible to find $k$ independent vertices?

**Complexity:**

- **Vertex Cover:** NP-complete
- **Independent Set:** NP-complete

**Brute force:**

- **Vertex Cover:** $O(n^k)$ possibilities
- **Independent Set:** $O(n^k)$ possibilities

- **Vertex Cover:** $O(2^k n^2)$ algorithm exists 😊
- **Independent Set:** No $n^{o(k)}$ algorithm known 😞
Bounded search tree method

Algorithm for **Vertex Cover:**

\[ e_1 = u_1 v_1 \]
Bounded search tree method

Algorithm for **Vertex Cover**:

\[ e_1 = u_1 v_1 \]
Bounded search tree method

Algorithm for **Vertex Cover:**

\[ e_1 = u_1 v_1 \]

\[ e_2 = u_2 v_2 \]
Bounded search tree method

Algorithm for **Vertex Cover:**

\[ e_1 = u_1 v_1 \]

\[ e_2 = u_2 v_2 \]
Bounded search tree method

Algorithm for Vertex Cover:

\[ e_1 = u_1 v_1 \]

\[ e_2 = u_2 v_2 \]

Height of the search tree \( \leq k \Rightarrow \) at most \( 2^k \) leaves \( \Rightarrow 2^k \cdot n^{O(1)} \) time algorithm.
Fixed-parameter tractability

Main definition

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

Examples of NP-hard problems that are FPT:

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

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

Examples of NP-hard problems that are FPT:
- Finding a vertex cover of size $k$.
- Finding a path of length $k$.
- Finding $k$ disjoint triangles.
- Drawing the graph in the plane with $k$ edge crossings.
- Finding disjoint paths that connect $k$ pairs of points.
- ...
More formally

- We consider only **decision problems** here.
- Let $\Sigma$ be a finite alphabet used to encode the inputs
  - $(\Sigma = \{0, 1\}$ for binary encodings)
More formally

- We consider only decision problems here.
- Let $\Sigma$ be a finite alphabet used to encode the inputs
  - $(\Sigma = \{0, 1\}$ for binary encodings)
- A parameterized problem is a set $P \subseteq \Sigma^* \times \mathbb{N}$
  - $P = \{(x_1, k_1), (x_2, k_2), \ldots \}$
- The set $P$ contains the tuples $(x, k)$ where the answer to the question encoded by $(x, k)$ is yes; $k$ is the parameter
More formally

- We consider only **decision problems** here.
- Let $\Sigma$ be a finite alphabet used to encode the inputs
  - $(\Sigma = \{0, 1\}$ for binary encodings)
- A **parameterized problem** is a set $P \subseteq \Sigma^* \times \mathbb{N}$
  - $P = \{(x_1, k_1), (x_2, k_2), \ldots\}$
- The set $P$ contains the tuples $(x, k)$ where the answer to the question encoded by $(x, k)$ is yes; $k$ is the **parameter**
- A parameterized problem $P$ is **fixed-parameter tractable** if there is an algorithm that, given an input $(x, k)$
  - decides if $(x, k)$ belongs to $P$ or not, and
  - the running time is $f(k)|x|^c$ for some computable function $f$ and constant $c$. 
FPT techniques

- Bounded-depth search trees
- Kernelization
- Algebraic techniques
- Treewidth
- Color coding
- Iterative compression
W[1]-hardness

Negative evidence similar to NP-completeness. If a problem is W[1]-hard, then the problem is not FPT unless FPT=W[1].

Some W[1]-hard problems:

- Finding a clique/independent set of size $k$.
- Finding a dominating set of size $k$.
- Finding $k$ pairwise disjoint sets.
- ...
**W[1]-hardness**

Negative evidence similar to NP-completeness. If a problem is \textbf{W[1]-hard}, then the problem is not FPT unless FPT=W[1].

Some W[1]-hard problems:

- Finding a clique/independent set of size $k$.
- Finding a dominating set of size $k$.
- Finding $k$ pairwise disjoint sets.
- ...

**General principle of hardness**

With an appropriate \textbf{reduction} from $k$-\textsc{Clique} to problem $P$, we show that if problem $P$ is FPT, then $k$-\textsc{Clique} is also FPT.
Parameterized complexity

Rod G. Downey
Michael R. Fellows

Parameterized Complexity
Springer 1999

- The study of parameterized complexity was initiated by Downey and Fellows in the early 90s.
- First monograph in 1999.
- By now, strong presence in most algorithmic conferences.
Course outline

- Basic techniques
  - bounded search trees
  - color coding
  - dynamic programming
  - iterative compression

- Complexity

- Kernelization

- Treewidth

- Advanced topics:
  - cuts and separators
  - matroids
  - algebraic techniques
Bounded search tree method
Bounded search tree method

Algorithm for **Vertex Cover**

- **Main idea**: reduce problem instance \((x, k)\) to solving a bounded number of instances with parameter \(< k\).
- We should be able to solve instance \((x, k)\) in polynomial time using the solutions of the new instances.
- If the parameter strictly decreases in every recursive call, then the depth is at most \(k\).

**Size of the search tree:**
- If we branch into \(c\) directions: \(c^k\).
- If we branch into \(O(k)\) directions: \(k^{O(k)} = 2^{O(k \log k)}\).
- (If we branch into \(O(\log n)\) directions: \(O(n) + 2^{O(k \log k)}\).)
Bounded search tree method

Algorithm for **Vertex Cover**

- **Main idea**: reduce problem instance \((x, k)\) to solving a bounded number of instances with parameter \(< k\).
- We should be able to solve instance \((x, k)\) in polynomial time using the solutions of the new instances.
- If the parameter strictly decreases in every recursive call, then the depth is at most \(k\).
- **Size of the search tree**:
  - If we branch into \(c\) directions: \(c^k\).
  - If we branch into \(O(k)\) directions: \(k^{O(k)} = 2^{O(k \log k)}\).
  - (If we branch into \(O(\log n)\) directions: \(O(n) + 2^{O(k \log k)}\).)

\[
e_1 = u_1 v_1
\]

\[
e_2 = u_2 v_2
\]

\[
\leq k
\]

Next: A \(1.41^k \cdot n^{O(1)}\) time algorithm for **Vertex Cover**.
Bounded search tree method

Algorithm for **Vertex Cover**

- **Main idea:** reduce problem instance \((x, k)\) to solving a bounded number of instances with parameter \(< k\).
- We should be able to solve instance \((x, k)\) in polynomial time using the solutions of the new instances.
- If the parameter strictly decreases in every recursive call, then the depth is at most \(k\).
- Size of the search tree:
  - If we branch into \(c\) directions: \(c^k\).
  - If we branch into \(O(k)\) directions: \(k^{O(k)} = 2^{O(k \log k)}\).
  - (If we branch into \(O(\log n)\) directions: \(O(n) + 2^{O(k \log k)}\).)

Next: A \(O^*(1.41^k)\) time algorithm for **Vertex Cover**.
Improved branching for **Vertex Cover**

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

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

- If every vertex has degree $\leq 2$, then the problem can be solved in polynomial time.

- **Branching rule:**
  - If there is a vertex $v$ with at least 3 neighbors, then
    - either $v$ is in the solution, $\Rightarrow k$ decreases by 1
    - or every neighbor of $v$ is in the solution. $\Rightarrow k$ decreases by at least 3

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

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

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

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

There is a standard technique for bounding such functions asymptotically.
Better analysis

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

\[
T(k) \leq T(k - 1) + T(k - 3)
\]

There is a standard technique for bounding such functions asymptotically. We prove by induction that \( T(k) \leq c^k \) for some \( c > 1 \) as small as possible.

What values of \( c \) are good? We need:

\[
c^k \geq c^{k-1} + c^{k-3}
\]

\[
c^3 - c^2 - 1 \geq 0
\]

We need to find the roots of the characteristic equation \( c^3 - c^2 - 1 = 0 \).

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

\[ c^3 - c^2 - 1 = 0 \]

\[ c = 1.4656 \text{ is a good value} \Rightarrow T(k) \leq 1.4656^k \]

\[ \Rightarrow \text{We have a } O^*(1.4656^k) \text{ algorithm for Vertex Cover.} \]
We showed that if $T(k) \leq T(k - 1) + T(k - 3)$, then $T(k) \leq 1.4656^k$ holds.

Is this bound tight? There are three questions:

- Can the function $T(k)$ be that large? Yes (ignoring rounding problems).
- Can the search tree of the \textsc{Vertex Cover} algorithm be that large? Difficult question, hard to answer in general.
- Is this the best \textsc{Vertex Cover} algorithm? Certainly not.
Branching vectors

The branching vector of our $O^*(1.4656^k)$ Vertex Cover algorithm was $(1, 3)$.

Example: Let us bound the search tree for the branching vector $(2, 5, 6, 6, 7, 7)$. (2 out of the 6 branches decrease the parameter by 7, etc.).
The branching vector of our $O^*(1.4656^k)$ Vertex Cover algorithm was $(1, 3)$.

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

The value $c > 1$ has to satisfy:

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

$$c^7 - c^5 - c^2 - 2c - 2 \geq 0$$

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

It is hard to compare branching vectors intuitively.
Branching vectors

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

\[
T(k) \leq T(k - i) + T(k - j) \Rightarrow c^k \geq c^{k-i} + c^{k-j}
\]

\[
c^j - c^{j-i} - 1 \geq 0
\]

We compute the unique positive root.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0000</td>
<td>1.6181</td>
<td>1.4656</td>
<td>1.3803</td>
<td>1.3248</td>
<td>1.2852</td>
</tr>
<tr>
<td>2</td>
<td>1.6181</td>
<td>1.4143</td>
<td>1.3248</td>
<td>1.2721</td>
<td>1.2366</td>
<td>1.2107</td>
</tr>
<tr>
<td>3</td>
<td>1.4656</td>
<td>1.3248</td>
<td>1.2560</td>
<td>1.2208</td>
<td>1.1939</td>
<td>1.1740</td>
</tr>
<tr>
<td>4</td>
<td>1.3803</td>
<td>1.2721</td>
<td>1.2208</td>
<td>1.1893</td>
<td>1.1674</td>
<td>1.1510</td>
</tr>
<tr>
<td>5</td>
<td>1.3248</td>
<td>1.2366</td>
<td>1.1939</td>
<td>1.1674</td>
<td>1.1487</td>
<td>1.1348</td>
</tr>
<tr>
<td>6</td>
<td>1.2852</td>
<td>1.2107</td>
<td>1.1740</td>
<td>1.1510</td>
<td>1.1348</td>
<td>1.1225</td>
</tr>
</tbody>
</table>
Example: **Triangle Free Deletion**

**Triangle Free Deletion**
Given \((G, k)\), remove at most \(k\) vertices to make the graph triangle free.

What is the running time of a simple branching algorithm?
Example: **Triangle Free Deletion**

**Triangle Free Deletion**

Given \((G, k)\), remove at most \(k\) vertices to make the graph triangle free.

What is the running time of a simple branching algorithm?

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

**Note:** If the answer is “NO”, then the search tree has exactly \(3^k\) leaves.
Graph modification problems

A general problem family containing tasks of the following type:

Given \((G, k)\), do at most \(k\) allowed operations on \(G\) to make it have property \(\mathcal{P}\).

- Allowed operations: vertex deletion, edge deletion, edge addition, . . .
- Property \(\mathcal{P}\): edgeless, no triangles, no cycles, planar, chordal, regular, disconnected, . . .

Examples:

- **Vertex Cover**: Delete \(k\) vertices to make \(G\) edgeless.
- **Triangle Free Deletion**: Delete \(k\) vertices to make \(G\) triangle free.
- **Feedback Vertex Set**: Delete \(k\) vertices to make \(G\) acyclic (forest).
Hereditary properties

Definition

A graph property $\mathcal{P}$ is hereditary or closed under induced subgraphs if whenever $G \in \mathcal{P}$, every induced subgraph of $G$ is also in $\mathcal{P}$.

“removing a vertex does not ruin the property”
(e.g., triangle free, maximum degree $\leq 10$, bipartite, planar)
Hereditary properties

Definition
A graph property $\mathcal{P}$ is **hereditary** or **closed under induced subgraphs** if whenever $G \in \mathcal{P}$, every induced subgraph of $G$ is also in $\mathcal{P}$.

“removing a vertex does not ruin the property”
(e.g., triangle free, maximum degree $\leq 10$, bipartite, planar)

Observation
Every hereditary property $\mathcal{P}$ can be characterized by a (finite or infinite) set $\mathcal{F}$ of “minimal bad graphs” or “forbidden induced subgraphs”: $G \in \mathcal{P}$ if and only if $G$ does **not** have an induced subgraph isomorphic to a member of $\mathcal{F}$.

**Example:** a graph is bipartite if and only if it does not contain an odd cycle as an induced subgraph.
Hereditary properties

Observation

Every hereditary property $\mathcal{P}$ can be characterized by a (finite or infinite) set $\mathcal{F}$ of “minimal bad graphs” or “forbidden induced subgraphs”: $G \in \mathcal{P}$ if and only if $G$ does not have an induced subgraph isomorphic to a member of $\mathcal{F}$. 


Graph properties

- All graph properties
  - Hereditary properties
    - Hereditary with finite set of forbidden induced subgraphs
      - Regular
      - Bipartite
      - Triangle free
      - Connected
      - Planar
      - Empty
      - Complete
      - Acyclic
Graph properties

- All graph properties
  - Regular
  - Hereditary properties
    - Hereditary with finite set of forbidden induced subgraphs
- Planar
- Bipartite
- Triangle free
- Connected
- Empty
- Complete
- Acyclic
Graph properties

- All graph properties
  - Regular
    - Hereditary properties
      - Bipartite
        - Hereditary with finite set of forbidden induced subgraphs
  - Planar
  - Empty
  - Triangle free
  - Complete
  - Connected
  - Acyclic
Graph properties

- All graph properties
  - Regular
  - Hereditary properties
    - Bipartite
      - Hereditary with finite set of forbidden induced subgraphs
        - Triangle free
  - Connected
  - Planar
  - Empty
  - Complete
  - Acyclic
Graph properties

- All graph properties
  - Regular
  - Connected

Hereditary properties
- Bipartite
- Hereditary with finite set of forbidden induced subgraphs
  - Triangle free

Planar, Empty, Complete, Acyclic
Graph properties

- All graph properties
  - Regular
  - Connected

- Hereditary properties
  - Bipartite
  - Planar
  - Hereditary with finite set of forbidden induced subgraphs
    - Triangle free

- Empty
- Complete
- Acyclic
Graph properties

all graph properties

regular connected

hereditary properties

bipartite planar

hereditary with finite set of forbidden induced subgraphs

triangle free empty

complete acyclic
Graph properties

- All graph properties
  - Regular
  - Connected

- Hereditary properties
  - Bipartite
  - Planar
  - Hereditary with finite set of forbidden induced subgraphs
    - Triangle free
    - Empty
    - Complete

- Acyclic
Graph properties

- All graph properties

- Regular
- Connected

- Hereditary properties

- Bipartite
- Planar
- Acyclic

- Hereditary with finite set of forbidden induced subgraphs
  - Triangle free
  - Empty
  - Complete
Graph properties

- All graph properties
  - Regular
  - Connected

Hereditary properties

- Bipartite
- Planar
- Acyclic
- Hereditary with finite set of forbidden induced subgraphs
  - Triangle free
  - Empty
  - Complete

FPT
Theorem

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

Proof:

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

A very wide and active research area in parameterized algorithms.

- If the set of forbidden subgraphs is finite, then the problem is immediately FPT (e.g., Vertex Cover, Triangle Free Deletion). Here the challenge is improving the naive running time.
- If the set of forbidden subgraphs is infinite, then very different techniques are needed to show that the problem is FPT (e.g., Feedback Vertex Set, Bipartite Deletion, Planar Deletion).
**Feedback Vertex Set**

**Feedback Vertex Set:**
Given \((G, k)\), find a set \(S\) of at most \(k\) vertices such that \(G - S\) has no cycles.

- We allow multiple parallel edges and self loops.
- A **feedback vertex set** is a set that hits every cycle in the graph.
Feedback Vertex Set

Feedback Vertex Set:
Given \((G, k)\), find a set \(S\) of at most \(k\) vertices such that \(G - S\) has no cycles.

- We allow multiple parallel edges and self loops.
- A feedback vertex set is a set that hits every cycle in the graph.
If we find a cycle, then we have to include at least one of its vertices into the solution. But the length of the cycle can be arbitrary large!

**Main idea:** We identify a set of $O(k)$ vertices such that any size-$k$ feedback vertex set has to contain one of these vertices.

But first: some reductions to simplify the problem.
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.
(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.
(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.
(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.
(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.

(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.

(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.
(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 
Reduction rules

(R1) If there is a loop at \( v \), then delete \( v \) and decrease \( k \) by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex \( v \) of degree at most 1, then delete \( v \).
(R4) If there is a vertex \( v \) of degree 2, then delete \( v \) and add an edge between the neighbors of \( v \).
Reduction rules

(R1) If there is a loop at $v$, then delete $v$ and decrease $k$ by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex $v$ of degree at most 1, then delete $v$.

(R4) If there is a vertex $v$ of degree 2, then delete $v$ and add an edge between the neighbors of $v$. 

![Graph diagram]
Reduction rules

(R1) If there is a loop at \( v \), then delete \( v \) and decrease \( k \) by one.
(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.
(R3) If there is a vertex \( v \) of degree at most 1, then delete \( v \).
(R4) If there is a vertex \( v \) of degree 2, then delete \( v \) and add an edge between the neighbors of \( v \).
Reduction rules

(R1) If there is a loop at \( v \), then delete \( v \) and decrease \( k \) by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex \( v \) of degree at most 1, then delete \( v \).

(R4) If there is a vertex \( v \) of degree 2, then delete \( v \) and add an edge between the neighbors of \( v \).

If the reduction rules cannot be applied, then every vertex has degree at least 3.
Branching

Let $G$ be a graph whose vertices have degree at least 3.

- Order the vertices as $v_1, v_2, \ldots, v_n$ by **decreasing** degree (breaking ties arbitrarily).
- Let $V_{3k} = \{v_1, \ldots, v_{3k}\}$ be the $3k$ largest-degree vertices.

**Lemma**

If $G$ has minimum degree at least 3, then every feedback vertex set $S$ of size at most $k$ contains a vertex from $V_{3k}$. 

36
Branching

Let $G$ be a graph whose vertices have degree at least 3.

- Order the vertices as $v_1, v_2, \ldots, v_n$ by **decreasing** degree (breaking ties arbitrarily).
- Let $V_{3k} = \{v_1, \ldots, v_{3k}\}$ be the $3k$ largest-degree vertices.

**Lemma**

If $G$ has minimum degree at least 3, then every feedback vertex set $S$ of size at most $k$ contains a vertex from $V_{3k}$.

**Algorithm:**

- Apply the reduction rules (poly time) $\Rightarrow$ graph has minimum degree 3.
- For each vertex $v \in V_{3k}$, recurse on the instance $(G - v, k - 1)$.
- Running time $(3k)^k \cdot n^{O(1)} = 2^{O(k \log k)} \cdot n^{O(1)}$. 
Proof of the lemma

Lemma

If $G$ has minimum degree at least 3, then every feedback vertex set $S$ of size at most $k$ contains a vertex from $V_{3k}$.

- $d := \text{minimum degree in } V_{3k}$,
  \[ X = V(G) - (S \cup V_{3k}). \]
- Total degree of $V_{3k} \cup X$: $\geq 3kd + 3|X|$
- Edges of $G[V_{3k} \cup X]$: $\leq 3k + |X| - 1$
- Total degree of these edges: $\leq 6k + 2|X| - 2$

As $d \geq 3$, we have $3(d - 2) \geq d$, contradiction.
Proof of the lemma

Lemma

If $G$ has minimum degree at least 3, then every feedback vertex set $S$ of size at most $k$ contains a vertex from $V_{3k}$.

- $d :=$ minimum degree in $V_{3k}$,
  \[ X = V(G) - (S \cup V_{3k}). \]
- Total degree of $V_{3k} \cup X$: $\geq 3kd + 3|X|$
- Edges of $G[V_{3k} \cup X]$: $\leq 3k + |X| - 1$
- Total degree of these edges: $\leq 6k + 2|X| - 2$
- Edges between $S$ and $V_{3k} \cup X$:
  - $\leq dk$
  - $\geq 3kd + 3|X| - (6k + 2|X| - 2) > 3(d - 2)k$
- As $d \geq 3$, we have $3(d - 2) \geq d$, contradiction.
Branching: wrap up

- Branching into $c$ directions: $O^*(c^k)$ algorithms.
- Branching into $k$ directions: $O^*(k^k)$ algorithms.
- Branching vectors and analysis of recurrences of the form
  \[ T(k) = T(k - 1) + 2T(k - 2) + T(k - 3) \]
- Graph modification problems where the graph property can be characterized by a finite set of forbidden induced subgraphs is FPT.
The race for better FPT algorithms

- Single exponential
- Subexponential
- Double exponential
- "Slightly super-exponential"
- Tower of exponentials

$$f(k)$$

Graph with exponential growth rates:
- $2^{O(k)}$
- $2^{2^{O(k)}}$
- $2^{2^{2^{O(k)}}}$
- $\ldots$
- $2^{f(k)}$