# Fixed Parameter Tractability: an alternative approach to NP-hard problems

Michał Pilipczuk

Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

12th April 2011

# FPT — motivation

- *NP*-hard problems presumably do not admit robust solutions,

# FPT — motivation

- *NP*-hard problems presumably do not admit robust solutions,
- but maybe some instances are 'easier' than other?

# FPT — motivation

- *NP*-hard problems presumably do not admit robust solutions,
- but maybe some instances are 'easier' than other?
- For example, seeking a clique of size 10 seems easier than of size $10^6$.

# FPT — definitions

- In FPT (Fixed Parameter Tractability) setting an instance comes with a parameter $k$,

# FPT — definitions

- In FPT (Fixed Parameter Tractability) setting an instance comes with a parameter $k$,
- which measures its 'difficulty'.

# FPT — definitions

- In FPT (Fixed Parameter Tractability) setting an instance comes with a parameter $k$,
- which measures its 'difficulty'.
- Anything can be a parameter: size of the solution, maximum degree, length of formula of some logic, treewidth...

# FPT — definitions

- In FPT (Fixed Parameter Tractability) setting an instance comes with a parameter $k$,
- which measures its 'difficulty'.
- Anything can be a parameter: size of the solution, maximum degree, length of formula of some logic, treewidth...
- We are looking for an algorithm running in time $f(k)n^{O(1)}$, where $f$ is some (usually exponential) function.

# FPT — definitions

- In FPT (Fixed Parameter Tractability) setting an instance comes with a parameter $k$,
- which measures its 'difficulty'.
- Anything can be a parameter: size of the solution, maximum degree, length of formula of some logic, treewidth...
- We are looking for an algorithm running in time $f(k)n^{O(1)}$, where $f$ is some (usually exponential) function.
- **Intuition:** The whole 'exponentiality' of the algorithm has to be encapsulated in the constant factor depending only on the parameter.

# Vertex Cover

## Vertex Cover

- **Input:** undirected graph $G = (V, E)$, integer $k$

# Vertex Cover

## Vertex Cover

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$

# Vertex Cover

## Vertex Cover

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** can one choose at most $k$ vertices from $G$, so that every edge has at least one endpoint chosen?

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';
- otherwise, if $k = 0$, terminate branch;

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';
- otherwise, if $k = 0$, terminate branch;
- otherwise, take any edge $uv$ and branch into two subcases:

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';
- otherwise, if $k = 0$, terminate branch;
- otherwise, take any edge $uv$ and branch into two subcases:
  - in the first take $u$ to the solution: delete it from the graph and decrease $k$ by 1,

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';
- otherwise, if $k = 0$, terminate branch;
- otherwise, take any edge $uv$ and branch into two subcases:
  - in the first take $u$ to the solution: delete it from the graph and decrease $k$ by 1,
  - in the second take $v$ to the solution: delete it from the graph and decrease $k$ by 1.

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';
- otherwise, if $k = 0$, terminate branch;
- otherwise, take any edge $uv$ and branch into two subcases:
  - in the first take $u$ to the solution: delete it from the graph and decrease $k$ by 1,
  - in the second take $v$ to the solution: delete it from the graph and decrease $k$ by 1.
- If none of the branches returned 'YES', return 'NO'.

# Vertex Cover— FPT algorithm

- If there is no edge in the graph, answer 'YES';
- otherwise, if $k = 0$, terminate branch;
- otherwise, take any edge $uv$ and branch into two subcases:
  - in the first take $u$ to the solution: delete it from the graph and decrease $k$ by 1,
  - in the second take $v$ to the solution: delete it from the graph and decrease $k$ by 1.
- If none of the branches returned 'YES', return 'NO'.
- Running time: $2^k n^{O(1)}$.

# Digression — not everything is FPT

- There are problems, for which existence of an FPT algorithm would yield a polynomial hierarchy collapse.

# Digression — not everything is FPT

- There are problems, for which existence of an FPT algorithm would yield a polynomial hierarchy collapse.
- Examples:

# Digression — not everything is FPT

- There are problems, for which existence of an FPT algorithm would yield a polynomial hierarchy collapse.
- Examples:
  - Clique (W[1]-complete),

# Digression — not everything is FPT

- There are problems, for which existence of an FPT algorithm would yield a polynomial hierarchy collapse.
- Examples:
  - Clique (W[1]-complete),
  - Dominating Set (W[2]-complete),

# Digression — not everything is FPT

- There are problems, for which existence of an FPT algorithm would yield a polynomial hierarchy collapse.
- Examples:
  - Clique (W[1]-complete),
  - Dominating Set (W[2]-complete),
  - Set Cover (W[2]-complete, parameterized by solution size).

# Kernelization — motivation

- Formalization of the idea of preprocessing.

# Kernelization — motivation

- Formalization of the idea of preprocessing.
- **Intuition:** Question about a vertex cover of size 30 in a graph with $10^6$ vertices is a nonsense.

# Kernelization — motivation

- Formalization of the idea of preprocessing.
- **Intuition:** Question about a vertex cover of size 30 in a graph with $10^6$ vertices is a nonsense.
- Maybe in such a situation we could preprocess the graph to shrink its size?

# Kernelization — definitions

- We say that a parameterized problem has a kernel iff there exists a polynomial time algorithm that

# Kernelization — definitions

- We say that a parameterized problem has a kernel iff there exists a polynomial time algorithm that
  - takes an instance $(G, k)$ of the problem,

# Kernelization — definitions

- We say that a parameterized problem has a kernel iff there exists a polynomial time algorithm that
  - takes an instance $(G, k)$ of the problem,
  - returns an equivalent instance $(G', k')$,

# Kernelization — definitions

- We say that a parameterized problem has a kernel iff there exists a polynomial time algorithm that
  - takes an instance $(G, k)$ of the problem,
  - returns an equivalent instance $(G', k')$,
  - where $|G'|, k' \leqslant f(k)$ for some function $f$.

# Kernelization — definitions

- We say that a parameterized problem has a kernel iff there exists a polynomial time algorithm that
  - takes an instance $(G, k)$ of the problem,
  - returns an equivalent instance $(G', k')$,
  - where $|G'|, k' \leqslant f(k)$ for some function $f$.
- Abstract nonsens: decidable problem is FPT iff it has a kernel.

# Kernelization — definitions

- We say that a parameterized problem has a kernel iff there exists a polynomial time algorithm that
  - takes an instance $(G, k)$ of the problem,
  - returns an equivalent instance $(G', k')$,
  - where $|G'|, k' \leqslant f(k)$ for some function $f$.
- Abstract nonsens: decidable problem is FPT iff it has a kernel.
- What is really interesting is whether a problem admits a **polynomial** kernel, i.e., such that $f$ is a polynomial.

# Polykernel Vertex Cover

- We seek a kernel for Vertex Cover.

# Polykernel Vertex Cover

- We seek a kernel for Vertex Cover.
- We state a sequence of reductions, we always apply the one with the lowest number among applicable.

# Polykernel Vertex Cover

- We seek a kernel for Vertex Cover.
- We state a sequence of reductions, we always apply the one with the lowest number among applicable.
- Reduction 1: if there is an isolated vertex, remove it.

# Polykernel Vertex Cover

- We seek a kernel for Vertex Cover.
- We state a sequence of reductions, we always apply the one with the lowest number among applicable.
- Reduction 1: if there is an isolated vertex, remove it.
- Reduction 2: if there is a vertex of degree more than $k$, delete it and decrease $k$ by 1.

# Polykernel Vertex Cover

- We seek a kernel for Vertex Cover.
- We state a sequence of reductions, we always apply the one with the lowest number among applicable.
- Reduction 1: if there is an isolated vertex, remove it.
- Reduction 2: if there is a vertex of degree more than $k$, delete it and decrease $k$ by 1.
- Reduction 3: if there are more than $k^2$ edges in the graph, answer 'NO'.

# Polykernel for Vertex Cover— digression

- We have a kernel of size $O(k^2)$ (quadratic).

# Polykernel for Vertex Cover— digression

- We have a kernel of size $O(k^2)$ (quadratic).
- It can be done better — a kernel with $2k$ vertices,

# Polykernel for Vertex Cover— digression

- We have a kernel of size $O(k^2)$ (quadratic).
- It can be done better — a kernel with $2k$ vertices,
  - One approach: Hall's theorem, Expansion Lemma.

# Polykernel for Vertex Cover— digression

- We have a kernel of size $O(k^2)$ (quadratic).
- It can be done better — a kernel with $2k$ vertices,
  - One approach: Hall's theorem, Expansion Lemma.
  - Second approach: Linear Programming.

# Polykernel for Vertex Cover— digression

- We have a kernel of size $O(k^2)$ (quadratic).
- It can be done better — a kernel with $2k$ vertices,
  - One approach: Hall's theorem, Expansion Lemma.
  - Second approach: Linear Programming.
  - Unless polynomial hierarchy collapses, the number of edges in the kernel cannot be $O(k^{2-\epsilon})$. [Dell, Melkebeek, STOC 2010]

# Kernelization — methods

- In order to achieve small kernels, we often use algorithmic versions of classical results of graph theory:

# Kernelization — methods

- In order to achieve small kernels, we often use algorithmic versions of classical results of graph theory:
  - mathing theory: Hall's theorem, Tutte's theorem;

# Kernelization — methods

- In order to achieve small kernels, we often use algorithmic versions of classical results of graph theory:
  - mathing theory: Hall's theorem, Tutte's theorem;
  - flows: Menger's theorem, Gallai's theorem

# Kernelization — methods

- In order to achieve small kernels, we often use algorithmic versions of classical results of graph theory:
  - mathing theory: Hall's theorem, Tutte's theorem;
  - flows: Menger's theorem, Gallai's theorem
  - set theory: $\Delta$–lemma,

# Kernelization — methods

- In order to achieve small kernels, we often use algorithmic versions of classical results of graph theory:
  - mathing theory: Hall's theorem, Tutte's theorem;
  - flows: Menger's theorem, Gallai's theorem
  - set theory: $\Delta$–lemma,
  - topological properties: grid theorem, separator theorems.

# Kernelization — methods

- In order to achieve small kernels, we often use algorithmic versions of classical results of graph theory:
  - mathing theory: Hall's theorem, Tutte's theorem;
  - flows: Menger's theorem, Gallai's theorem
  - set theory: $\Delta$–lemma,
  - topological properties: grid theorem, separator theorems.
- What is really interesting, is that we can show **negative** results about existence of polynomial kernels!

# Compositionality

## Compositionality

We say that a parameterized problem $L$ is compositional, iff there exists a polynomial time algorithm that

- takes a sequence of instances $(x_1, k), (x_2, k), \ldots, (x_m, k)$;

# Compositionality

## Compositionality

We say that a parameterized problem $L$ is compositional, iff there exists a polynomial time algorithm that

- takes a sequence of instances $(x_1, k), (x_2, k), \ldots, (x_m, k)$;
- returns one instance $(x', k')$, where

# Compositionality

## Compositionality

We say that a parameterized problem $L$ is compositional, iff there exists a polynomial time algorithm that

- takes a sequence of instances $(x_1, k), (x_2, k), \ldots, (x_m, k)$;
- returns one instance $(x', k')$, where
  - $k' = poly(k)$,

# Compositionality

## Compositionality

We say that a parameterized problem $L$ is compositional, iff there exists a polynomial time algorithm that

- takes a sequence of instances $(x_1, k), (x_2, k), \ldots, (x_m, k)$;
- returns one instance $(x', k')$, where
  - $k' = poly(k)$,
  - $(x', k')$ is a YES-instance iff at least one $(x_i, k)$ is a YES-instance.

# Compositionality means hardness

- Compositionality — logical OR without parameter blow-up.

# Compositionality means hardness

- Compositionality — logical OR without parameter blow-up.

**Theorem [Fortnow, Santhanam, STOC 2008; Bodlaender et al.]**

If a problem $L$ is in $NP$ when unparameterized, and is compositional, then it has no polykernel unless $\Sigma_3^p = PH$.

# Longest Path

## Longest Path

- **Input:** undirected graph $G = (V, E)$, integer $k$

# Longest Path

## Longest Path

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$

# Longest Path

## Longest Path

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** is there a path of length $k$ in $G$?

# No polykernel for Longest Path

- Longest Path is compositional — we take the disjoint sum of the instances;

# No polykernel for Longest Path

- Longest Path is compositional — we take the disjoint sum of the instances;
- hence, it admits no polykernel, unless polynomial hierarchy collapses.

# No polykernel for Longest Path

- Longest Path is compositional — we take the disjoint sum of the instances;
- hence, it admits no polykernel, unless polynomial hierarchy collapses.
- Unfortunately, for many problems no composition algorithm can be seen at the first glance.

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that
  - given an instance $(G, k)$ of $K$,

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that
  - given an instance $(G, k)$ of $K$,
  - produces an equivalent instance $(G', k')$ of $L$,

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that
  - given an instance $(G, k)$ of $K$,
  - produces an equivalent instance $(G', k')$ of $L$,
  - such that $k' = poly(k)$.

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that
  - given an instance $(G, k)$ of $K$,
  - produces an equivalent instance $(G', k')$ of $L$,
  - such that $k' = poly(k)$.
- We denote $K \leqslant_{ptp} L$.

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that
  - given an instance $(G, k)$ of $K$,
  - produces an equivalent instance $(G', k')$ of $L$,
  - such that $k' = poly(k)$.
- We denote $K \leqslant_{ptp} L$.
- If $K$ does not admit a polynomial kernel, then $L$ also does not admit (assuming $K, L$ are $NP$-complete).

# Parameterized reductions

- A parameterized reduction between problems $K$ and $L$ is a polynomial time algorithm that
  - given an instance $(G, k)$ of $K$,
  - produces an equivalent instance $(G', k')$ of $L$,
  - such that $k' = poly(k)$.
- We denote $K \leqslant_{ptp} L$.
- If $K$ does not admit a polynomial kernel, then $L$ also does not admit (assuming $K, L$ are $NP$-complete).
- Otherwise we could apply the reduction, perform kernelization, and reduce $L$ to $K$ using $NP$-completeness.

# Auxillary problem technique: example

- **Technique:** construct a reducible, easier problem that admits a simple compositional algorithm.

# Auxillary problem technique: example

- **Technique:** construct a reducible, easier problem that admits a simple compositional algorithm.
- **Example:**

# Auxillary problem technique: example

- **Technique:** construct a reducible, easier problem that admits a simple compositional algorithm.
- **Example:**
  - Steiner Tree (ST),

# Auxillary problem technique: example

- **Technique:** construct a reducible, easier problem that admits a simple compositional algorithm.
- **Example:**
  - Steiner Tree (ST),
  - Connected Dominating Set (CDS),

# Auxillary problem technique: example

- **Technique:** construct a reducible, easier problem that admits a simple compositional algorithm.
- **Example:**
  - Steiner Tree (ST),
  - Connected Dominating Set (CDS),
- both in $d$-degenerate graphs.

# Definitions

## Connected Dominating Set

- **Input:** undirected graph $G = (V, E)$, integer $k$

# Definitions

## Connected Dominating Set

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$

# Definitions

## Connected Dominating Set

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** can one choose a set of vertices $X$ of cardinality at most $k$, such that $G[X]$ is connected and $N[X] = V$?

# Definitions

## Connected Dominating Set

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** can one choose a set of vertices $X$ of cardinality at most $k$, such that $G[X]$ is connected and $N[X] = V$?

## Steiner Tree

- **Input:** undirected graph $G = (V, E)$, a set of terminals $T$, integer $k$

# Definitions

## Connected Dominating Set

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** can one choose a set of vertices $X$ of cardinality at most $k$, such that $G[X]$ is connected and $N[X] = V$?

## Steiner Tree

- **Input:** undirected graph $G = (V, E)$, a set of terminals $T$, integer $k$
- **Parameter:** $k + |T|$

# Definitions

## Connected Dominating Set

- **Input:** undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** can one choose a set of vertices $X$ of cardinality at most $k$, such that $G[X]$ is connected and $N[X] = V$?

## Steiner Tree

- **Input:** undirected graph $G = (V, E)$, a set of terminals $T$, integer $k$
- **Parameter:** $k + |T|$
- **Question:** can one choose a set of vertices $X$ of cardinality at most $k$, such that $G[X \cup T]$ is connected?

# $d$–degenerate graphs

**$d$–degeneracy**

A graph $G$ is called $d$–degenerate is every its subgraph contains a vertex of degree at most $d$.

- Examples:

# $d$–degenerate graphs

**$d$–degeneracy**

A graph $G$ is called $d$–degenerate is every its subgraph contains a vertex of degree at most $d$.

- Examples:
  - 1–degenerate graphs are exactly forests,

# $d$–degenerate graphs

**$d$–degeneracy**

A graph $G$ is called $d$–degenerate is every its subgraph contains a vertex of degree at most $d$.

- Examples:
  - 1–degenerate graphs are exactly forests,
  - every planar graph is 5–degenerate.

# $d$–degenerate graphs

**$d$–degeneracy**

A graph $G$ is called $d$–degenerate is every its subgraph contains a vertex of degree at most $d$.

- Examples:
  - 1–degenerate graphs are exactly forests,
  - every planar graph is 5–degenerate.
- However, $d$-degenerate graphs do not have 'topological' structure.

# $d$–degenerate graphs

**$d$–degeneracy**

A graph $G$ is called $d$–degenerate is every its subgraph contains a vertex of degree at most $d$.

- Examples:
  - 1–degenerate graphs are exactly forests,
  - every planar graph is 5–degenerate.
- However, $d$-degenerate graphs do not have 'topological' structure.
  - Given a graph $G$ obtain $G'$ by inserting a vertex inside each edge. Then $G'$ is 2–degenerate.

# Motivation

- CDS in general graphs is known to be $W[2]$–hard.

# Motivation

- $\mathrm{CDS}$ in general graphs is known to be $W[2]$–hard.
- However, FPT in $d$–degenerate graphs for bounded $d$.

# Motivation

- CDS in general graphs is known to be $W[2]$–hard.
- However, FPT in $d$–degenerate graphs for bounded $d$.
- Dominating Set has a polynomial kernel in $K_{d+1,d+1}$–free graphs (a superclass).

# Motivation

- CDS in general graphs is known to be $W[2]$–hard.
- However, FPT in $d$–degenerate graphs for bounded $d$.
- Dominating Set has a polynomial kernel in $K_{d+1,d+1}$–free graphs (a superclass).
- A natural question: does CDS have a polynomial kernel as well?

# Colourful Graph Motif

## Colourful Graph Motif (CGM)

- **Input:** graph $G = (V, E)$ and a colouring function $\mathcal{C} : V \to \{1, 2, \ldots, k\}$

# Colourful Graph Motif

## Colourful Graph Motif (CGM)

- **Input:** graph $G = (V, E)$ and a colouring function $\mathcal{C} : V \to \{1, 2, \ldots, k\}$
- **Parameter:** $k$

# Colourful Graph Motif

## Colourful Graph Motif (CGM)

- **Input:** graph $G = (V, E)$ and a colouring function $\mathcal{C} : V \to \{1, 2, \ldots, k\}$
- **Parameter:** $k$
- **Question:** Does there exists a connected subgraph $H$ of $G$ containing exactly one vertex of each colour?

# Colourful Graph Motif

## Colourful Graph Motif (CGM)

- **Input:** graph $G = (V, E)$ and a colouring function $\mathcal{C} : V \to \{1, 2, \ldots, k\}$
- **Parameter:** $k$
- **Question:** Does there exists a connected subgraph $H$ of $G$ containing exactly one vertex of each colour?

- Introduced by Fellows et al. [ICALP'07].

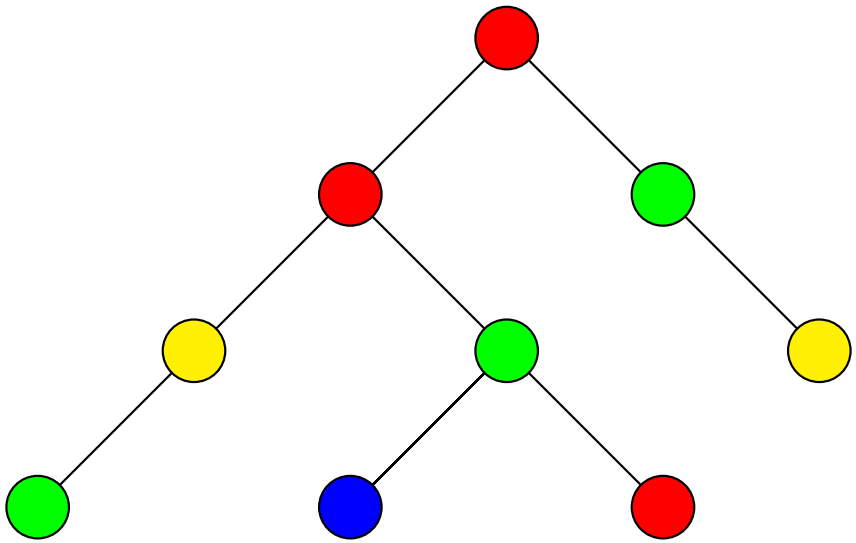
# Colourful Graph Motif

## Colourful Graph Motif (CGM)

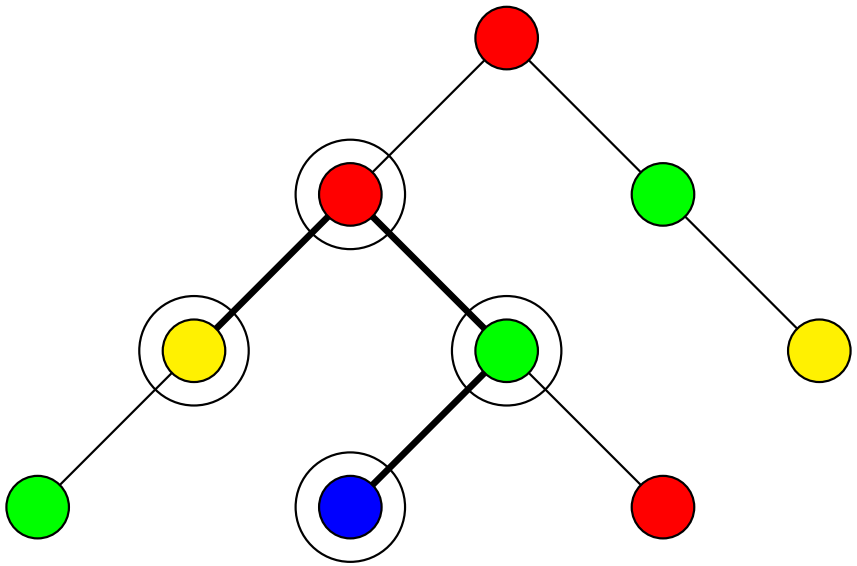
- **Input:** graph $G = (V, E)$ and a colouring function $\mathcal{C} : V \to \{1, 2, \ldots, k\}$
- **Parameter:** $k$
- **Question:** Does there exists a connected subgraph $H$ of $G$ containing exactly one vertex of each colour?

- Introduced by Fellows et al. [ICALP'07].
- Special case of Group Steiner Tree — $2^k|G|^{O(1)}$ FPT algorithm.

# Colourful Graph Motif – example

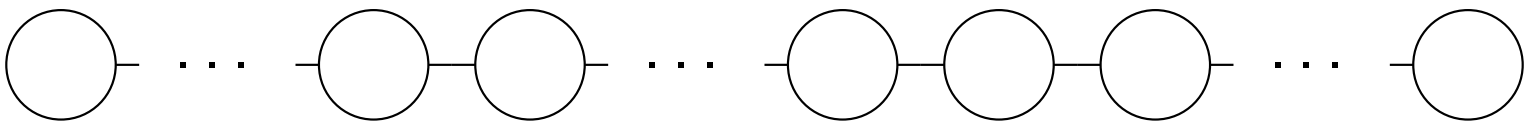


# Colourful Graph Motif – example



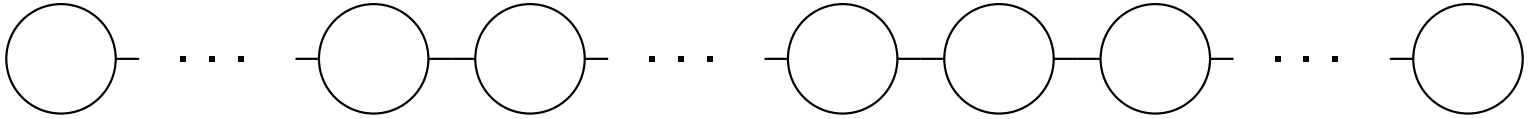
# Colourful Graph Motif – NP hardness

# Colourful Graph Motif – NP hardness

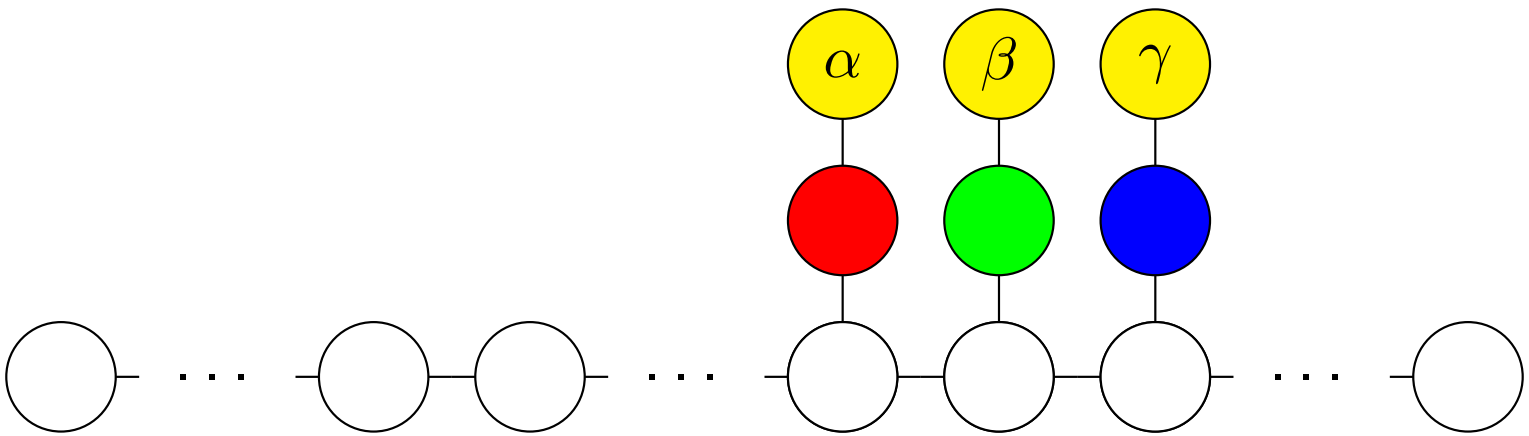


# Colourful Graph Motif – NP hardness

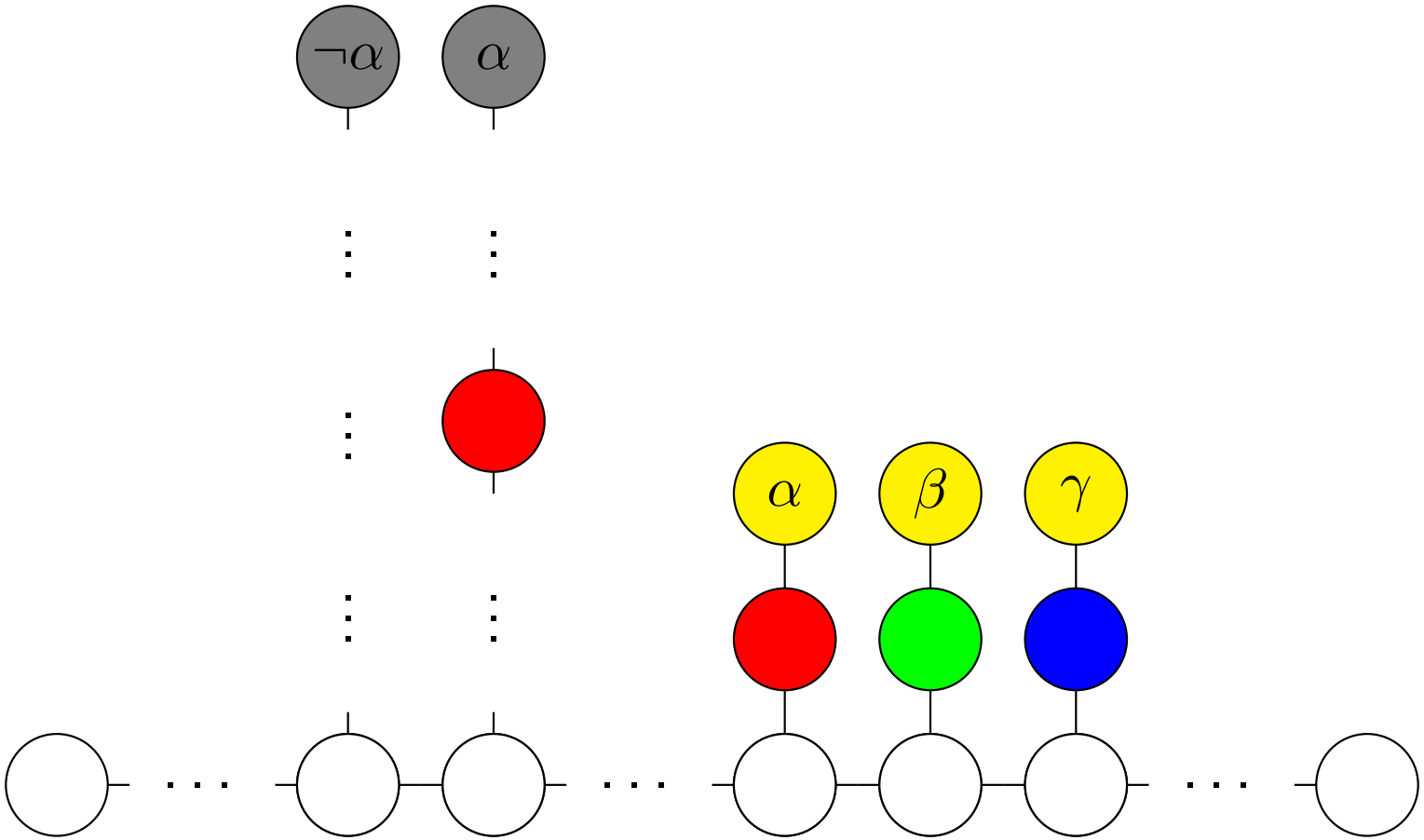
$$\alpha \vee \beta \vee \gamma$$



# Colourful Graph Motif – NP hardness



# Colourful Graph Motif – NP hardness



# Colourful Graph Motif – composition

- The resulting graph is a special type of a tree – a spine with paths attached (a **comb**).

# Colourful Graph Motif – composition

- The resulting graph is a special type of a tree – a spine with paths attached (a **comb**).
- Composition algorithm: disjoint sum.

# Colourful Graph Motif – composition

- The resulting graph is a special type of a tree – a spine with paths attached (a **comb**).
- Composition algorithm: disjoint sum.
- But **reusing** the colours — the parameter does not increase!

# Colourful Graph Motif – composition

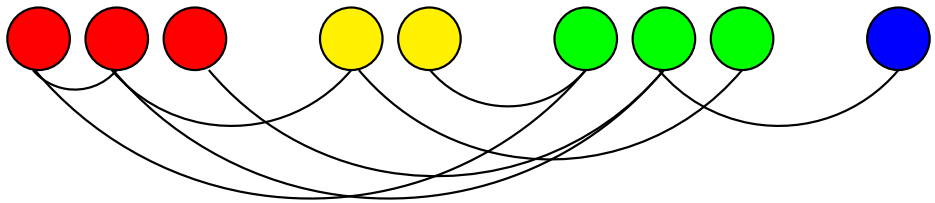
- The resulting graph is a special type of a tree – a spine with paths attached (a **comb**).
- Composition algorithm: disjoint sum.
- But **reusing** the colours — the parameter does not increase!
- Thus, Colourful Graph Motif does not admit polynomial kernel in sets of disjoint combs (unless $\Sigma_3 = PH$).

# Colourful Graph Motif – composition

- The resulting graph is a special type of a tree – a spine with paths attached (a **comb**).
- Composition algorithm: disjoint sum.
- But **reusing** the colours — the parameter does not increase!
- Thus, Colourful Graph Motif does not admit polynomial kernel in sets of disjoint combs (unless $\Sigma_3 = PH$).
  - **Remark:** There is a simple reduction to a case of one comb (technical).

# Connected Dominating Set

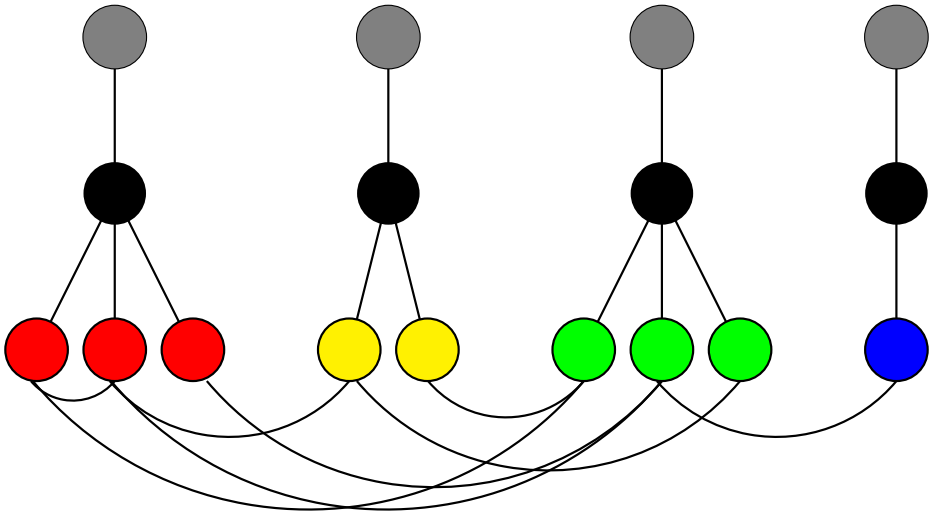
$$\text{1-deg-CGM} \leqslant \text{2-deg-CDS}$$



- Ask for Connected Dominating Set with $2k = 8$ nodes.

# Connected Dominating Set

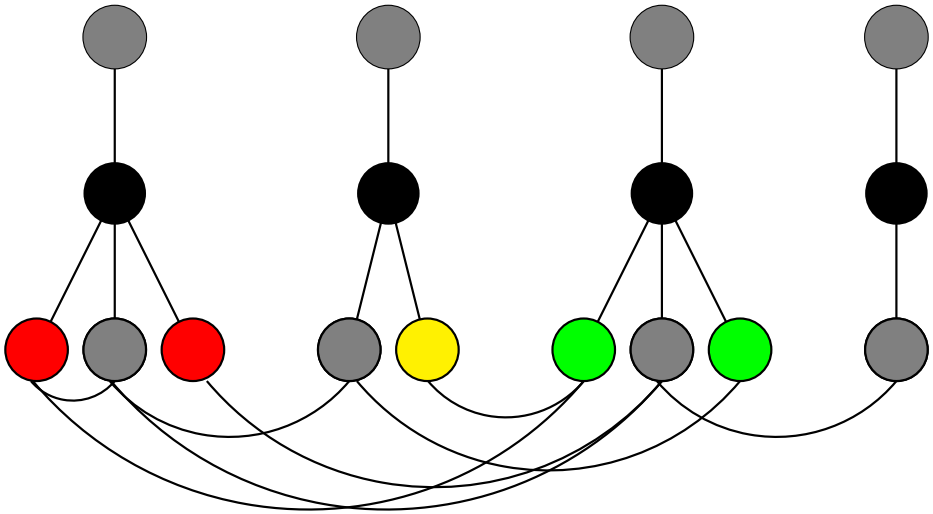
$$\text{1-deg-CGM} \leqslant \text{2-deg-CDS}$$



- Ask for Connected Dominating Set with $2k = 8$ nodes.
- Need to use black nodes and one node of each colour.

# Connected Dominating Set

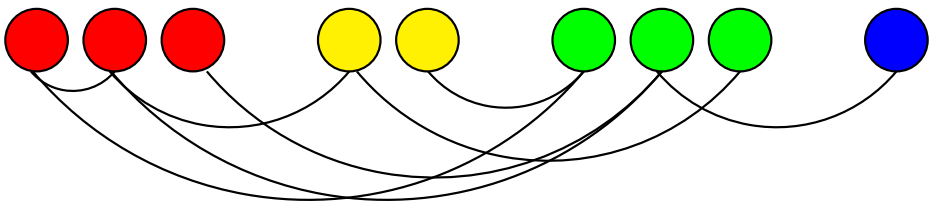
$$1\text{-deg-CGM} \leqslant 2\text{-deg-CDS}$$



- Ask for Connected Dominating Set with $2k = 8$ nodes.
- Need to use black nodes and one node of each colour.

# Steiner Tree

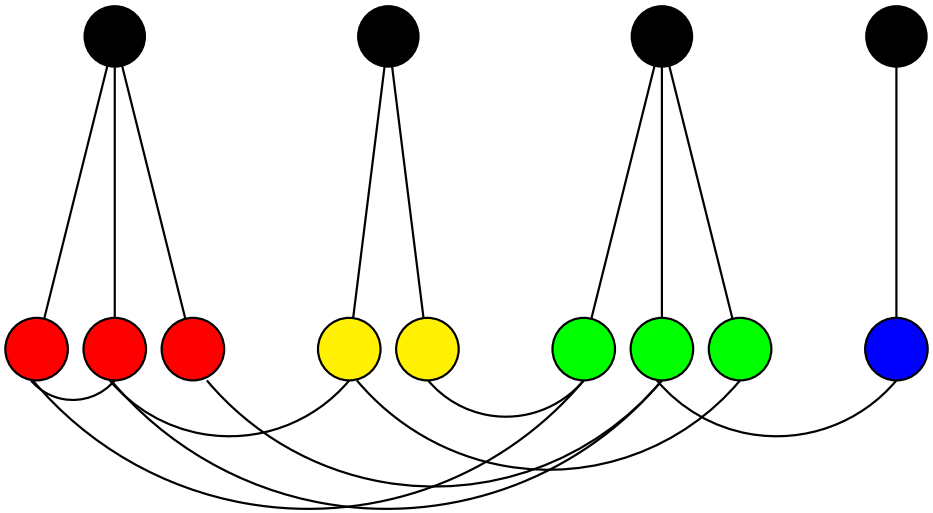
$$1\text{-deg-CGM} \leqslant 2\text{-deg-ST}$$



- Black nodes are terminals.

# Steiner Tree

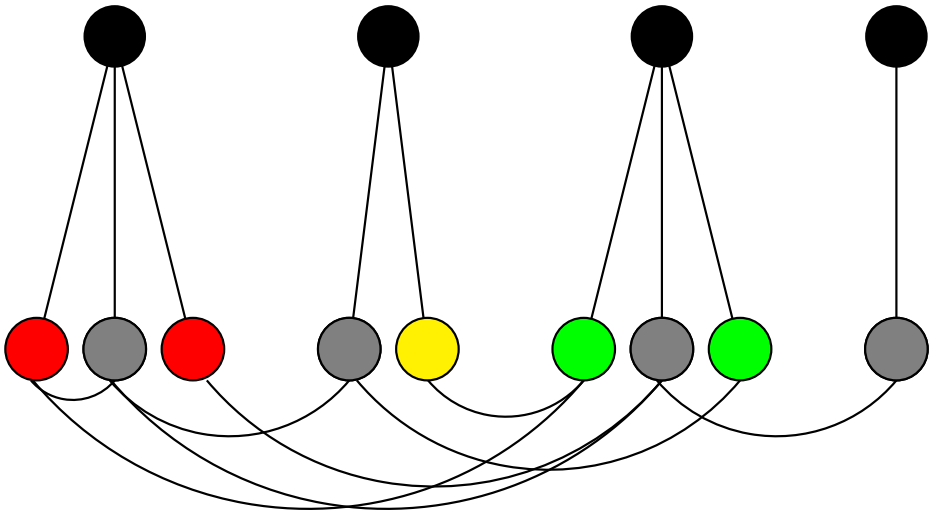
$$\text{1-deg-CGM} \leqslant \text{2-deg-ST}$$



- Black nodes are terminals.
- Ask for Steiner Tree with $k = 4$ Steiner nodes.

# Steiner Tree

$$\text{1-deg-CGM} \leqslant \text{2-deg-ST}$$



- Black nodes are terminals.
- Ask for Steiner Tree with $k = 4$ Steiner nodes.

# FPT algorithm for Longest Path

- We will present a **randomized** FPT algorithm for Longest Path, using **colour coding**.

# FPT algorithm for Longest Path

- We will present a **randomized** FPT algorithm for Longest Path, using **colour coding**.
- Firstly, randomly colour the vertices of the graph in $k$ colours (each vertex independently, with uniform distribution).

# FPT algorithm for Longest Path

- We will present a **randomized** FPT algorithm for Longest Path, using **colour coding**.
- Firstly, randomly colour the vertices of the graph in $k$ colours (each vertex independently, with uniform distribution).
- Then, using dynamic programming, determine, whether there exists a path with the first vertex of the first colour, the second of the second colour etc.

# FPT algorithm for Longest Path: analysis

- If there is no $k$-path, no matter, which the colouring will be chosen, the algorithm will not find a $k$-path.

# FPT algorithm for Longest Path: analysis

- If there is no $k$-path, no matter, which the colouring will be chosen, the algorithm will not find a $k$-path.
- If there is a $k$-path, it will be properly coloured and found with probability $\frac{1}{k^k}$.

# FPT algorithm for Longest Path: analysis

- If there is no $k$-path, no matter, which the colouring will be chosen, the algorithm will not find a $k$-path.
- If there is a $k$-path, it will be properly coloured and found with probability $\frac{1}{k^k}$.
- We can independently run the algorithm $k^k \ln 2$ times, and the probability that the solution was not found is bounded by $\frac{1}{2}$.

# FPT algorithm for Longest Path: analysis

- If there is no $k$-path, no matter, which the colouring will be chosen, the algorithm will not find a $k$-path.
- If there is a $k$-path, it will be properly coloured and found with probability $\frac{1}{k^k}$.
- We can independently run the algorithm $k^k \ln 2$ times, and the probability that the solution was not found is bounded by $\frac{1}{2}$.
- **Remark:** We could look for a multi-coloured $k$-path, relaxing the condition about the order of the colours. The dynamic program will now run in $2^k|V|^{O(1)}$ time, but the probability of finding a solution in a single run will be around $e^{-k}$. Thus we have $(2e)^k|V|^{O(1)}$ algorithm.

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.
- We use a *splitter*.

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.
- We use a *splitter*.
- $(n, k, l)$–splitter is a family $\mathcal{F}$ of functions $[n] \to [l]$ such that for every subset $X \subseteq [n]$ of cardinality $k$, there is a function from $\mathcal{F}$ injective on $X$.

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.
- We use a *splitter*.
- $(n, k, l)$–splitter is a family $\mathcal{F}$ of functions $[n] \to [l]$ such that for every subset $X \subseteq [n]$ of cardinality $k$, there is a function from $\mathcal{F}$ injective on $X$.
- A $(n, r, r^2)$–splitter of size $O(r^6 \log r \log n)$ can be found in polynomial time [Naor et al].

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.
- We use a *splitter*.
- $(n, k, l)$–splitter is a family $\mathcal{F}$ of functions $[n] \to [l]$ such that for every subset $X \subseteq [n]$ of cardinality $k$, there is a function from $\mathcal{F}$ injective on $X$.
- A $(n, r, r^2)$–splitter of size $O(r^6 \log r \log n)$ can be found in polynomial time [Naor et al].
  - Technique: perfect hashing families.

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.
- We use a *splitter*.
- $(n, k, l)$–splitter is a family $\mathcal{F}$ of functions $[n] \to [l]$ such that for every subset $X \subseteq [n]$ of cardinality $k$, there is a function from $\mathcal{F}$ injective on $X$.
- A $(n, r, r^2)$–splitter of size $O(r^6 \log r \log n)$ can be found in polynomial time [Naor et al].
  - Technique: perfect hashing families.
- Derandomize the $k^k$ algorithm: compose whole $(n, k, k^2)$–splitter with all the functions $[k^2] \to [k]$, brutally iterate.

# Longest Path: derandomization

- We need to iterate through a family of colourings, such that for every sequence of $k$ vertices there is a colouring that colours the sequence in colours $1, 2, 3, \ldots, k$.
- We use a *splitter*.
- $(n, k, l)$–splitter is a family $\mathcal{F}$ of functions $[n] \longrightarrow [l]$ such that for every subset $X \subseteq [n]$ of cardinality $k$, there is a function from $\mathcal{F}$ injective on $X$.
- A $(n, r, r^2)$–splitter of size $O(r^6 \log r \log n)$ can be found in polynomial time [Naor et al].
  - Technique: perfect hashing families.
- Derandomize the $k^k$ algorithm: compose whole $(n, k, k^2)$–splitter with all the functions $[k^2] \longrightarrow [k]$, brutally iterate.
- Derandomize the $(2e)^k$ algorithm: take a smarter hashing family $[k^2] \longrightarrow [k]$ than all the functions.

# Eulerian Edge Deletion

## Eulerian Edge Deletion

- **Input:** connected undirected graph $G = (V, E)$, integer $k$

# Eulerian Edge Deletion

## Eulerian Edge Deletion

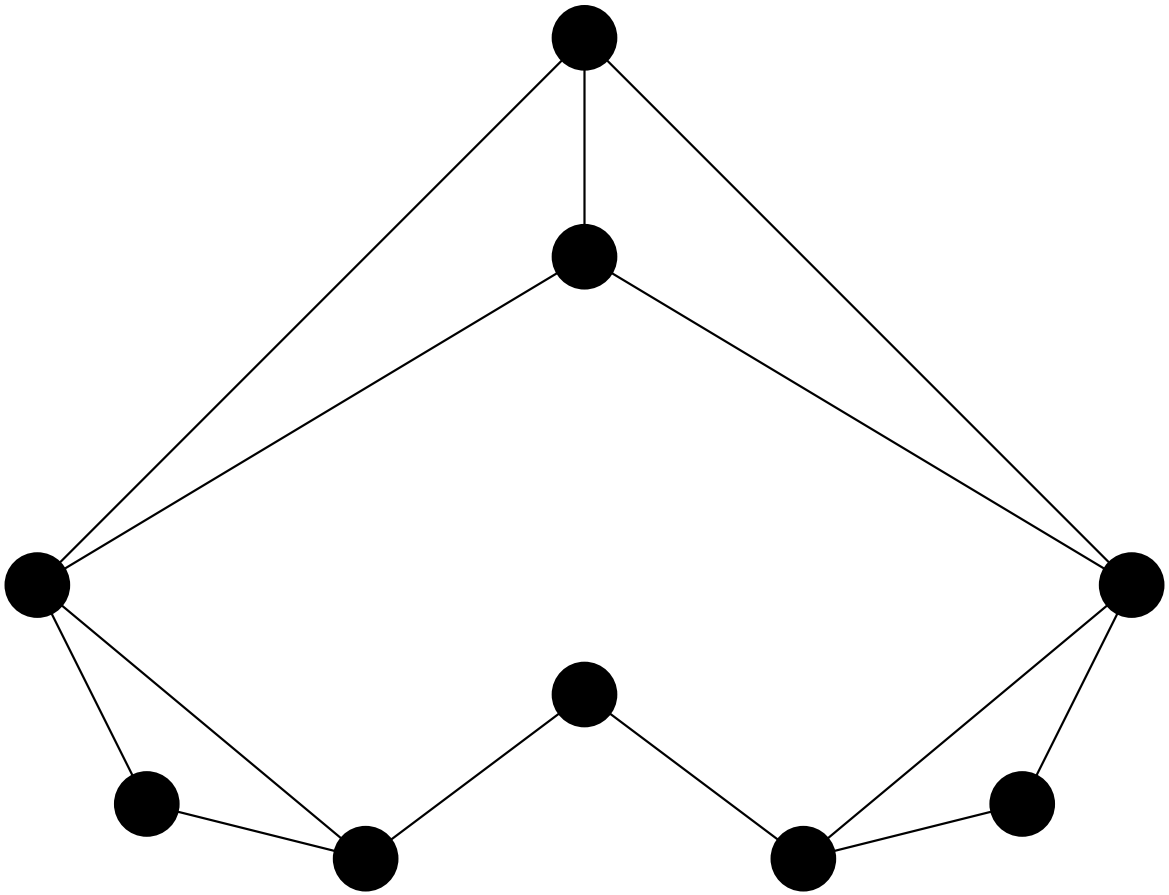
- **Input:** connected undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$

# Eulerian Edge Deletion

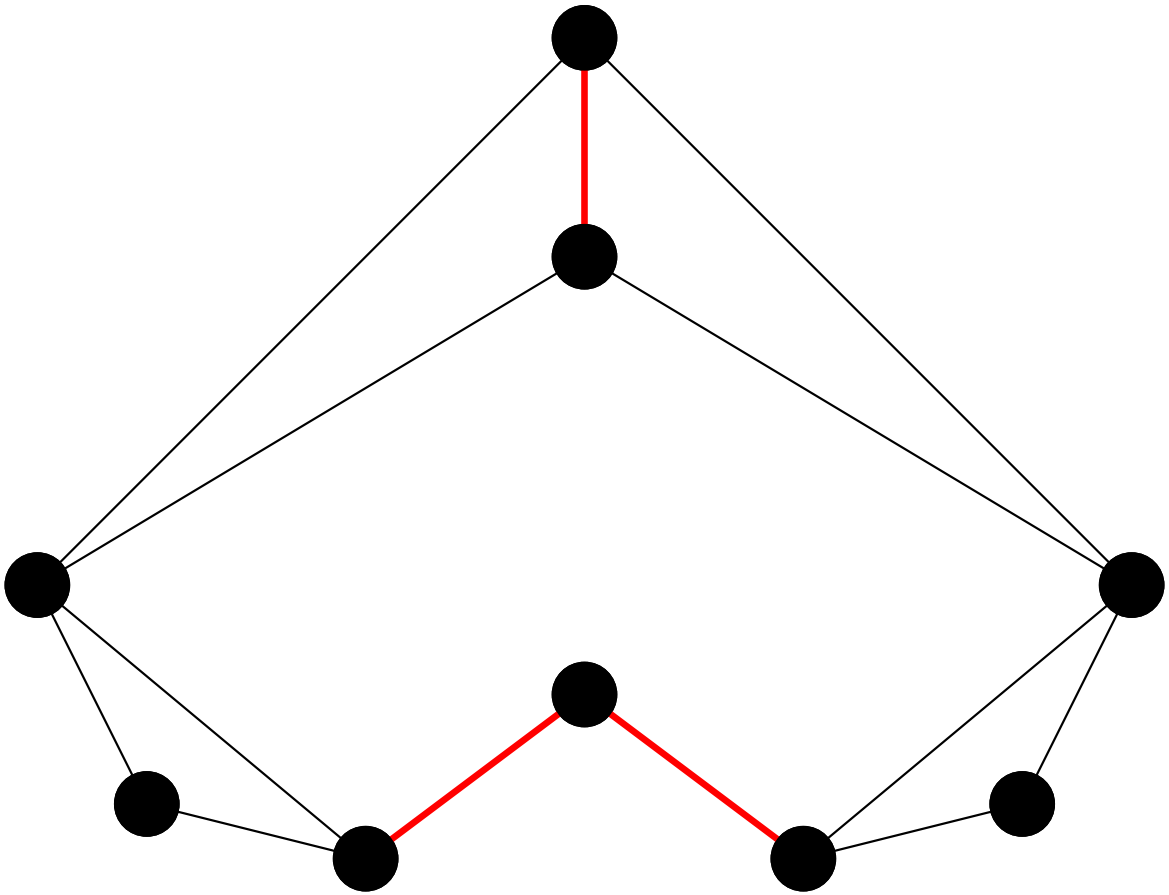
## Eulerian Edge Deletion

- **Input:** connected undirected graph $G = (V, E)$, integer $k$
- **Parameter:** $k$
- **Question:** can one delete at most $k$ edges from $G$, so that $G$ becomes Eulerian, i.e., is connected and all the degrees are even?

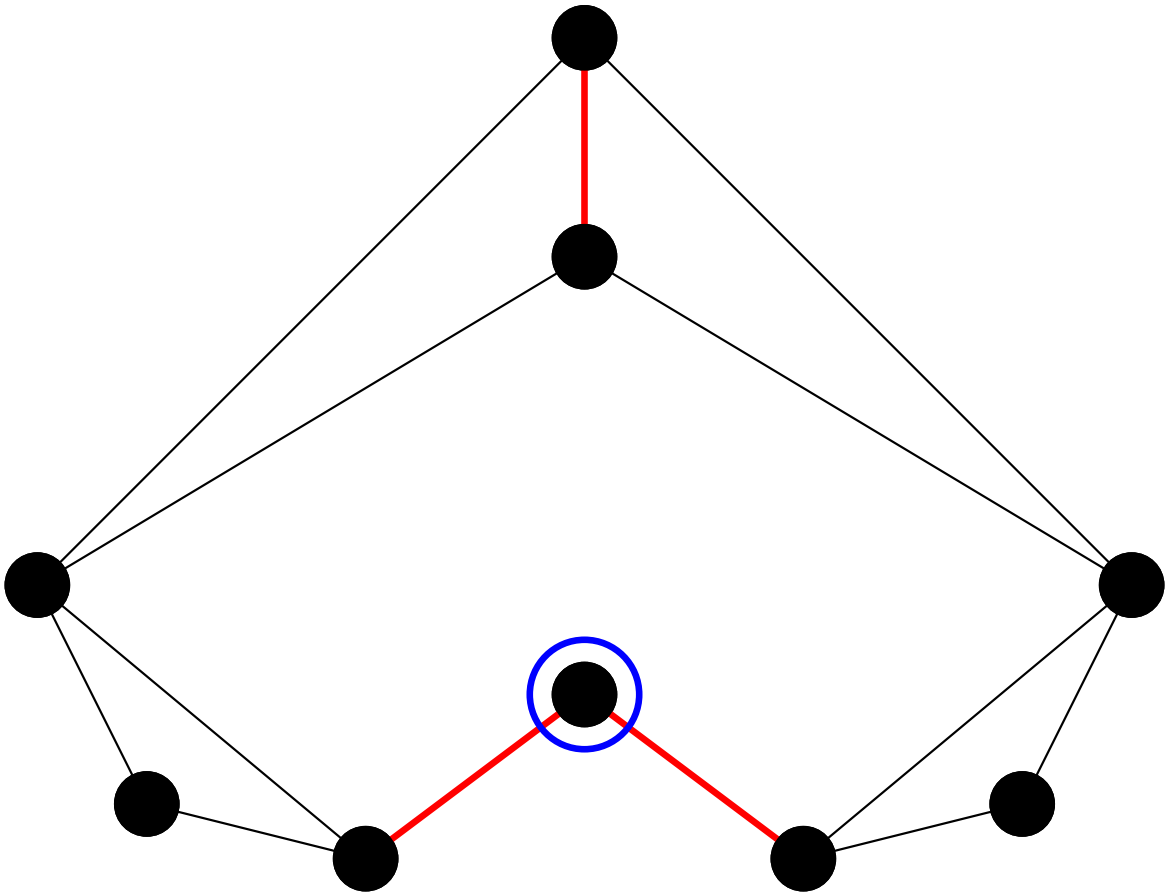
# Eulerian Edge Deletion— example



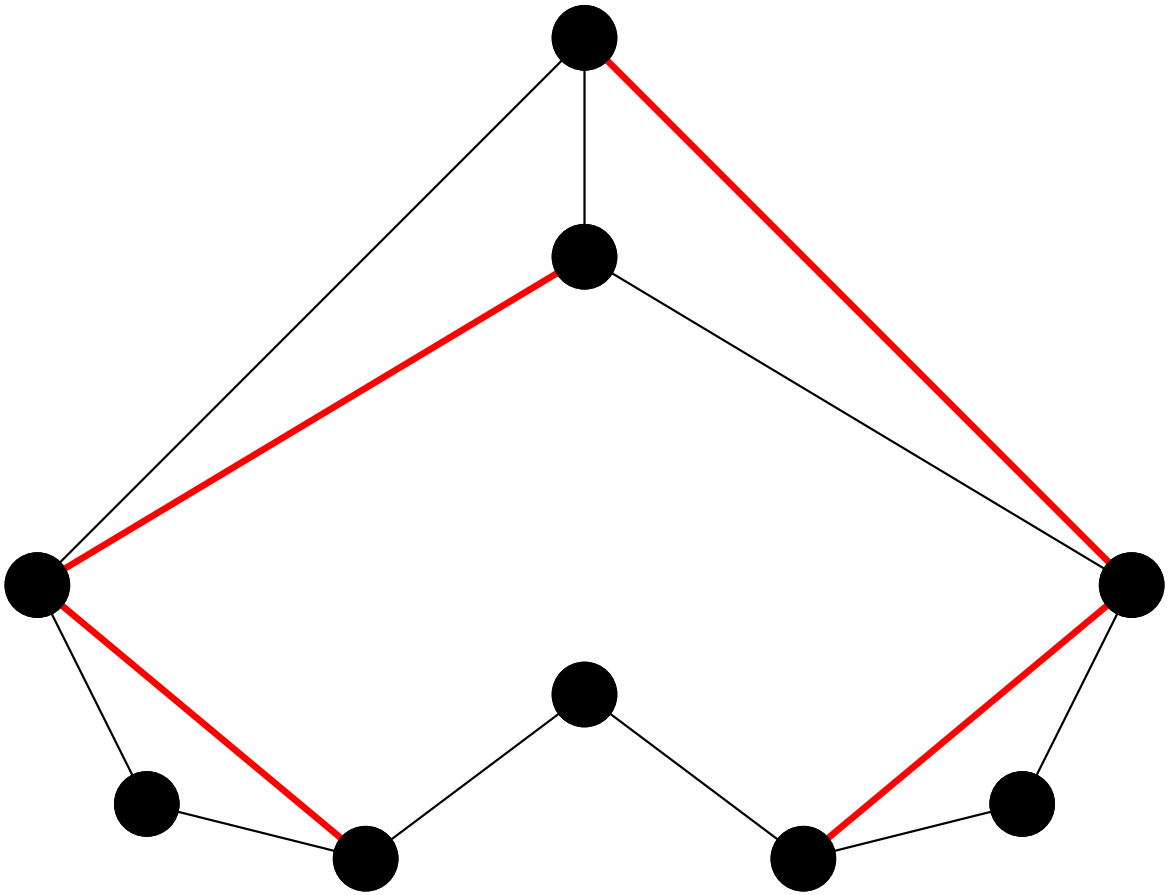
# Eulerian Edge Deletion— example



# Eulerian Edge Deletion— example



# Eulerian Edge Deletion— example



# Relaxing connectivity

- What, if we only require the graph after the deletion to have even degrees?

# Relaxing connectivity

- What, if we only require the graph after the deletion to have even degrees?
- Denote the set of vertices with odd degrees in $G$ by $T$.

# Relaxing connectivity

- What, if we only require the graph after the deletion to have even degrees?
- Denote the set of vertices with odd degrees in $G$ by $T$.
- We need to find a subset of edges of cardinality at most $k$, inducing a subgraph with exactly vertices from $T$ having odd degrees.

# Relaxing connectivity

- What, if we only require the graph after the deletion to have even degrees?
- Denote the set of vertices with odd degrees in $G$ by $T$.
- We need to find a subset of edges of cardinality at most $k$, inducing a subgraph with exactly vertices from $T$ having odd degrees.
- This is known as Minimal T-Join and can be done in polynomial time (using weighted matching in general graphs).

# Connectivity's back

- The problem: find a T-Join of cardinality at most $k$, such that the graph stays connected.

# Connectivity's back

- The problem: find a T-JOIN of cardinality at most $k$, such that the graph stays connected.
- Let $S \subseteq E$ be a solution, and $X$ be the set of endpoints of $S$. Then there exists a set of edges $W \subseteq E \setminus S$, such that $W$ connects all the vertices from $X$.

# Connectivity's back

- The problem: find a $\mathrm{T}$-Join of cardinality at most $k$, such that the graph stays connected.
- Let $S \subseteq E$ be a solution, and $X$ be the set of endpoints of $S$. Then there exists a set of edges $W \subseteq E \setminus S$, such that $W$ connects all the vertices from $X$.
- $W$ will be called a *witness*.

# Connectivity's back

- The problem: find a T-JOIN of cardinality at most $k$, such that the graph stays connected.
- Let $S \subseteq E$ be a solution, and $X$ be the set of endpoints of $S$. Then there exists a set of edges $W \subseteq E \setminus S$, such that $W$ connects all the vertices from $X$.
- $W$ will be called a *witness*.
- **Idea:** reformulate the problem — ask for a pair $(S, W)$, such that $S$ is a T-JOIN and $W$ is its witness.

# Dealing with the witness

- An edge is *close* iff at least one of its endpoints is in distance at most $k$ from the set $T$.

# Dealing with the witness

- An edge is *close* iff at least one of its endpoints is in distance at most $k$ from the set $T$.
- **Observation:** Solution can use only close edges.

# Dealing with the witness

- An edge is *close* iff at least one of its endpoints is in distance at most $k$ from the set $T$.
- **Observation:** Solution can use only close edges.
- Crucial Lemma:

**Witness Lemma**

If $S$ is a solution of size $k$, then $S$ has a witness $W$ with at most $(2k-1)(2k+2)$ close edges.

# Witness Lemma — proof [1/2]

- $X$, the set of endpoints of $S$, has cardinality at most $2k$.

# Witness Lemma — proof [1/2]

- $X$, the set of endpoints of $S$, has cardinality at most $2k$.
- We construct the witness $W$ inductively, beginning with the empty witness and every vertex from $X$ in a separate component.

# Witness Lemma — proof [1/2]

- $X$, the set of endpoints of $S$, has cardinality at most $2k$.
- We construct the witness $W$ inductively, beginning with the empty witness and every vertex from $X$ in a separate component.
- At each step we connect two different components of $W$ by a path — we choose the shortest among possible (disjoint with $S$) paths connecting two distinct components.

# Witness Lemma — proof [1/2]

- $X$, the set of endpoints of $S$, has cardinality at most $2k$.
- We construct the witness $W$ inductively, beginning with the empty witness and every vertex from $X$ in a separate component.
- At each step we connect two different components of $W$ by a path — we choose the shortest among possible (disjoint with $S$) paths connecting two distinct components.
- **Claim:** in this shortest path $P$ only first and last $k+1$ edges can be close.

# Witness Lemma — proof [2/2]

- Assume that there is a close edge among those in the middle of the path $P$.

# Witness Lemma — proof [2/2]

- Assume that there is a close edge among those in the middle of the path $P$.
- Then, one of its endpoints has to be in distance at most $k$ from some vertex from $T$, so it can be connected to some componenet by a path of lenght at most $k$, disjointfrom $S$.

# Witness Lemma — proof [2/2]

- Assume that there is a close edge among those in the middle of the path $P$.
- Then, one of its endpoints has to be in distance at most $k$ from some vertex from $T$, so it can be connected to some componenet by a path of lenght at most $k$, disjointfrom $S$.
- This component is a different from at least one of components we were to connect — contradiction with minimality of $P$.

# Witness Lemma — proof [2/2]

- Assume that there is a close edge among those in the middle of the path $P$.
- Then, one of its endpoints has to be in distance at most $k$ from some vertex from $T$, so it can be connected to some componenet by a path of lenght at most $k$, disjointfrom $S$.
- This component is a different from at least one of components we were to connect — contradiction with minimality of $P$.
- Hence, in each step we add at most $2k + 2$ close edges and the claim follows.

# Eulerian Edge Deletion— FPT algorithm

- Determine, which edges are close, and which are far.

# Eulerian Edge Deletion— FPT algorithm

- Determine, which edges are close, and which are far.
- Independently colour the edges:

# Eulerian Edge Deletion— FPT algorithm

- Determine, which edges are close, and which are far.
- Independently colour the edges:
  - every far edge is blue,

# Eulerian Edge Deletion— FPT algorithm

- Determine, which edges are close, and which are far.
- Independently colour the edges:
  - every far edge is blue,
  - every close edge is red with probability $\frac{1}{k^2}$ and blue otherwise.

# Eulerian Edge Deletion— FPT algorithm

- Determine, which edges are close, and which are far.
- Independently colour the edges:
  - every far edge is blue,
  - every close edge is red with probability $\frac{1}{k^2}$ and blue otherwise.
- If the whole $T$ is not in one connected component of the blue edges, return NO. Otherwise, name this component $K$

# Eulerian Edge Deletion— FPT algorithm

- Determine, which edges are close, and which are far.
- Independently colour the edges:
  - every far edge is blue,
  - every close edge is red with probability $\frac{1}{k^2}$ and blue otherwise.
- If the whole $T$ is not in one connected component of the blue edges, return NO. Otherwise, name this component $K$
- Solve Minimal T-Join in $G_R$, the graph induced by all the red edges with both endpoints in $K$. Return the answer.

# Eulerian Edge Deletion— analysis

- If the algorithm returns an answer to the Minimal T-Join, then $K$ is its witness.

# Eulerian Edge Deletion— analysis

- If the algorithm returns an answer to the Minimal T-Join, then $K$ is its witness.
- If there is a solution $(S, W)$, with probability at least $\frac{1}{k^{2k}}$ the whole $S$ is coloured red, and (independently) with probability at least $(1 - \frac{1}{k^2})^{(2k-1)(2k+2)} = \Omega(1)$ the whole $W$ is coloured blue.

# Eulerian Edge Deletion— analysis

- If the algorithm returns an answer to the Minimal T-Join, then $K$ is its witness.
- If there is a solution $(S, W)$, with probability at least $\frac{1}{k^{2k}}$ the whole $S$ is coloured red, and (independently) with probability at least $(1 - \frac{1}{k^2})^{(2k-1)(2k+2)} = \Omega(1)$ the whole $W$ is coloured blue.
- Therefore, if a solution exists, it will be found with probability at least $\frac{C}{k^{2k}}$ for some constant $C$. We can run the algorithm $O(k^{2k})$ times in order to reduce the probability of not founding the solution to constant.

# Eulerian Edge Deletion— analysis

- If the algorithm returns an answer to the Minimal T-Join, then $K$ is its witness.
- If there is a solution $(S, W)$, with probability at least $\frac{1}{k^{2k}}$ the whole $S$ is coloured red, and (independently) with probability at least $(1 - \frac{1}{k^2})^{(2k-1)(2k+2)} = \Omega(1)$ the whole $W$ is coloured blue.
- Therefore, if a solution exists, it will be found with probability at least $\frac{C}{k^{2k}}$ for some constant $C$. We can run the algorithm $O(k^{2k})$ times in order to reduce the probability of not founding the solution to constant.
- **Derandomization:** similar to Longest Path example.

# Open problems

- Biclique
  Problem: Is there a subgraph $K_{t,t}$ of the given graph?
  Parameter: $t$
  Is it FPT?

# Open problems

- Biclique
  Problem: Is there a subgraph $K_{t,t}$ of the given graph?
  Parameter: $t$
  Is it FPT?
- Odd Cycle Transversal
  Problem: delete at most $k$ vertices in order to obtain a bipartite graph.
  Parameter: $k$
  Is there a polykernel?

# Thank you for your attention

Questions?