Improved Edge Coloring with Three Colors

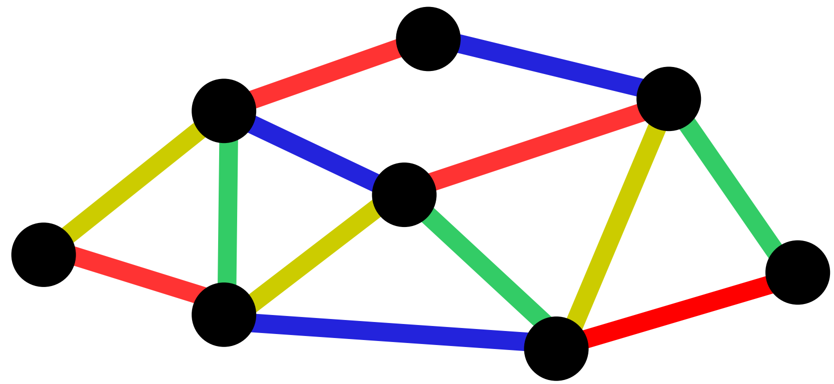
Łukasz Kowalik

Max Planck Institut für Informatik, Saarbrücken

Instytut Informatyki, Warsaw University

Edge-Coloring

Assign colors to edges so that incident edges get distinct colors.

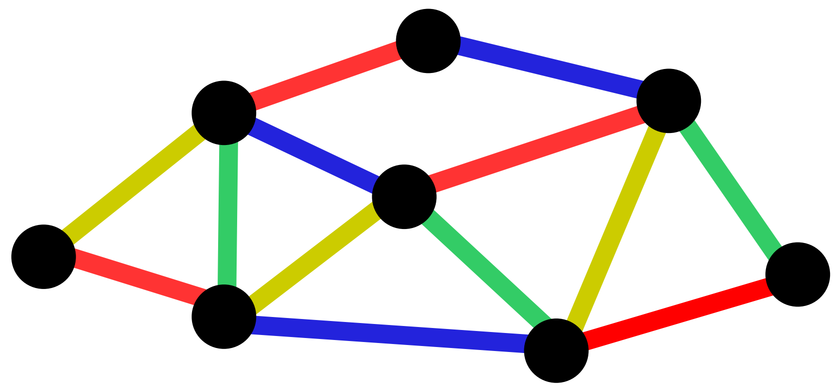


What is known? ($\Delta = \max_v \mathrm{degree}(v)$)

- Δ colors needed (trivial)
- $\Delta + 1$ colors suffice (Vizing)
- Deciding "$\Delta/(\Delta+1)$" is NP-complete even when $\Delta = 3$.

Edge-Coloring

Assign colors to edges so that incident edges get distinct colors.



What is known? ($\Delta = \max_v \mathrm{degree}(v)$)

- Δ colors needed (trivial)
- $\Delta + 1$ colors suffice (Vizing)
- Deciding "$\Delta / (\Delta + 1)$" is NP-complete even when $\Delta = 3$.

We will focus on the $\Delta = 3$ case (subcubic graphs).

3-Edge-Coloring: Results

Let G be the input graph, $n = |V(G)|$.

- Naive backtracking: $O(2^{|E(G)|}) = O(2^{3/2n}) = O(2.83^n)$.
- Approach: vertex-coloring the line graph $L(G)$.
  3-coloring algorithm by Beigel & Eppstein [JAlg'05] gives time:
  $O(1.3289^{|V(L(G))|}) = O(1.3289^{|E(G)|}) = O(1.532^n)$.
- (for $\geq$ 4 colors the above approach is the best known.)
- Beigel & Eppstein [JAlg'05]: nontrivial preprocessing + reduction to $(3, 2)$-CSP.
  Time: $O(1.415^n) = O(2^{n/2})$.

3-Edge-Coloring: Results

Let G be the input graph, $n = |V(G)|$.

- Naive backtracking: $O(2^{|E(G)|}) = O(2^{3/2n}) = O(2.83^n)$.
- Approach: vertex-coloring the line graph $L(G)$.
  3-coloring algorithm by Beigel & Eppstein [JAlg'05] gives time:
  $O(1.3289^{|V(L(G))|}) = O(1.3289^{|E(G)|}) = O(1.532^n)$.
- (for $\geq$ 4 colors the above approach is the best known.)
- Beigel & Eppstein [JAlg'05]: nontrivial preprocessing + reduction to $(3, 2)$-CSP.
  Time: $O(1.415^n) = O(2^{n/2})$.
- This work: $O(1.344^n) = O(2^{0.427n})$

Basic Idea

(Counterpart of Lawler's '76 algorithm for 3-vertex-coloring)

A matching M in graph G is *fitting* when $G - M$ is 2-edge-colorable.

- G is 3-edge-colorable iff G contains a fitting matching.
- G is 3-edge-colorable iff G contains a **(inclusion-wise) maximal** matching which is fitting.
- 2-edge-colorability is in P.

Algorithm 1: generate all maximal matchings, for each verify whether it is fitting.

Basic Idea Refined

Observation: Fitting matching matches every 3-vertex.

A matching which matches every 3-vertex will be called *semi-perfect*.

Algorithm 2: generate all maximal semi-perfect matchings, for each verify whether it is fitting.

Basic Idea Refined

Observation: Fitting matching matches every 3-vertex.

A matching which matches every 3-vertex will be called *semi-perfect*.

Algorithm 2: generate all maximal semi-perfect matchings, for each verify whether it is fitting.

Observation: Good for cubic graphs.

Conclusion: Reduce 3-edge-coloring for subcubic graphs to 3-edge-coloring in graphs “close to cubic”...

Basic Idea Refined

Observation: Fitting matching matches every 3-vertex.

A matching which matches every 3-vertex will be called *semi-perfect*.

Algorithm 2: generate all maximal semi-perfect matchings, for each verify whether it is fitting.

Observation: Good for cubic graphs.

Conclusion: Reduce 3-edge-coloring for subcubic graphs to 3-edge-coloring in graphs “close to cubic”...

= *semi-cubic*: vertices of degree 2 and 3, distance between 2-vertices at least 3

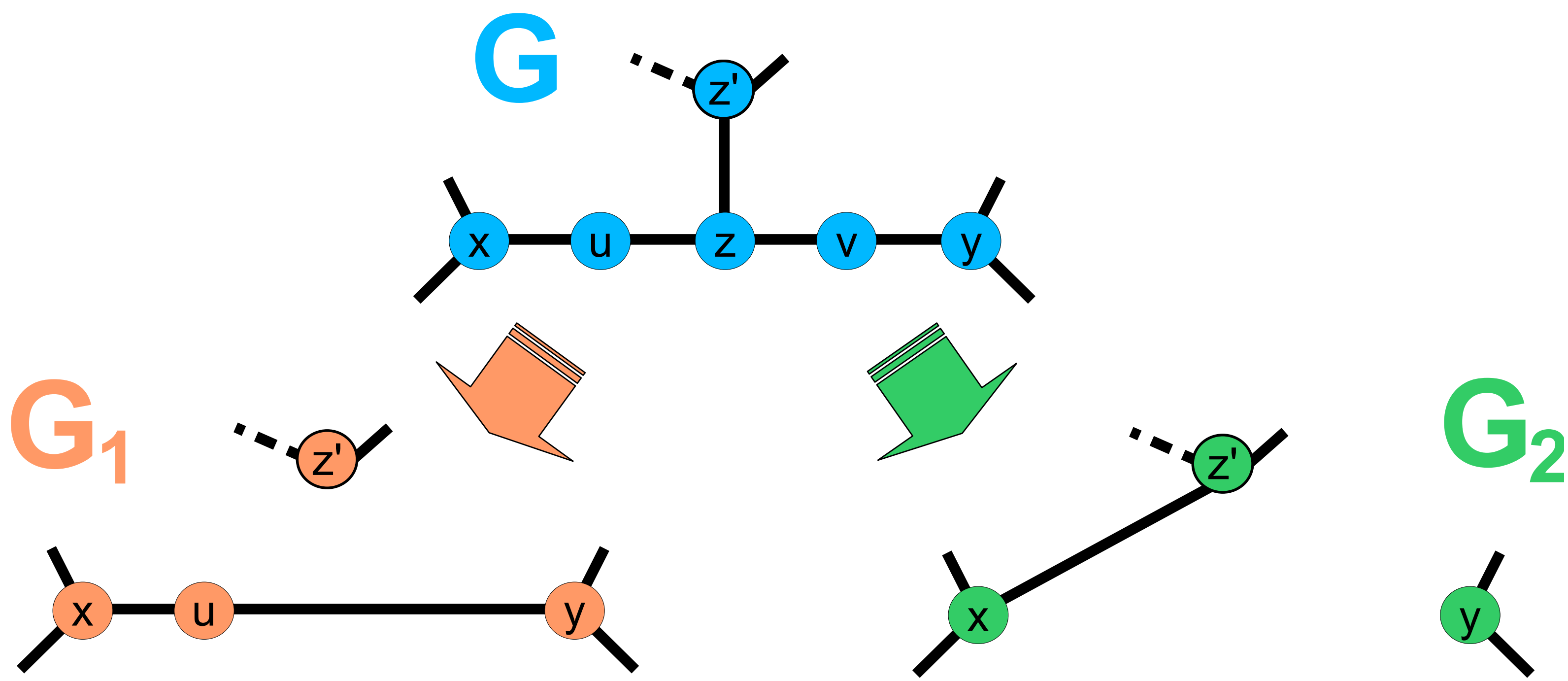
Reducing to a semi-cubic graph

Let G be the input graph.

- Assume G contains a 1-vertex v. Then G is 3-edge-colorable iff $G - v$ is 3-edge-colorable.
- Assume G contains an edge uv, $\deg(u) = \deg(v) = 2$. Then G is 3-edge-colorable iff $G - uv$ is 3-edge-clrble.

Reducing to a semi-cubic graph, contd.

- Assume G contains a path $xuzvy$, $\deg(u) = \deg(v) = 2$.

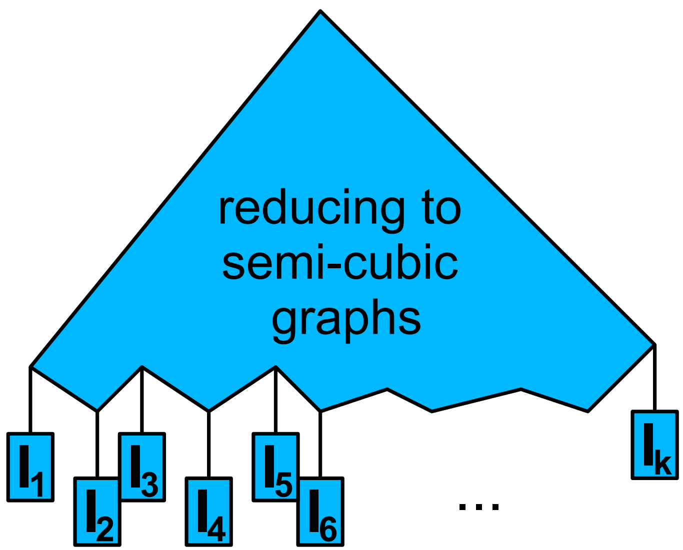


G is 3-edge-colorable iff G_1 or G_2 is 3-edge-colorable.

How expensive is it? $T(n) = T(n-2) + T(n-3) + \mathrm{poly}(n)$, so $T(n) = O(1.325^n)$

Reducing to a semi-cubic graph, contd.

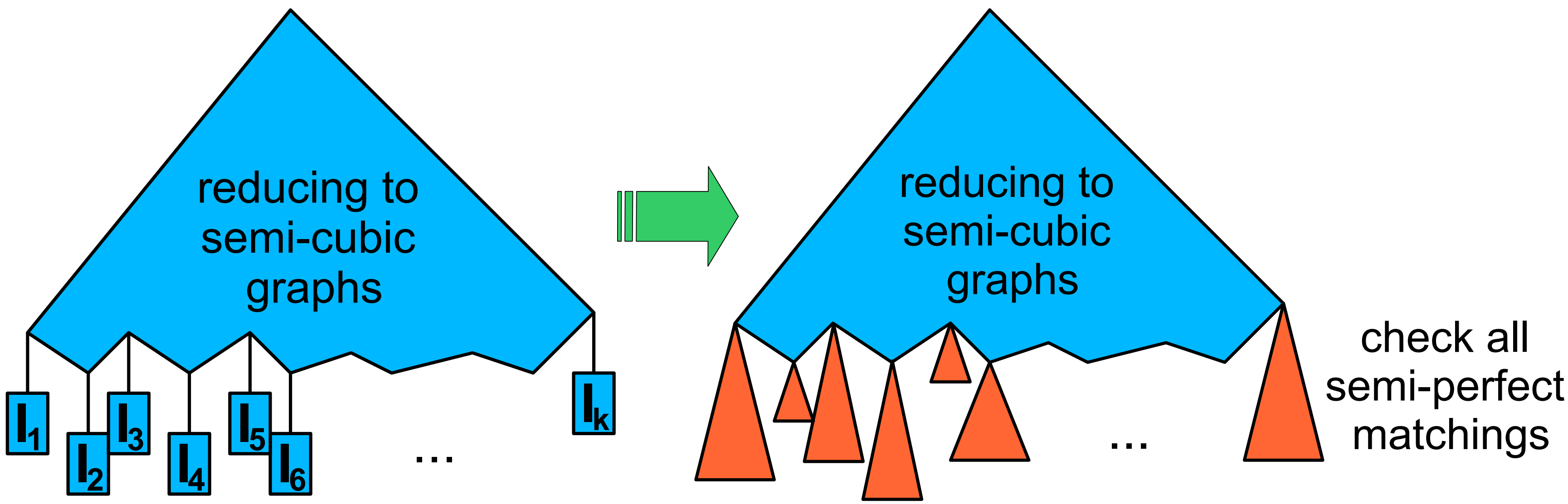
We get a recursion tree:



Each instance I_j is a semi-cubic graph.

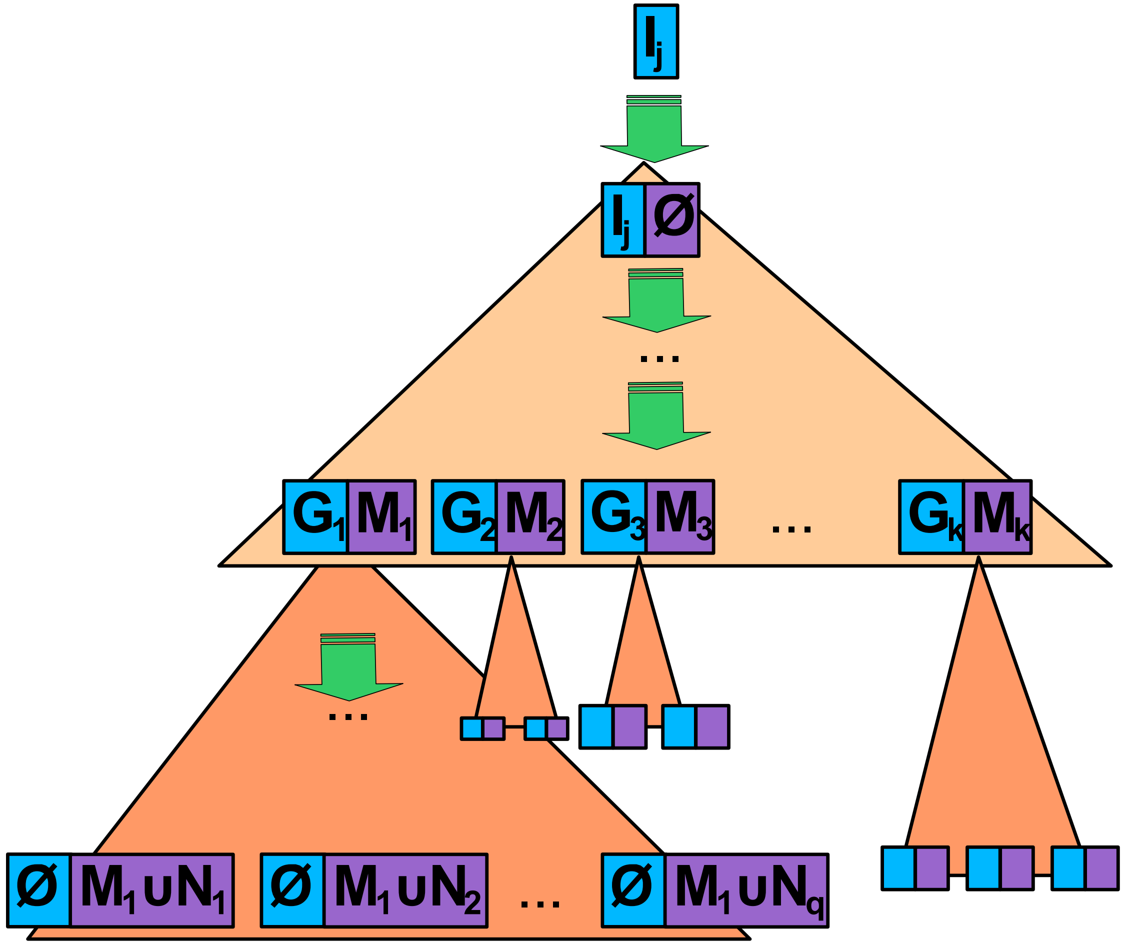
Reducing to a semi-cubic graph, contd.

We get a recursion tree:



Each instance I_j is a semi-cubic graph.
In each I_j we want to check all semi-perfect matchings.

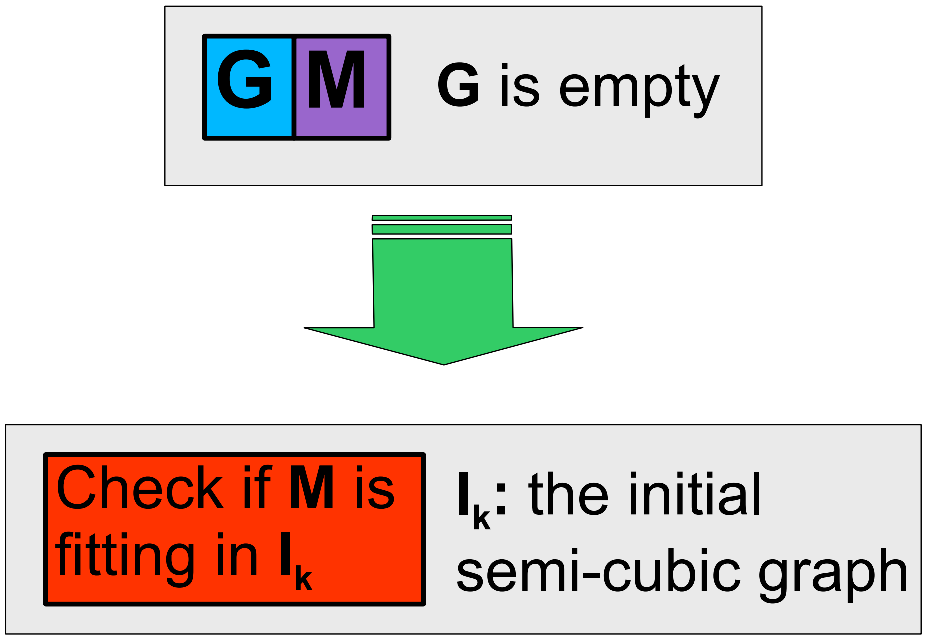
Checking all semi-perfect matchings



The recursion tree rooted at 

generates all semi-perfect matchings that extend M_i using edges from G_i (e.g. $N_q \subset E(G_1)$).

Base Case

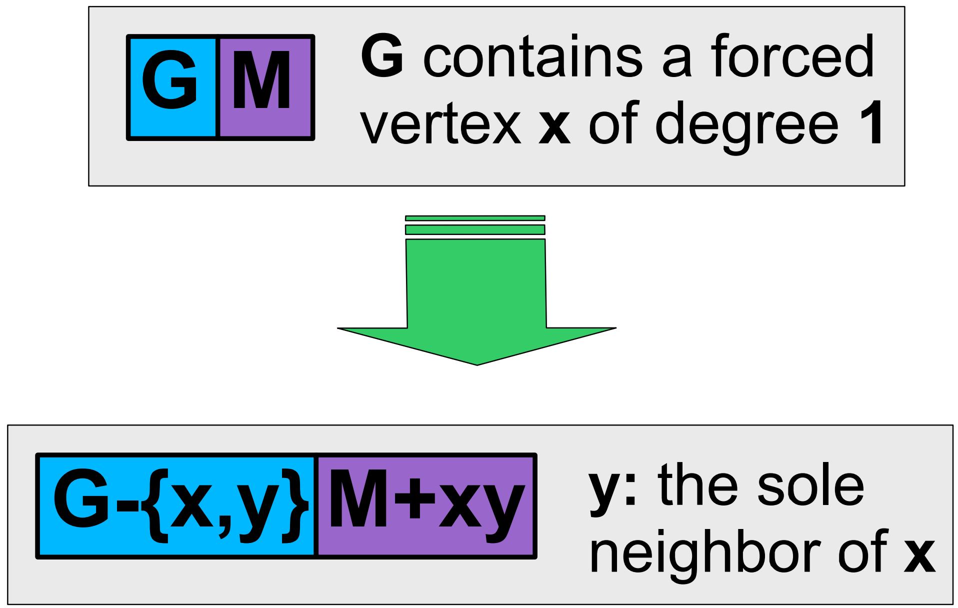

G
M
G is empty
Check if **M** is fitting in I_k
I_k: the initial semi-cubic graph

Forced and Unforced Vertices

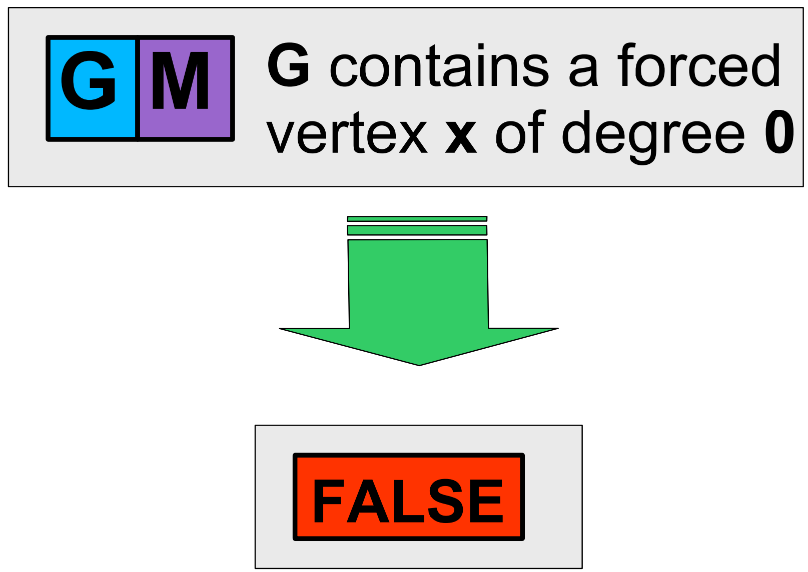
Let I be the initial semi-cubic graph in which we generate semi-perfect matchings.

- a vertex of degree 3 will be called *forced*.
- other vertices (of degree 2) are *unforced*.

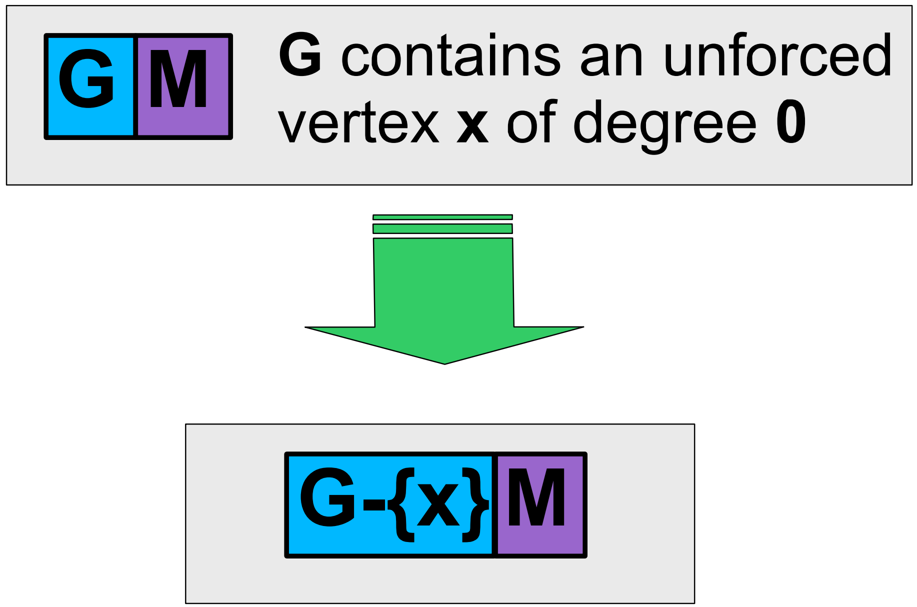
Trivial Case 1



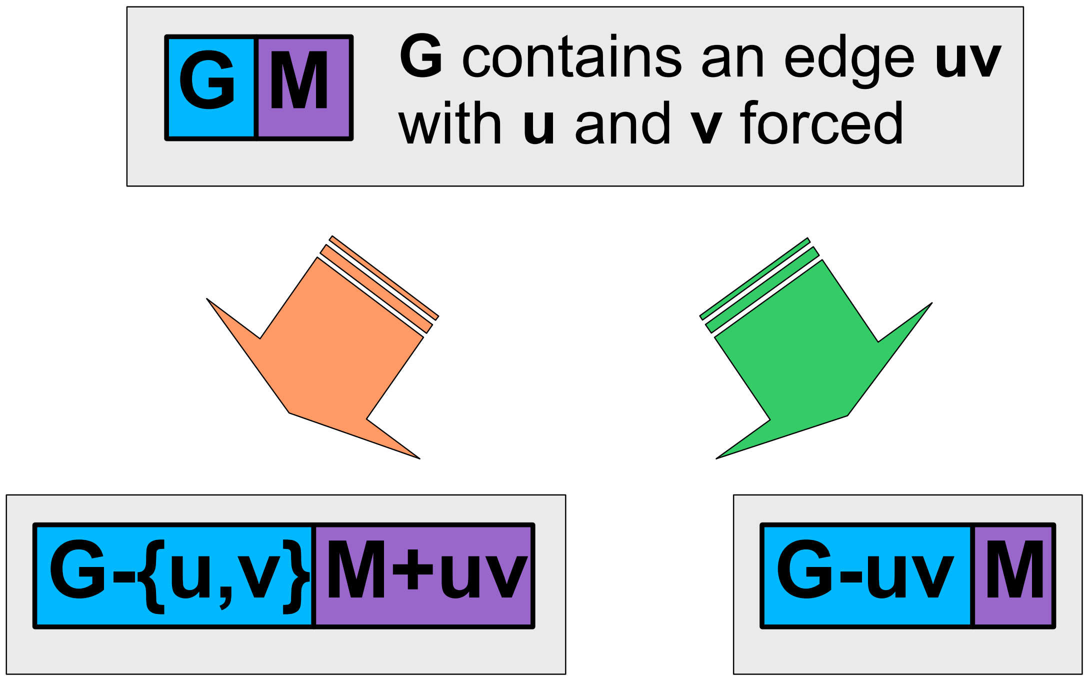
Trivial Case 2



Trivial Case 3



Branching



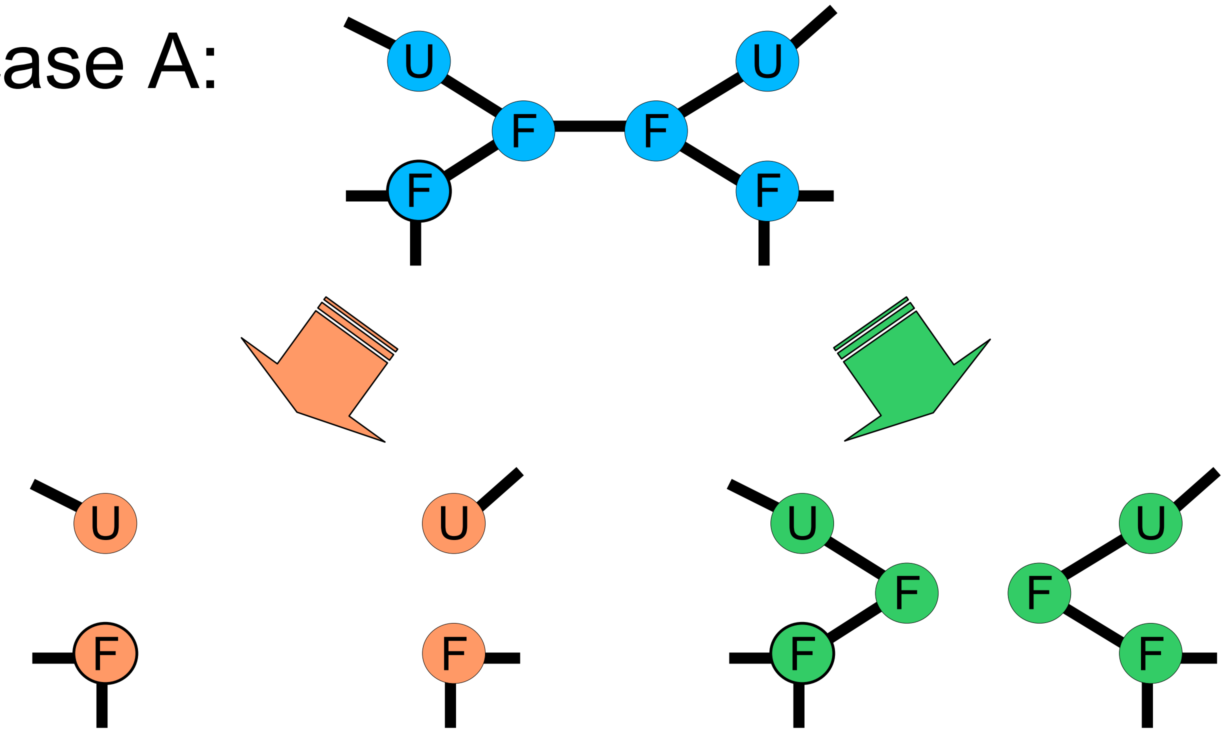
Checking all semi-perfect matchings

procedure FITTINGMATCH(I,G,M)

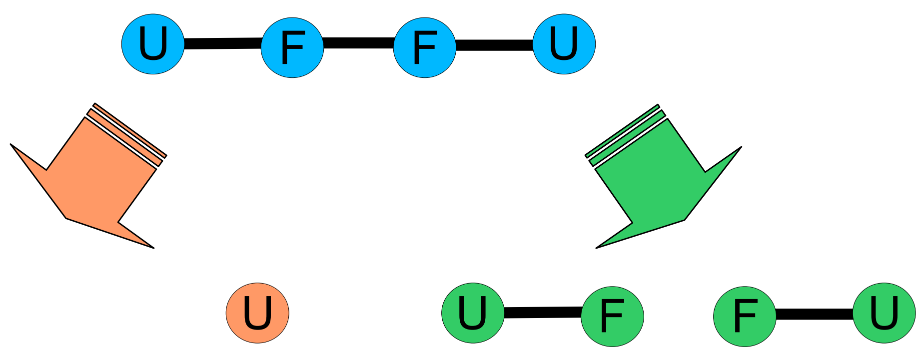
if $V(G) = \emptyset$ **then**
    if M is fitting in I **then return** TRUE **else return** FALSE
else if exists a forced vertex $v \in V(G)$ such that $\deg_G(v) = 0$ **then**
    return FALSE
else if exists a non-forced vertex $v \in V(G)$ such that $\deg_G(v) = 0$ **then**
    return FITTINGMATCH(I, $G - \{v\}$, M)
else if exists a forced vertex $v \in V(G)$ such that $\deg_G(v) = 1$ **then**
    $u \leftarrow$ the neighbor of v in G
    return FITTINGMATCH(I, $G - \{u, v\}$, $M \cup \{uv\}$)
else
    $uv \leftarrow$ any edge in G with both ends forced.
    return FITTINGMATCH(I, $G - \{u, v\}$, $M \cup \{uv\}$) **or** FITTINGMATCH(I, $G - uv$, M)

Two sample cases of branching

case A:



case B:



One more trick (details skipped)

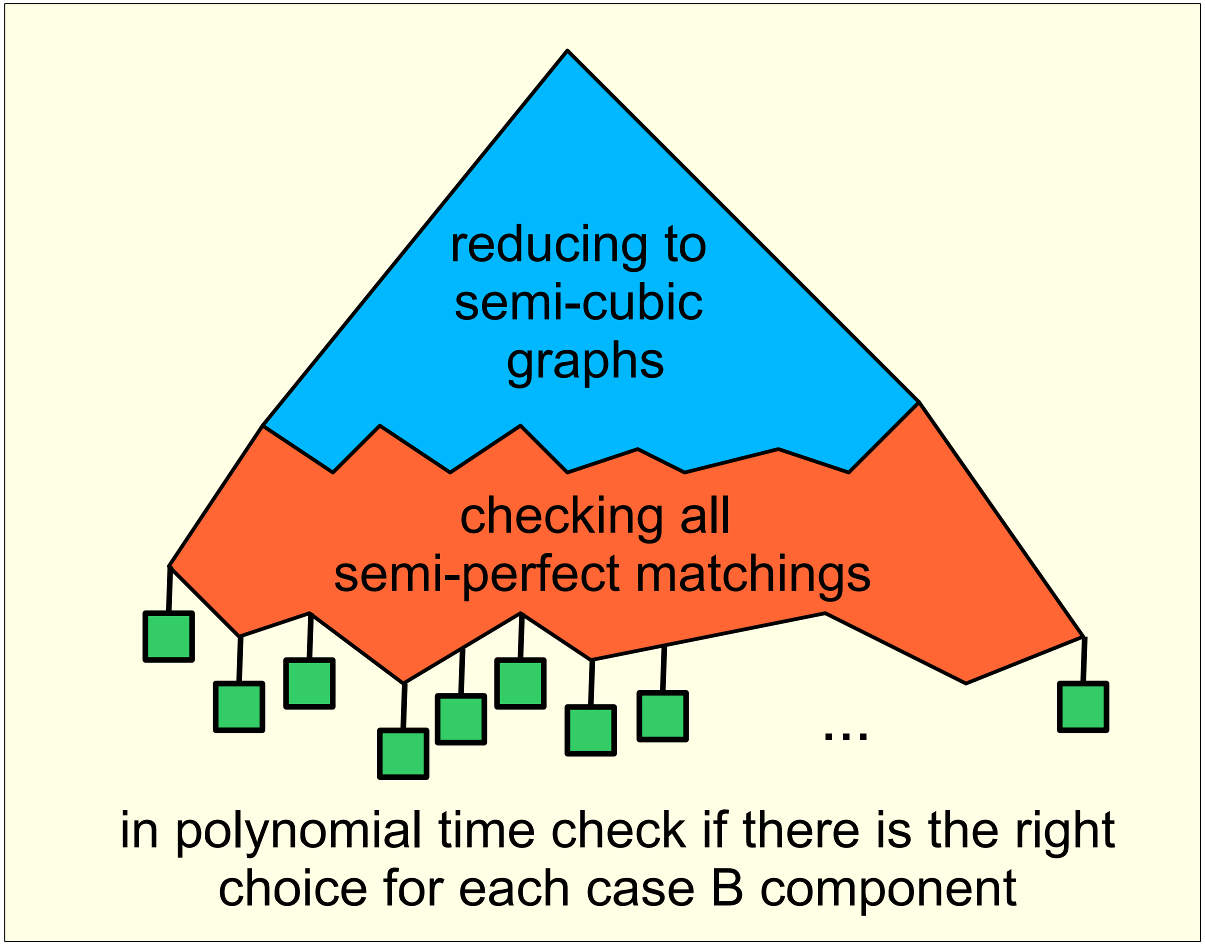
G | **M** — Each connected component of **G** is a path from case B

↓

Check in poly time if **M** extends to a fitting matching in $\mathbf{I_k}$ (for each case B component find *the right choice* if it exists)

$\mathbf{I_k}$**:** the initial semi-cubic graph

The full picture



Instances in the leaves are triples (G_0, G, M) such that G is a collection of 4-paths from case B.

Conclusion

To sum up:

- Time complexity is $O(1.344^n)$,
- Space complexity is $O(n)$,
- the algorithm is simple to implement,
- main ingredients:
  - “cheap” reduction to instances of special structure,
  - solving special cases polynomially,
  - “measure and conquer” technique for analysis.