# Addressing Bias in Algorithmic Solutions: Exploring Vertex Cover and Feedback Vertex Set


Sheikh Shakil Akhtar

Chennai Mathematical Institute


December 6, 2024

Joint work with Jayakrishnan Madathil, Pranabendu Misra and Geevarghese Philip

# What is Parameterised Complexity

Parameterised Complexity is a paradigm of algorithm design used for tackling NP-COMPLETENESS.

# What is Parameterised Complexity

Parameterised Complexity is a paradigm of algorithm design used for tackling NP-COMPLETENESS.

P.C.

## What do we know about NP-COMPLETENESS

For NP-COMPLETE problems, in *general instances*, one does not **expect** *exact deterministic algorithms* which run in *polynomial time*.

approximation

randomisation

Parameter

↓

V.F.

Exact Expon.

$(1.6)^n \leftarrow 2^n$

$(\sqrt{n})^n \leftarrow n^n$

# What is Parameterised Complexity

Parameterised Complexity is a paradigm of algorithm design used for tackling NP-COMPLETENESS.

## What do we know about NP-COMPLETENESS

For NP-COMPLETE problems, in *general instances*, one does not **expect** *exact deterministic algorithms* which run in *polynomial time*.

What if we relax the above requirments?

# What is Parameterised Complexity

Parameterised Complexity is a paradigm of algorithm design used for tackling NP-COMPLETENESS.

**What do we know about NP-COMPLETENESS**

For NP-COMPLETE problems, in *general instances*, one does not **expect** *exact deterministic algorithms* which run in *polynomial time*.

What if we relax the above requirments?
Parameterised Complexity deals with algorithms which run efficiently for some instances.

# Fixed-Paramter Tractable

$G, k$ $|V.C.| \leq k$

## Definition of a parameterised problem

A *parameterised problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where $\Sigma$ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, $k$ is called the *parameter*.

P FPT

## Definition of a parameterised problem

A *parameterised problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where $\Sigma$ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, $k$ is called the *parameter*.

## FIXED-PARAMTER TRACTABLE

A parameterised problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (FPT) if there exists an algorithm $\mathcal{A}$ (called a *fixed-parameter algorithm*), a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant $c$ such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

$\Pi$ $G$ $n$ $k$ $\Pi \in FPT$ $n^c \times f(k)$

$f(k) \le n^c$

$t$ $\quad [t] := \{1, 2, \ldots, t\}$

Let $t$ be constant positive integer. We will refer to members of $\{1, 2, \ldots, t\}$ as colours. A graph $G$ is said to be *$t$-coloured* if there exists a function $c : V(G) \to 2^{\{1,2,\ldots,t\}} \setminus \emptyset$. Given a $t$-tuple of non-negative integers, $\mathbb{T} = (k_i)_{i=1}^{t}$, a set $S \subseteq V(G)$ is said to be $\mathbb{T}$-fair if for each $i \in [t]$, we have $|\{v \in S | i \in c(v)\}| = k_i$.

$\mathbb{T} = (k_1, k_2, \ldots, k_t)$

2 $\quad$ 3, 4

$S = \{v_1, v_2, \ldots, v_{10}\}$

$k_1$, $k_2$

# $\mathbb{T}-$ Fair Problems(contd.)

We will discuss the following problems

$\mathbb{T}-$fair Vertex Cover
Input: An undirected $t$-coloured graph $G = (V, E)$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: $\Sigma_{i=1}^{t} k_i$
Question: Does $G$ have a $\mathbb{T}-$fair vertex cover?

We will discuss the following problems

$\mathbb{T}-$fair Vertex Cover
Input: An undirected $t$-coloured graph $G = (V, E)$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: $\Sigma_{i=1}^{t} k_i$
Question: Does $G$ have a $\mathbb{T}-$fair vertex cover?

$\mathbb{T}-$fair Feedback Vertex Set
Input: An undirected $t$-coloured graph $G = (V, E)$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: $\Sigma_{i=1}^{t} k_i$
Question: Does $G$ have a $\mathbb{T}-$fair feedback vertex set?

# Kernelisation

$\Pi$. $(G, k) \xrightarrow{\mathcal{A}} (G', k')$ ← kernel

A *kernelisation algorithm* for a parameterised problem $\Pi$ is a deterministic algorithm $\mathcal{A}$ that, given an instance $(I, k)$ of $\Pi$, works in polynomial time and returns an equivalent instance $(I', k')$, such that $|I'| + k' \leq g(k)$, where $g : \mathbb{N} \rightarrow \mathbb{N}$ is a computable function. The instance $(I', k')$ is called a *kernel*.

$\mathcal{A}$ in $\underline{n^{O(1)}}$ time

$|G'| + k' \leq g(k)$

FPT

$T(k)$

$n^{O(1)} + T(k) \leq \underline{n^{O(1)} \cdot T(k)}$

# Kernelisation

A *kernelisation algorithm* for a parameterised problem $\Pi$ is a deterministic algorithm $\mathcal{A}$ that, given an instance $(I, k)$ of $\Pi$, works in polynomial time and returns an equivalent instance $(I', k')$, such that $|I'| + k' \leq g(k)$, where $g : \mathbb{N} \to \mathbb{N}$ is a computable function. The instance $(I', k')$ is called a *kernel*.
(Intuitively speaking, a kernel is a "smaller" equivalent instance, where one can afford to use brute force algorithms)

# Kernelisation

$n \longrightarrow g(k)$

A *kernelisation algorithm* for a parameterised problem $\Pi$ is a deterministic algorithm $\mathcal{A}$ that, given an instance $(I, k)$ of $\Pi$, works in polynomial time and returns an equivalent instance $(I', k')$, such that $|I'| + k' \leq g(k)$, where $g : \mathbb{N} \rightarrow \mathbb{N}$ is a computable function. The instance $(I', k')$ is called a *kernel*.

(Intuitively speaking, a kernel is a "smaller" equivalent instance, where one can afford to use brute force algorithms)

**Theorem**

*A problem $\Pi$ is in* FPT *if and only if it admits a kernelisation algorithm.*

Hint $\underline{n^c \cdot f(k)}$ $A$ $n^{c+1}$

# Polynomial Kernel for $\mathbb{T}-$fair Vertex Cover

We apply the following rules once at the beginning.

# Polynomial Kernel for $\mathbb{T}-$FAIR VERTEX COVER

We apply the following rules once at the beginning.

- If $|\{v \in V(G) | c(i) \in v\}| < k_i$ holds for at least one $i \in [t]$ then return NO.

# Polynomial Kernel for $\mathbb{T}$−fair Vertex Cover


$k_B = 2$

$\sum k_i \leq t$

We apply the following rules once at the beginning.

- If $|\{v \in V(G) | c(i) \in v\}| < k_i$ holds for at least one $i \in [t]$ then return No.
- Let $k_{max} = \max_{1 \leq i \leq t} k_i$. For each non-empty $X \subseteq [t]$, $V_X = \{v \in V(G) | deg(v) = 0 \wedge c(v) = X\}$. If $|V_X| > k_{max}$, then keep any $k_{max}$ of them and remove the rest from $V_X$. Finally, let $I^* = \cup_{X \subseteq [t] \wedge X \neq \emptyset} V_X$.

$|I^*| \leq k_{max} 2^t \leq \sum k_i 2^t$

We apply the following rules once at the beginning.

- If $|\{v \in V(G)|c(i) \in v\}| < k_i$ holds for at least one $i \in [t]$ then return NO.
  $\geq m_i$
- Let $k_{max} = \max_{1 \leq i \leq t} k_i$. For each non-empty $X \subseteq [t]$, $V_X = \{v \in V(G)|deg(v) = 0 \wedge c(v) = X\}$. If $|V_X| > k_{max}$, then keep any $k_{max}$ of them and remove the rest from $V_X$. Finally, let $I^* = \cup_{X \subseteq [t] \wedge X \neq \emptyset} V_X$.

We then apply the following rules in order, until they can't be applied any more.

# Polynomial Kernel for $\mathbb{T}-$FAIR VERTEX COVER(contd.)

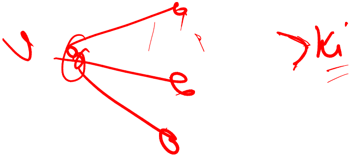
- $\exists(i,j) \in [t] \times [t]$, such that $(i) k_i = k_j = 0$ and $(ii) \exists uv \in E(G)$, such that $i \in c(u) \wedge j \in c(v)$. Return NO.

# Polynomial Kernel for $\mathbb{T}-$FAIR VERTEX COVER(contd.)

- $\exists(i,j) \in [t] \times [t]$, such that $(i) k_i = k_j = 0$ and $(ii) \exists uv \in E(G)$, such that $i \in c(u) \wedge j \in c(v)$. Return NO.
- $\exists v \in V(G), deg(v) = 0 \wedge v \notin I^*$, then we return the instance $(G - v, c', \mathbb{T})$, where $c'$ is the restriction of $c$ on $V(G) \setminus \{v\}$.

# Polynomial Kernel for $\mathbb{T}$−FAIR VERTEX COVER(contd.)

- $\exists (i,j) \in [t] \times [t]$, such that $(i) k_i = k_j = 0$ and $(ii) \exists uv \in E(G)$, such that $i \in c(u) \wedge j \in c(v)$. Return NO.
- $\exists v \in V(G), deg(v) = 0 \wedge v \notin I^*$, then we return the instance $(G - v, c', \mathbb{T})$, where $c'$ is the restriction of $c$ on $V(G) \setminus \{v\}$.
- $\exists v \in V(G)$, such that for some $i \in [t], |\{w \in N(v) | i \in c(w)\}| > k_i$, then return $(G - v, c', k'_i)$, where $c'$ is the restriction of $c$ on $V(G) \setminus \{v\}$ and for each $i \in [t], k'_i = k_i - |\{w \in \{v\} | i \in c(w)\}|$.



# Polynomial Kernel for $\mathbb{T}$−fair Vertex Cover(contd.)

- $\exists(i,j) \in [t] \times [t]$, such that $(i) k_i = k_j = 0$ and $(ii) \exists uv \in E(G)$, such that $i \in c(u) \wedge j \in c(v)$. Return No.
- $\exists v \in V(G), deg(v) = 0 \wedge v \notin I^*$, then we return the instance $(G - v, c', \mathbb{T})$, where $c'$ is the restriction of $c$ on $V(G) \setminus \{v\}$.
- $\exists v \in V(G)$, such that for some $i \in [t], |\{w \in N(v) | i \in c(w)\}| > k_i$, then return $(G - v, c', k_i')$, where $c'$ is the restriction of $c$ on $V(G) \setminus \{v\}$ and for each $i \in [t], k_i' = k_i - |\{w \in \{v\} | i \in c(w)\}|$.

## Return a kernel

If none of the above rules are applicable, then return No, if $|V(G)| > (\sum_{i=1}^{t} k_i)^2 + \sum_{i=1}^{t} k_i \times (1 + 2^t)$ or $|E(G)| > (\sum_{i=1}^{t} k_i)^2$. Otherwise, we return the instance $(G, c, \mathbb{T})$.

$|F| = k$ $\qquad$ $|V(G - F)| \leq K|F|$ $\qquad$ $|V(G)| \leq k^2 + k$

$\neq$ V.C. $\qquad$ $|E(G)| \leq k^2$

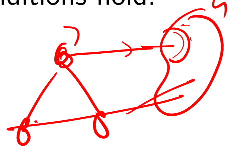
# Tree Decomposition of a Graph

Given a graph, its *treewidth* is a measure of how "tree-like" it is. Forests have treewidth 1, whereas cycles have treewidth 2.

$n-1$

# Tree Decomposition of a Graph

Given a graph, its *treewidth* is a measure of how "tree-like" it is. Forests have treewidth 1, whereas cycles have treewidth 2.
Formally, a tree decomposition of a graph $G$ is a pair $(T, \phi)$, where $T$ is a tree and $\phi : V(T) \to 2^{V(G)}$, such that the following conditions hold:



# Tree Decomposition of a Graph

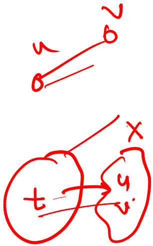
Given a graph, its *treewidth* is a measure of how "tree-like" it is. Forests have treewidth 1, whereas cycles have treewidth 2.
Formally, a tree decomposition of a graph $G$ is a pair $(T, \phi)$, where $T$ is a tree and $\phi : V(T) \to 2^{V(G)}$, such that the following conditions hold:

- $\cup_{t \in V(T)} \phi(t) = V(G)$

# Tree Decomposition of a Graph

Given a graph, its *treewidth* is a measure of how “tree-like” it is. Forests have treewidth 1, whereas cycles have treewidth 2.
Formally, a tree decomposition of a graph $G$ is a pair $(T, \phi)$, where $T$ is a tree and $\phi : V(T) \to 2^{V(G)}$, such that the following conditions hold:

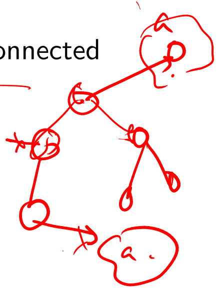
- $\cup_{t \in V(T)} \phi(t) = V(G)$
- $\forall uv \in E(G),\ \exists t \in V(T)$ such that $\{u, v\} \subseteq \phi(t)$



# Tree Decomposition of a Graph

Given a graph, its *treewidth* is a measure of how "tree-like" it is. Forests have treewidth 1, whereas cycles have treewidth 2.
Formally, a tree decomposition of a graph $G$ is a pair $(T, \phi)$, where $T$ is a tree and $\phi : V(T) \rightarrow 2^{V(G)}$, such that the following conditions hold:

- $\cup_{t \in V(T)} \phi(t) = V(G)$
- $\forall uv \in E(G),\ \exists t \in V(T)$ such that $\{u, v\} \subseteq \phi(t)$
- $\forall v \in V(G)$, the set $T_u := \{t \in V(T) | v \in \phi(t)\}$ is connected



# Tree Decomposition of a Graph

Given a graph, its *treewidth* is a measure of how "tree-like" it is. Forests have treewidth 1, whereas cycles have treewidth 2.
Formally, a tree decomposition of a graph $G$ is a pair $(T, \phi)$, where $T$ is a tree and $\phi : V(T) \to 2^{V(G)}$, such that the following conditions hold:

- $\cup_{t \in V(T)} \phi(t) = V(G)$
- $\forall uv \in E(G),\ \exists t \in V(T)$ such that $\{u, v\} \subseteq \phi(t)$
- $\forall v \in V(G)$, the set $T_u := \{t \in V(T) | v \in \phi(t)\}$ is connected

node

For any $t \in V(T)$, $\phi(t)$ is called the bag of the node $t$. The width of tree decomposition $(T, \phi)$ equals $\max_{t \in V(T)} |\phi(t)| - 1$, that is, the maximum size of its bag minus 1. The treewidth of a graph $G$, denoted by $tw(G)$, is the minimum possible width of a tree decomposition of $G$.

# Nice Tree Decomposition

For the purpose of designing algorithms we consider *nice tree decomposition* of a graph $G$. A tree decomposition $(T, \phi)$ of a graph $G$ is called *nice* if the following conditions hold:

- The tree $T$ is rooted at a node, say $r$.

# Nice Tree Decomposition

For the purpose of designing algorithms we consider *nice tree decomposition* of a graph $G$. A tree decomposition $(T, \phi)$ of a graph $G$ is called *nice* if the following conditions hold:

- The tree $T$ is rooted at a node, say $r$.
- $\phi(r) = \emptyset$.

# Nice Tree Decomposition

For the purpose of designing algorithms we consider *nice tree decomposition* of a graph $G$. A tree decomposition $(T, \phi)$ of a graph $G$ is called *nice* if the following conditions hold:

- The tree $T$ is rooted at a node, say $r$.
- $\phi(r) = \emptyset$.
- $\phi(l) = \emptyset$ for every leaf $l$ of $T$.

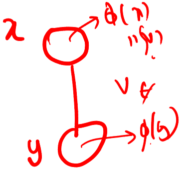
# Nice Tree Decomposition

For the purpose of designing algorithms we consider *nice tree decomposition* of a graph $G$. A tree decomposition $(T, \phi)$ of a graph $G$ is called *nice* if the following conditions hold:

- The tree $T$ is rooted at a node, say $r$.
- $\phi(r) = \emptyset$.
- $\phi(l) = \emptyset$ for every leaf $l$ of $T$.
- Every non-leaf node is one of the following types.

# Nice Tree Decomposition

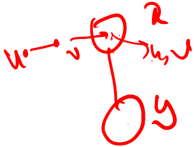
For the purpose of designing algorithms we consider *nice tree decomposition* of a graph $G$. A tree decomposition $(T, \phi)$ of a graph $G$ is called *nice* if the following conditions hold:




- The tree $T$ is rooted at a node, say $r$.
- $\phi(r) = \emptyset$.
- $\phi(l) = \emptyset$ for every leaf $l$ of $T$.
- Every non-leaf node is one of the following types.
    - **Introduce vertex node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y) \cup \{v\}$ for some $v \notin \phi(y)$; we say that $v$ is *introduced* at $x$.

# Nice Tree Decomposition

For the purpose of designing algorithms we consider *nice tree decomposition* of a graph $G$. A tree decomposition $(T, \phi)$ of a graph $G$ is called *nice* if the following conditions hold:



- The tree $T$ is rooted at a node, say $r$.
- $\phi(r) = \emptyset$.
- $\phi(l) = \emptyset$ for every leaf $l$ of $T$.
- Every non-leaf node is one of the following types.
  - **Introduce vertex node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y) \cup \{v\}$ for some $v \notin \phi(y)$; we say that $v$ is *introduced* at $x$.
  - **Introduce edge node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y)$ and for $u, v \in \phi(x)$, where $uv \in E(G)$ we label the node $x$ with $uv$; we say that $uv$ is *introduced* at $x$.

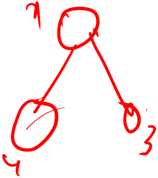
# Nice Tree Decomposition(contd.)

- **Forget node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y) \setminus \{v\}$ for some $v \in \phi(y)$; we say that $v$ is *forgotten* at $x$.

# Nice Tree Decomposition(contd.)

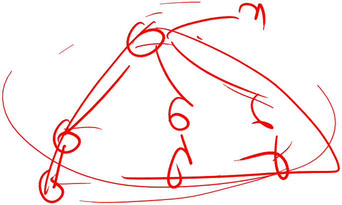
- **Forget node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y) \setminus \{v\}$ for some $v \in \phi(y)$; we say that $v$ is *forgotten* at $x$.
- **Join node**: This is a node $x$ of $T$ with exactly two children $y$ and $z$, such that $\phi(x) = \phi(y) = \phi(z)$.

For a node $x$ of $T$, we define $G_x = (V_x, E_x)$ as follows:




- $V_x = \cup_{y \text{ is a descendant of } x} \phi(y)$
- $E_x = \{$All edges introduced at $x$ and its descendants$\}$

$G_x$



# Nice Tree Decomposition(contd.)

- **Forget node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y) \setminus \{v\}$ for some $v \in \phi(y)$; we say that $v$ is *forgotten* at $x$.
- **Join node**: This is a node $x$ of $T$ with exactly two children $y$ and $z$, such that $\phi(x) = \phi(y) = \phi(z)$.

For a node $x$ of $T$, we define $G_x = (V_x, E_x)$ as follows:

- $V_x = \cup_{y \text{ is a descendant of } x} \phi(y)$
- $E_x = \{\text{All edges introduced at } x \text{ and its descendants}\}$

We say that $G_x$ is the subgraph of $G$ *hanging from* $x$.

# Nice Tree Decomposition(contd.)

- **Forget node**: This is a node $x$ of $T$, with exactly one child $y$ such that $\phi(x) = \phi(y) \setminus \{v\}$ for some $v \in \phi(y)$; we say that $v$ is *forgotten* at $x$.
- **Join node**: This is a node $x$ of $T$ with exactly two children $y$ and $z$, such that $\phi(x) = \phi(y) = \phi(z)$.

For a node $x$ of $T$, we define $G_x = (V_x, E_x)$ as follows:

- $V_x = \cup_{y \text{ is a descendant of } x} \phi(y)$
- $E_x = \{\text{All edges introduced at } x \text{ and its descendants}\}$

We say that $G_x$ is the subgraph of $G$ *hanging from* $x$.

If a graph $G$ has a tree decomposition of width $k$, then it admits a nice tree decomposition of width at most $k$. Moreover, given a tree decomposition $(T, \phi)$ of width $k$, one can find a nice tree decomposition of width $k$, in time $O(k^2 \cdot \max(|V(T)|, |V(G)|))$ that has $O(k|V(G)|)$ nodes.

# Revisit the problems

$\mathbb{T}-$FAIR VERTEX COVER
Input: An undirected $t$-coloured graph $G = (V, E)$, a nice tree decompositon $(T, \phi)$ of $G$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: width of $(T, \phi)$
Question: Does $G$ have a $\mathbb{T}-$fair vertex cover?

# Revisit the problems

$\mathbb{T}-$FAIR VERTEX COVER
Input: An undirected $t$-coloured graph $G = (V, E)$, a nice tree decompositon $(T, \phi)$ of $G$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: width of $(T, \phi)$
Question: Does $G$ have a $\mathbb{T}-$fair vertex cover?

$\mathbb{T}-$FAIR FEEDBACK VERTEX SET
Input: An undirected $t$-coloured graph $G = (V, E)$, a nice tree decompositon $(T, \phi)$ of $G$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: width of $(T, \phi)$
Question: Does $G$ have a $\mathbb{T}-$fair feedback vertex set?

# Revisit the problems

$\mathbb{T}-$FAIR VERTEX COVER
Input: An undirected $t$-coloured graph $G = (V, E)$, a nice tree decomposition $(T, \phi)$ of $G$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: width of $(T, \phi)$
Question: Does $G$ have a $\mathbb{T}-$fair vertex cover?

$\mathbb{T}-$FAIR FEEDBACK VERTEX SET
Input: An undirected $t$-coloured graph $G = (V, E)$, a nice tree decomposition $(T, \phi)$ of $G$ and $t$-tuple of integers, $\mathbb{T} = (k_i)_{i=1}^{t}$.
Parameter: width of $(T, \phi)$
Question: Does $G$ have a $\mathbb{T}-$fair feedback vertex set?

$t_w :=$ width of $(T, \phi)$.

# Fair Vertex Cover for graphs with bounded Treewidth

$\mathbb{T} = (k_i)_{i=}$

We will use dynamic programming to solve Fair Vertex Cover for graphs with bounded Treewidth.

For a node $x$ of $T$, a subset $S$ of $\phi(x)$, a $t$-tuple of integers $(r_i)_{i=1}^{t}$, where for each $i \in [t]$, $0 \leq r_i \leq k_i$, we define $I_x[S, (r_i)_{i=1}^{t}]$ as follows:

- $I_x[S, (r_i)_{i=1}^{t}] = 1$, if $G_x$ has an $(r_i)_{i=1}^{t}$-fair vertex cover which intersects $\phi(x)$ at $S$.
- $I_x[S, (r_i)_{i=1}^{t}] = 0$, otherwise

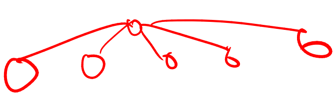
If all of the above entries are correctly evaluated, then $G$ has $\mathbb{T}$-fair vertex cover if and only if $I_r[\emptyset, \mathbb{T}] = 1$.

$\phi(x)$ $x$ $S$

$(r_1, r_2, \dots, r_t)$

$x$ $G_x$

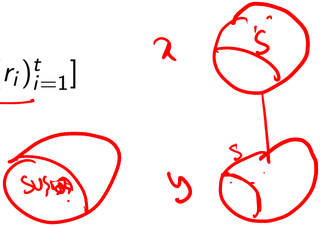
# Fair VC for graphs with bounded Treewidth(contd.)



$(r_i) = 0$

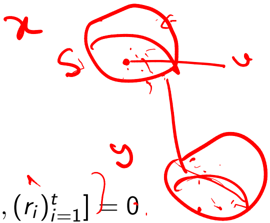
We will now compute the entries of DP table. For a node $x$ of $T$, a subset $S$ of $\phi(x)$, and a $t$-tuple of integers $(r_i)_{i=1}^{t}$, we determine the value of $I_x[S, (r_i)_{i=1}^{t}]$ as follows.

- $x$ is a leaf node. If $\forall i \in [t], r_i = 0$, then $I_x[\emptyset, (r_i)_{i=1}^{t}] = 1$, otherwise $I_x[\emptyset, (r_i)_{i=1}^{t}] = 0$.
- $x$ has a child $y$ and forgets vertex $v$.
  $I_x[S, (r_i)_{i=1}^{t}] = I_y[S, (r_i)_{i=1}^{t}] \oplus I_y[S \cup \{v\}, (r_i)_{i=1}^{t}]$



# Fair VC for graphs with bounded Treewidth(contd.)

$I_x[S,(r_i))$




- $x$ has a child $y$ and introduces vertex $v$.
  - If $v \notin S$, $I_x[S,(r_i)_{i=1}^t] = I_y[S,(r_i)_{i=1}^t]$
  - $v \in S$ and for some $j \in c(v)$, $r_j = 0$, then $I_x[S,(r_i)_{i=1}^t] = 0$
  - In all other cases, $I_x[S,(r_i)_{i=1}^t] = I_y[S \setminus \{v\},(r'_i)_{i=1}^t]$, where for $i \in [t]$, $r'_i = |\{w \in \{v\} | i \in c(w)\}|$

$v \in S$

$r'_i = r_i - 1$

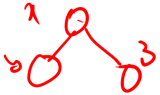
$r_i = r_i$

# Fair VC for graphs with bounded Treewidth(contd.)

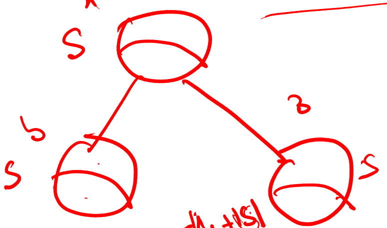
- $x$ has a child $y$ and introduces vertex $v$.
  - If $v \notin S$, $I_x[S, (r_i)_{i=1}^t] = I_y[S, (r_i)_{i=1}^t]$
  - $v \in S$ and for some $j \in c(v)$, $r_j = 0$, then $I_x[S, (r_i)_{i=1}^t] = 0$
  - In all other cases, $I_x[S, (r_i)_{i=1}^t] = I_y[S \setminus \{v\}, (r'_i)_{i=1}^t]$, where for $i \in [t]$, $r'_i = |\{w \in \{v\} | i \in c(w)\}|$
- $x$ has a child $y$ and introduces edge $uv$.
  - $S \cap \{u, v\} = \emptyset$, then $I_x[S, (r_i)_{i=1}^t] = 0$
  - Otherwise, $I_x[S, (r_i)_{i=1}^t] = I_y[S, (r_i)_{i=1}^t]$

# Fair VC for graphs with bounded Treewidth(contd.)

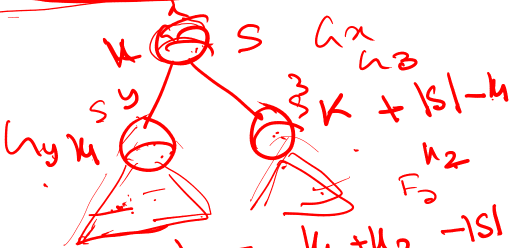
$S \subseteq \phi(x)$




If $x$ is a join node with children $y$ and $z$, we do as follows:
Consider all tuples $(a_i)_{i=1}^t$, such that $0 \leq a_i \leq r_i$. Evaluate the following.
$I_x[S,(r_i)_{i=1}^t] \leftarrow$
$(I_x[S,(r_i)_{i=1}^t] \oplus (I_y[S,(a_i)_{i=1}^t] \odot I_z[S,(r_i + |\{w \in S | i \in c(w)\}| - a_i)_{i=1}^t]))$




$n_2 = n - n_1 + |S|$



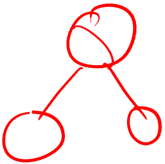

$n = |F_y \cup F_y| = n_1 + n_2 - |S|$

# Fair VC for graphs with bounded Treewidth(contd.)

If $x$ is a join node with children $y$ and $z$, we do as follows:
Consider all tuples $(a_i)_{i=1}^t$, such that $0 \leq a_i \leq r_i$. Evaluate the following.
$I_x[S,(r_i)_{i=1}^t] \leftarrow$
$(I_x[S,(r_i)_{i=1}^t] \oplus (I_y[S,(a_i)_{i=1}^t] \odot I_z[S,(r_i + |\{w \in S | i \in c(w)\}| - a_i)_{i=1}^t]))$

$2^{tw+1}$ S

tw



The runing time of the algorithm is $n^{\mathcal{O}(1)} \cdot 2^{t_w}$.


Let $v$ be a vertex of degree 3 or more. Let $H'$ be the graph obtained by deleting the vertex $v$ from $G$, $c'$ be the function obtained by restricting $c$ to $V(H') = (V(G) \setminus \{v\})$, and let $\mathbb{T}' = \{k'_1, k'_2, \ldots, k'_t\}$ where for each $i \in [t]$ we have $k'_i = k_i - |\{w \in \{v\} | i \in c(w)\}|$.

$k_i' = k_i - 1$

$(G, c, \mathbb{T})$

$G - v, k_i -$

$(G - v, c', \mathbb{T}')$

# Fpt Algorithm for Fair Vertex Cover

Let $v$ be a vertex of degree 3 or more. Let $H'$ be the graph obtained by deleting the vertex $v$ from $G$, $c'$ be the function obtained by restricting $c$ to $V(H') = (V(G) \setminus \{v\})$, and let $\mathbb{T}' = \{k'_1, k'_2, \ldots, k'_t\}$ where for each $i \in [t]$ we have $k'_i = k_i - |\{w \in \{v\} | i \in c(w)\}|$.
Let $H''$ be the graph obtained by deleting the open neighbourhood $N(v)$ from $G$, let $c''$ be the function obtained by restricting $c$ to $V(H'') = (V(G) \setminus N(v))$, and let $\mathbb{T}'' = \{k''_1, k''_2, \ldots, k''_t\}$ where for each $i \in [t]$ we have $k''_i = k_i - |\{w \in N(v) | i \in c(w)\}|$.

Let $v$ be a vertex of degree 3 or more. Let $H'$ be the graph obtained by deleting the vertex $v$ from $G$, $c'$ be the function obtained by restricting $c$ to $V(H') = (V(G) \setminus \{v\})$, and let $\mathbb{T}' = \{k'_1, k'_2, \ldots, k'_t\}$ where for each $i \in [t]$ we have $k'_i = k_i - |\{w \in \{v\} | i \in c(w)\}|$.
Let $H''$ be the graph obtained by deleting the open neighbourhood $N(v)$ from $G$, let $c''$ be the function obtained by restricting $c$ to $V(H'') = (V(G) \setminus N(v))$, and let $\mathbb{T}'' = \{k''_1, k''_2, \ldots, k''_t\}$ where for each $i \in [t]$ we have $k''_i = k_i - |\{w \in N(v) | i \in c(w)\}|$.
The branching rule recursively solves the two instances $(H', c', \mathbb{T}')$ and $(H'', c'', \mathbb{T}'')$. If at least one of these recursive calls returns YES, then the rule returns YES; otherwise it returns NO.

# FPT Algorithm for Fair Vertex Cover(contd.)

For solving the base case, use the bounded treewidth algorithm as a routine.

# Fpt Algorithm for Fair Vertex Cover(contd.)

1 vs 3

Branches

$n^{O(1)} f(k)$

For solving the base case, use the bounded treewidth algorithm as a routine.

The branching algorithm follows the relation $T(k) \leq T(k-1) + T(k-3)$, when $k \geq 3$. Substituting $\sum_{i=1}^{t} k_i$ for $k$, we can get an upper bound on the number of recursive calls. Thus, the total time taken by the algorithm is $n^{O(1)} \cdot 1.4656^{\sum_{i=1}^{t} k_i}$.

$n^{O(1)} 1.4656^{\sum k_i}$