Parameterized algorithms: Tree width and dynamic programming

Maria Serna

Fall 2023

1. Parameterizing by treewidth
2. Nice tree decomposition
3. Algorithmic meta theorems

Equivalent tw parameterizations for property P

Equivalent tw parameterizations for property P

TW-K-P

Input: A graph G, a tree decomposition (T, X) of G
and an integer k,

Parameter: $width(T, X) + k$

Question: Is P(G, k) true?

Equivalent tw parameterizations for property P

TW-K-P

Input: A graph G, a tree decomposition (T, X) of G
and an integer k,
Parameter: $width(T, X) + k$
Question: Is $\mathrm{P}(G, k)$ true?

TW-K-P

Input: A graph G and integers w and k,
Parameter: $w + k$
Question: Is $tw(G) \leq w$ and $\mathrm{P}(G, k)$ true?

Small tree decomposition

Small tree decomposition

- A tree decomposition (T, X) is small if for distinct $u, v \in V(T)$, $X_u \not\subseteq X_v$ and $X_v \not\subseteq X_u$.

Small tree decomposition

- A tree decomposition (T, X) is small if for distinct $u, v \in V(T)$, $X_u \not\subseteq X_v$ and $X_v \not\subseteq X_u$.
- When given a tree decomposition of G, in polynomial time we can construct a small tree decomposition of G with the same width.

Small tree decomposition

- A tree decomposition (T, X) is small if for distinct $u, v \in V(T)$, $X_u \not\subseteq X_v$ and $X_v \not\subseteq X_u$.
- When given a tree decomposition of G, in polynomial time we can construct a small tree decomposition of G with the same width.
  - If a tree decomposition is not small there should be two adjacent nodes $u, v \in V(T)$ with $X_u \subseteq X_v$.

Small tree decomposition

- A tree decomposition (T, X) is small if for distinct $u, v \in V(T)$, $X_u \not\subseteq X_v$ and $X_v \not\subseteq X_u$.
- When given a tree decomposition of G, in polynomial time we can construct a small tree decomposition of G with the same width.
  - If a tree decomposition is not small there should be two adjacent nodes $u, v \in V(T)$ with $X_u \subseteq X_v$.
  - Contracting uv into a new node w with $X_w = X_v$ gives a smaller tree decomposition for G.

Small tree decomposition

- A tree decomposition (T, X) is small if for distinct $u, v \in V(T)$, $X_u \not\subseteq X_v$ and $X_v \not\subseteq X_u$.
- When given a tree decomposition of G, in polynomial time we can construct a small tree decomposition of G with the same width.
  - If a tree decomposition is not small there should be two adjacent nodes $u, v \in V(T)$ with $X_u \subseteq X_v$.
  - Contracting uv into a new node w with $X_w = X_v$ gives a smaller tree decomposition for G.
  - repeating the above procedure we get a small tree decomposition.

Small tree decomposition

- If (T, X) is a small tree decomposition of G. Then $|V(T)| \leq |V(G)|$

Small tree decomposition

- If (T, X) is a small tree decomposition of G. Then $|V(T)| \leq |V(G)|$

  Exercise: proof by induction

Small tree decomposition

- If (T, X) is a small tree decomposition of G. Then $|V(T)| \leq |V(G)|$

  Exercise: proof by induction

- We can make use the FPT algorithm that given a (G, w) decides iwhether $tw(G) > w$ or produces a tree decomposition of width $4w + 4$.

Small tree decomposition

- If (T, X) is a small tree decomposition of G. Then $|V(T)| \leq |V(G)|$

  Exercise: proof by induction

- We can make use the FPT algorithm that given a (G, w) decides iwhether $tw(G) > w$ or produces a tree decomposition of width $4w + 4$.
- We can further assume that such a tree decomposition is small.

Rooted tree decomposition

- Let (T, X) be a tree decomposition of G of width w.

Rooted tree decomposition

- Let (T, X) be a tree decomposition of G of width w.
- Make T into a rooted tree by choosing a root $r \in V(T)$, and replacing edges by arcs in such a way that every node points to its parent.

Rooted tree decomposition

- Let (T, X) be a tree decomposition of G of width w.
- Make T into a rooted tree by choosing a root $r \in V(T)$, and replacing edges by arcs in such a way that every node points to its parent.
- For $v \in V(T)$, $R_T(v)$ denotes the nodes in the subtree rooted at v (including v).

Rooted tree decomposition

- Let (T, X) be a tree decomposition of G of width w.
- Make T into a rooted tree by choosing a root $r \in V(T)$, and replacing edges by arcs in such a way that every node points to its parent.
- For $v \in V(T)$, $R_T(v)$ denotes the nodes in the subtree rooted at v (including v).
- For $v \in V(T)$, $V(v) = X(R_T(v))$, and $G(v) = G[V(v)]$.

Rooted tree decomposition

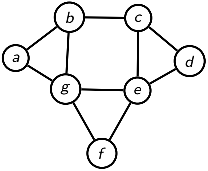
- Let (T, X) be a tree decomposition of G of width w.
- Make T into a rooted tree by choosing a root $r \in V(T)$, and replacing edges by arcs in such a way that every node points to its parent.
- For $v \in V(T)$, $R_T(v)$ denotes the nodes in the subtree rooted at v (including v).
- For $v \in V(T)$, $V(v) = X(R_T(v))$, and $G(v) = G[V(v)]$.

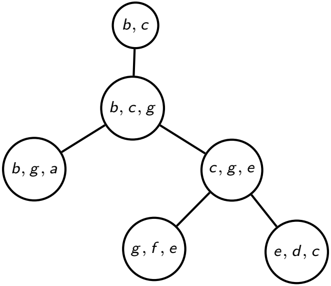
  Thus, we have an induced subgraph associated to each node.

Rooted tree decomposition

- Let (T, X) be a tree decomposition of G of width w.
- Make T into a rooted tree by choosing a root $r \in V(T)$, and replacing edges by arcs in such a way that every node points to its parent.
- For $v \in V(T)$, $R_T(v)$ denotes the nodes in the subtree rooted at v (including v).
- For $v \in V(T)$, $V(v) = X(R_T(v))$, and $G(v) = G[V(v)]$.

  Thus, we have an induced subgraph associated to each node.
- Notice that X_v is a separator in G.





Exercise For this rooted tree decomposition. Draw the graphs associated to each node in the tree. What are the differences among parent-child graphs?

tw-k-Vertex Coloring belongs to FPT

- To show that tw-k-Vertex Coloring $\in$ FPT we use dynamic programming on the tree decomposition.

tw-k-Vertex Coloring belongs to FPT

- To show that tw-k-Vertex Coloring $\in$ FPT we use dynamic programming on the tree decomposition.
- Let H_1 and H_2 be two subgraphs of G, with valid k-colorings α_1 and α_2 respectively.

tw-k-Vertex Coloring belongs to FPT

- To show that tw-k-Vertex Coloring $\in$ FPT we use dynamic programming on the tree decomposition.
- Let H_1 and H_2 be two subgraphs of G, with valid k-colorings α_1 and α_2 respectively.
- α_2 is α_1-compatible if for all $v \in V(H_1) \cap V(H_2)$, $\alpha_2(v) = \alpha_1(v)$.

tw-k-Vertex Coloring belongs to FPT

- To show that TW-K-VERTEX COLORING $\in$ FPT we use dynamic programming on the tree decomposition.
- Let H_1 and H_2 be two subgraphs of G, with valid k-colorings α_1 and α_2 respectively.
- α_2 is α_1-compatible if for all $v \in V(H_1) \cap V(H_2)$, $\alpha_2(v) = \alpha_1(v)$.
- Let (T, X) be a rooted tree decomposition of G with width w.

tw-k-Vertex Coloring belongs to FPT

- To show that TW-K-VERTEX COLORING $\in$ FPT we use dynamic programming on the tree decomposition.
- Let H_1 and H_2 be two subgraphs of G, with valid k-colorings α_1 and α_2 respectively.
- α_2 is α_1-compatible if for all $v \in V(H_1) \cap V(H_2)$, $\alpha_2(v) = \alpha_1(v)$.
- Let (T, X) be a rooted tree decomposition of G with width w.
  - For every $v \in V(T)$ and every proper k-coloring α of $G[Xv]$, define $P_v(\alpha) = 1$ iff $G(v)$ has an α-compatible k-coloring β.

tw-k-Vertex Coloring belongs to FPT

- To show that TW-K-VERTEX COLORING $\in$ FPT we use dynamic programming on the tree decomposition.
- Let H_1 and H_2 be two subgraphs of G, with valid k-colorings α_1 and α_2 respectively.
- α_2 is α_1-compatible if for all $v \in V(H_1) \cap V(H_2)$, $\alpha_2(v) = \alpha_1(v)$.
- Let (T, X) be a rooted tree decomposition of G with width w.
  - For every $v \in V(T)$ and every proper k-coloring α of $G[Xv]$, define $P_v(\alpha) = 1$ iff $G(v)$ has an α-compatible k-coloring β.
- Our algorithm computes $P_v(\alpha)$, for each node in T, from leaves to root.

tw-k-Vertex Coloring belongs to FPT

Lemma

$P_u(\alpha) = 1$ iff for all children v of u, there is an α-compatible coloring β of $G[X_v]$ with $P_v(\beta) = 1$.

Proof.

tw-k-Vertex Coloring belongs to FPT

Lemma

$P_u(\alpha) = 1$ iff for all children v of u, there is an α-compatible coloring β of $G[X_v]$ with $P_v(\beta) = 1$.

Proof.

- $(\Rightarrow)$

tw-k-Vertex Coloring belongs to FPT

Lemma

$P_u(\alpha) = 1$ iff for all children v of u, there is an α-compatible coloring β of $G[X_v]$ with $P_v(\beta) = 1$.

Proof.

- ($\Rightarrow$) Let γ be an α-compatible coloring of $G(u)$. $G(v)$ is a subgraph of $G(u)$, so restricting to X_v gives the desired coloring β.

tw-k-Vertex Coloring belongs to FPT

- ($\Leftarrow$)

tw-k-Vertex Coloring belongs to FPT

- ($\Leftarrow$) Consider two children *v* and *w* of *u*, and suppose they have α-compatible colorings β and γ respectively.

tw-k-Vertex Coloring belongs to FPT

- ($\Leftarrow$) Consider two children v and w of u, and suppose they have α-compatible colorings β and γ respectively.
  - Since (T, X) is a tree decomposition, $V(v) \cap V(w) \subset X_u$, so β is γ-compatible.

tw-k-Vertex Coloring belongs to FPT

- ($\Leftarrow$) Consider two children v and w of u, and suppose they have α-compatible colorings β and γ respectively.
  - Since (T, X) is a tree decomposition, $V(v) \cap V(w) \subset X_u$, so β is γ-compatible.
  - Combining β and γ gives $\delta : V(u) \to \{1, \ldots, k\}$.

tw-k-Vertex Coloring belongs to FPT

- ($\Leftarrow$) Consider two children v and w of u, and suppose they have α-compatible colorings β and γ respectively.
  - Since (T, X) is a tree decomposition, $V(v) \cap V(w) \subset X_u$, so β is γ-compatible.
  - Combining β and γ gives $\delta : V(u) \to \{1, \ldots, k\}$.
  - Since (T, X) is a tree decomposition, there are no edges $xy \in E(G)$ with $x \in V(v) - X_u$ and $y \in V(w) - X_u$,
    so δ is a proper k-coloring of $G(u)$.

tw-k-Vertex Coloring belongs to FPT

- ($\Leftarrow$) Consider two children v and w of u, and suppose they have α-compatible colorings β and γ respectively.
    - Since (T, X) is a tree decomposition, $V(v) \cap V(w) \subset X_u$, so β is γ-compatible.
    - Combining β and γ gives $\delta : V(u) \to \{1, \ldots, k\}$.
    - Since (T, X) is a tree decomposition, there are no edges $xy \in E(G)$ with $x \in V(v) - X_u$ and $y \in V(w) - X_u$,
      so δ is a proper k-coloring of $G(u)$.
    - The same can be done for all children of u simultaneously.

EndProof

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1}n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1} n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

- For $v \in V(T)$ and a k-coloring α of $G[X_v]$, we compute $P_v(\alpha)$ starting at the leaves of T, and using the recurrence.

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1}n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

- For $v \in V(T)$ and a k-coloring α of $G[X_v]$, we compute $P_v(\alpha)$ starting at the leaves of T, and using the recurrence.
- $G = G(r)$ is k-colorable iff $Pr(\alpha) = 1$, for some α.

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1}n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

- For $v \in V(T)$ and a k-coloring α of $G[X_v]$, we compute $P_v(\alpha)$ starting at the leaves of T, and using the recurrence.
- $G = G(r)$ is k-colorable iff $Pr(\alpha) = 1$, for some α.
- testing whether α is a $G[Xv]$ coloring can be done in $O(w^2)$.

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1}n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

- For $v \in V(T)$ and a k-coloring α of $G[X_v]$, we compute $P_v(\alpha)$ starting at the leaves of T, and using the recurrence.
- $G = G(r)$ is k-colorable iff $Pr(\alpha) = 1$, for some α.
- testing whether α is a $G[Xv]$ coloring can be done in $O(w^2)$.
- computing $Pv(\alpha)$ for a valid α can be done in $n^{O(1)}$

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1}n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

- For $v \in V(T)$ and a k-coloring α of $G[X_v]$, we compute $P_v(\alpha)$ starting at the leaves of T, and using the recurrence.
- $G = G(r)$ is k-colorable iff $Pr(\alpha) = 1$, for some α.
- testing whether α is a $G[Xv]$ coloring can be done in $O(w^2)$.
- computing $Pv(\alpha)$ for a valid α can be done in $n^{O(1)}$
- the total complexity is mainly determined by the number of candidates for α which is $k^{|X_v|}$:

tw-k-Vertex Coloring belongs to FPT

Theorem

Let (T, X) be a rooted small tree decomposition of G of width w. In time $k^{w+1}n^{O(1)}$ we can decide whether G is k-colorable.

Proof.

- For $v \in V(T)$ and a k-coloring α of $G[X_v]$, we compute $P_v(\alpha)$ starting at the leaves of T, and using the recurrence.
- $G = G(r)$ is k-colorable iff $Pr(\alpha) = 1$, for some α.
- testing whether α is a $G[Xv]$ coloring can be done in $O(w^2)$.
- computing $Pv(\alpha)$ for a valid α can be done in $n^{O(1)}$
- the total complexity is mainly determined by the number of candidates for α which is $k^{|X_v|}$: $|V(T)|k^{w+1}n^{O(1)}$.

□

Nice tree decomposition

Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.

Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.
- A rooted tree decomposition (T, X) is nice if for every $u \in V(T)$

Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.
- A rooted tree decomposition (T, X) is nice if for every $u \in V(T)$
  - $|X_u| = 1$ (start)

Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.
- A rooted tree decomposition (T, X) is nice if for every $u \in V(T)$
  - $|X_u| = 1$ (start)
  - u has one child v

Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.
- A rooted tree decomposition (T, X) is nice if for every $u \in V(T)$
  - $|X_u| = 1$ (start)
  - u has one child v
    with $X_u \subseteq X_v$ and $|X_u| = |Xv| - 1$ (forget)

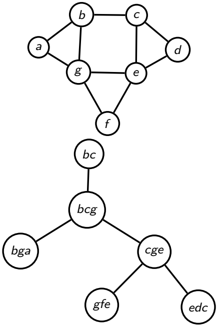
Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.
- A rooted tree decomposition (T, X) is nice if for every $u \in V(T)$
  - $|X_u| = 1$ (start)
  - u has one child v
    with $X_u \subseteq X_v$ and $|X_u| = |Xv| - 1$ (forget)
    with $X_v \subseteq X_u$ and $|Xu| = |Xv| + 1$ (introduce)

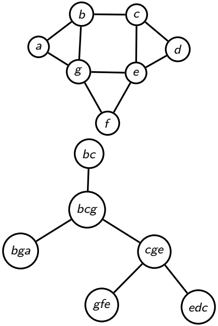
Nice tree decomposition

- A nice tree decomposition is a variant in which the structure of the tree is simpler.
- A rooted tree decomposition (T, X) is nice if for every $u \in V(T)$
  - $|X_u| = 1$ (start)
  - u has one child v
    - with $X_u \subseteq X_v$ and $|X_u| = |Xv| - 1$ (forget)
    - with $X_v \subseteq X_u$ and $|Xu| = |Xv| + 1$ (introduce)
  - u has two children v and w with $X_u = X_v = X_w$ (join)

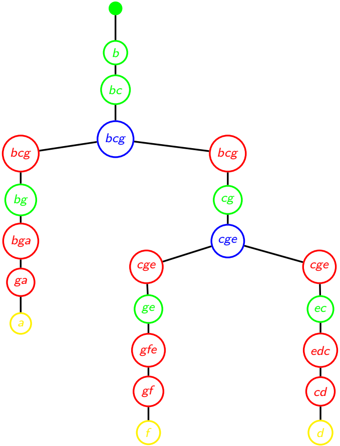
Nice tree decomposition



Nice tree decomposition



Start Introduce Forget Join



Nice tree decomposition

Lemma

Computing a rooted nice tree decomposition with width at most k, given a small tree decomposition of width at most k takes $O(kn)$ time.

Nice tree decomposition

- Nodes in the tree
  node u holds a subset of vertices X_u, and has a subgraph G_u associated to it.
- the root r has $X_r = \emptyset$ and $G_r = G$.
- nodes can be of four types:
  Start Introduce Forget Join

A parameterization for Min Vertex Cover

A parameterization for Min Vertex Cover

TW-MIN VERTEX COVER
Input: A graph G, a tree decomposition (T, X),
Parameter: $width(T, X)$
Question: Compute a minimum size vertex cover of G

The algorithm for tw-Min Vertex Cover

The algorithm for tw-Min Vertex Cover

- We can assume that (T, X) is nice

The algorithm for tw-Min Vertex Cover

- We can assume that (T, X) is nice
- For each node $v \in V(T)$ we keep a table $s_v(C)$ for each $C \subseteq X_v$ holding the minimum size of a vertex cover C' of $G(v)$ with $C' \cap X_v = C$ if such a C' exists, and $s_v(C) = \infty$ otherwise.

The algorithm for tw-Min Vertex Cover

- We can assume that (T, X) is nice
- For each node $v \in V(T)$ we keep a table $s_v(C)$ for each $C \subseteq X_v$ holding the minimum size of a vertex cover C' of $G(v)$ with $C' \cap X_v = C$ if such a C' exists, and $s_v(C) = \infty$ otherwise.
- The value of $s_r(\emptyset)$ is the size of a minimum vertex cover of G.

The algorithm for tw-Min Vertex Cover

- We can assume that (T, X) is nice
- For each node $v \in V(T)$ we keep a table $s_v(C)$ for each $C \subseteq X_v$ holding the minimum size of a vertex cover C' of $G(v)$ with $C' \cap X_v = C$ if such a C' exists, and $s_v(C) = \infty$ otherwise.
- The value of $s_r(\emptyset)$ is the size of a minimum vertex cover of G.

- We deal with each type of node separately

Start node

Start node

Claim

Let u be a leaf of T with $X_u = \{x\}$.
$s_u(\{x\}) = 1$ and $s_u(\emptyset) = 0$.

Introduce node

Introduce node

Claim

Let u be an *introduce* node, let v be its unique child and assume that $\{x\} = X_u - X_v$.

Introduce node

Claim

Let u be an *introduce* node, let v be its unique child and assume that $\{x\} = X_u - X_v$.
Then for all $C \subseteq X_u$

Introduce node

Claim

Let u be an *introduce* node, let v be its unique child and assume that $\{x\} = X_u - X_v$.
Then for all $C \subseteq X_u$

- If C is not a vertex cover of $G[X_u]$ then $s_u(C) = \infty$.

Introduce node

Claim

Let u be an *introduce* node, let v be its unique child and assume that $\{x\} = X_u - X_v$.
Then for all $C \subseteq X_u$

- If C is not a vertex cover of $G[X_u]$ then $s_u(C) = \infty$.
- If C is a vertex cover of $G[X_u]$ and $x \in C$ then $s_u(C) = s_v(C - x) + 1$.

Introduce node

Claim

Let u be an *introduce* node, let v be its unique child and assume that $\{x\} = X_u - X_v$.
Then for all $C \subseteq X_u$

- If C is not a vertex cover of $G[X_u]$ then $s_u(C) = \infty$.
- If C is a vertex cover of $G[X_u]$ and $x \in C$ then $s_u(C) = s_v(C - x) + 1$.
- If C is a vertex cover of $G[X_u]$ and $x \notin C$ then $s_u(C) = s_v(C)$.

Introduce node

Claim

Let u be an *introduce* node, let v be its unique child and assume that $\{x\} = X_u - X_v$.
Then for all $C \subseteq X_u$

- If C is not a vertex cover of $G[X_u]$ then $s_u(C) = \infty$.
- If C is a vertex cover of $G[X_u]$ and $x \in C$ then $s_u(C) = s_v(C - x) + 1$.
- If C is a vertex cover of $G[X_u]$ and $x \notin C$ then $s_u(C) = s_v(C)$.

  All neighbors of x in $G(u)$ are in X_v and therefore in C since C is a vertex cover of $G[X_u]$.

Forget node

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.
Then for all $C \subseteq X_u$, $s_u(C) = \min\{s_v(C), s_v(C + x)\}$.

Proof.

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.
Then for all $C \subseteq X_u$, $s_u(C) = \min\{s_v(C), s_v(C + x)\}$.

Proof.

- ($\geq$) Let C' be a min$\textsc{vc}$ of $G(u) = G(v)$ with $C' \cap X_u = C$.

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.
Then for all $C \subseteq X_u$, $s_u(C) = \min\{s_v(C), s_v(C+x)\}$.

Proof.

- $(\geq)$ Let C' be a minVC of $G(u) = G(v)$ with $C' \cap X_u = C$.
  - If $x \notin C'$ then $C' \cap X_u = C$ so $|C'| \geq s_v(C)$.

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.
Then for all $C \subseteq X_u$, $s_u(C) = \min\{s_v(C), s_v(C + x)\}$.

Proof.

- ($\geq$) Let C' be a min VC of $G(u) = G(v)$ with $C' \cap X_u = C$.
  - If $x \notin C'$ then $C' \cap X_u = C$ so $|C'| \geq s_v(C)$.
  - If $x \in C'$ then similarly $|C'| \geq s_v(C + x)$.

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.
Then for all $C \subseteq X_u$, $s_u(C) = \min\{s_v(C), s_v(C + x)\}$.

Proof.

- $(\geq)$ Let C' be a min$\textsc{vc}$ of $G(u) = G(v)$ with $C' \cap X_u = C$.
  - If $x \notin C'$ then $C' \cap X_u = C$ so $|C'| \geq s_v(C)$.
  - If $x \in C'$ then similarly $|C'| \geq s_v(C + x)$.
- $(\leq)$ Let C_1 and C_2 be the $\textsc{vc}$s that determine $s_v(C)$ and $s_v(C + x)$ respectively.

Forget node

Claim

Let u be a *forget* node, let v be its unique child and assume that $\{v\} = X_v - X_u$.
Then for all $C \subseteq X_u$, $s_u(C) = \min\{s_v(C), s_v(C + x)\}$.

Proof.

- $(\geq)$ Let C' be a min$\textsc{vc}$ of $G(u) = G(v)$ with $C' \cap X_u = C$.
  - If $x \notin C'$ then $C' \cap X_u = C$ so $|C'| \geq s_v(C)$.
  - If $x \in C'$ then similarly $|C'| \geq s_v(C + x)$.
- $(\leq)$ Let C_1 and C_2 be the $\textsc{vc}$s that determine $s_v(C)$ and $s_v(C + x)$ respectively. C_1, C_2 are $\textsc{vc}$ of $G(u)$ compatible with C, so $s_v(C) \leq \min\{|C_1|, |C_2|\}$.

$\square$

Join node

Join node

Claim

Let u be a join node of T with children v and w.
Then for all $C \subseteq X_u$: $s_u(C) = s_v(C) + s_w(C) - |C|$.

Proof.

Join node

Claim

Let u be a join node of T with children v and w.
Then for all $C \subseteq X_u$: $s_u(C) = s_v(C) + s_w(C) - |C|$.

Proof.

- ($\geq$) If C' is a vertex cover of $G(u)$ with $C' \cap X_u = C$, then $C' \cap V(v)$ is a vertex cover of $G(v)$ and $C' \cap V(w)$ is a vertex cover of $G(w)$, which share $|C|$ vertices.

Join node

Claim

Let u be a join node of T with children v and w.
Then for all $C \subseteq X_u$: $s_u(C) = s_v(C) + s_w(C) - |C|$.

Proof.

- ($\geq$) If C' is a vertex cover of $G(u)$ with $C' \cap X_u = C$, then $C' \cap V(v)$ is a vertex cover of $G(v)$ and $C' \cap V(w)$ is a vertex cover of $G(w)$, which share $|C|$ vertices.
- ($\leq$) Two C-compatible vertex covers of $G(v)$ and $G(w)$ of size $s_v(C)$ and $s_w(C)$ can be combined to a vertex cover of $G(u)$ of size $s_v(C) + s_w(C) - |C|$.

□

Min Vertex cover parameterized by treewidth

Theorem

Let (T, X) be a rooted nice tree decomposition of width w of a graph G on n vertices. In time $2^{w+1}n^{O(1)}$ the size of a minimum vertex cover of G can be computed.

Proof.

Min Vertex cover parameterized by treewidth

Theorem

Let (T, X) be a rooted nice tree decomposition of width w of a graph G on n vertices. In time $2^{w+1}n^{O(1)}$ the size of a minimum vertex cover of G can be computed.

Proof.

- We can construct a minimum vertex cover as well, by tracing back through the tree decomposition.

Min Vertex cover parameterized by treewidth

Theorem

Let (T, X) be a rooted nice tree decomposition of width w of a graph G on n vertices. In time $2^{w+1}n^{O(1)}$ the size of a minimum vertex cover of G can be computed.

Proof.

- We can construct a minimum vertex cover as well, by tracing back through the tree decomposition.
- $+ \; O(f(k)n^{O(1)})$ to get the tree decomposition if needed.

$\square$

Closure properties

- A contraction of an edge (u, v) in a graph G consists in replacing u, v by a new vertex w which keeps as neighbors $N(u) \cup N(v)$

Closure properties

- A contraction of an edge (u, v) in a graph G consists in replacing u, v by a new vertex w which keeps as neighbors $N(u) \cup N(v)$
- A graph H is a minor of G if H can be obtained from G by a series of edge contractions.

Closure properties

- A contraction of an edge (u, v) in a graph G consists in replacing u, v by a new vertex w which keeps as neighbors $N(u) \cup N(v)$
- A graph H is a minor of G if H can be obtained from G by a series of edge contractions.
- If H is a minor of G then $tw(H) \leq tw(G)$

Closure properties

- A contraction of an edge (u, v) in a graph G consists in replacing u, v by a new vertex w which keeps as neighbors $N(u) \cup N(v)$
- A graph H is a minor of G if H can be obtained from G by a series of edge contractions.
- If H is a minor of G then $tw(H) \leq tw(G)$

  (T, X) is a tree decomposition. Contract xy into z. The tree decomposition (T, X') in which we replace x, y by z in any bag containing x or y (or both) is a valid tree decomposition.

Closure properties

- A contraction of an edge (u, v) in a graph G consists in replacing u, v by a new vertex w which keeps as neighbors $N(u) \cup N(v)$
- A graph H is a minor of G if H can be obtained from G by a series of edge contractions.
- If H is a minor of G then $tw(H) \leq tw(G)$

  (T, X) is a tree decomposition. Contract xy into z. The tree decomposition (T, X') in which we replace x, y by z in any bag containing x or y (or both) is a valid tree decomposition.

- If H is a subgraph of G then $tw(H) \leq tw(G)$

Algorithmic theorems

Algorithmic theorems

- Algorithmic theorems provide algorithms, i.e., a proof of the existence of an algorithm.

Algorithmic theorems

- Algorithmic theorems provide algorithms, i.e., a proof of the existence of an algorithm.
  - Vertex Cover, Dominating Set, 3-Coloring are solvable in linear time on graphs of constant treewidth.

Algorithmic theorems

- Algorithmic theorems provide algorithms, i.e., a proof of the existence of an algorithm.
  - Vertex Cover, Dominating Set, 3-Coloring are solvable in linear time on graphs of constant treewidth.
  - Vertex Cover, Feedback Vertex Set can be solved in sub-exponential time on planar graphs

Algorithmic theorems

- Algorithmic theorems provide algorithms, i.e., a proof of the existence of an algorithm.
  - Vertex Cover, Dominating Set, 3-Coloring are solvable in linear time on graphs of constant treewidth.
  - Vertex Cover, Feedback Vertex Set can be solved in sub-exponential time on planar graphs
- To get an algorithm, as we have done, you should working out all the details of the DP!

Algorithmic meta theorems

Algorithmic meta theorems

- Algorithmic meta theorems. No algorithm is constructed!

Algorithmic meta theorems

- Algorithmic meta theorems. No algorithm is constructed!
- But the existence of an algorithm is proved

Algorithmic meta theorems

- Algorithmic meta theorems. No algorithm is constructed!
- But the existence of an algorithm is proved
- Main uses: quick complexity classification tools, mapping the limits of applicability for specific techniques.

Algorithmic meta theorems

- Algorithmic meta theorems. No algorithm is constructed!
- But the existence of an algorithm is proved
- Main uses: quick complexity classification tools, mapping the limits of applicability for specific techniques.
- Usually they are grounded in logics or other properties

First Order Logic on graphs

- We express graph properties using logic

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example:

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example: Dominating Set of size 2

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example: Dominating Set of size 2

$$\exists x_1 \exists x_2 \forall y \; E(x_1, y) \vee E(x_2, y) \vee x_1 = y \vee x_2 = y)$$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example:

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \dots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example: Vertex Cover of size 2

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example: Vertex Cover of size 2

$$\exists x_1 \exists x_2 \forall y \forall z \; E(y, z) \rightarrow (y = x_1 \vee y = x_2 \vee z = x_1 \vee z = x_2$$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example:

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example: Clique of size 3

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg, \rightarrow$
  - Quantifiers $\forall$, $\exists$
- Example: Clique of size 3

  $\exists x_1 \exists x_2 \exists x_3 \; E(x_1, x_2) \wedge E(x_1, x_3) \wedge E(x_2, x_3)$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg$
  - Quantifiers $\forall$, $\exists$

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg$
  - Quantifiers $\forall$, $\exists$
- Many standard (parameterized) problems can be expressed in FO logic.

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg$
  - Quantifiers $\forall$, $\exists$
- Many standard (parameterized) problems can be expressed in FO logic.
- But some easy problems are inexpressible (e.g. connectivity).

First Order Logic on graphs

- We express graph properties using logic
- Basic vocabulary
  - Vertex variables: $x, y, z, \ldots$
  - Edge predicate $E(x, y)$, Equality $x = y$
  - Boolean connectives $\vee, \wedge, \neg$
  - Quantifiers $\forall$, $\exists$
- Many standard (parameterized) problems can be expressed in FO logic.
- But some easy problems are inexpressible (e.g. connectivity).
- Rule of thumb: FO = local properties

Monadic Second Order Logic

- MSO logic: we add to FO logic

Monadic Second Order Logic

- MSO logic: we add to FO logic
  - set variables $S_1, S_2, \ldots$

Monadic Second Order Logic

- MSO logic: we add to FO logic
  - set variables $S_1, S_2, \ldots$
  - and the $a \in$ predicate.

Monadic Second Order Logic

- MSO logic: we add to FO logic
  - set variables $S_1, S_2, \ldots$
  - and the $a \in$ predicate.
  - Quantifiers $\forall$, $\exists$

    MSO_1 logic: we can quantify over sets of vertices

    MSO_2 logic: we can quantify over sets of vertices and sets of edges

Monadic Second Order Logic

- MSO logic: we add to FO logic
  - set variables $S_1, S_2, \ldots$
  - and the $a \in$ predicate.
  - Quantifiers $\forall$, $\exists$

    MSO_1 logic: we can quantify over sets of vertices

    MSO_2 logic: we can quantify over sets of vertices and sets of edges
- Example:

Monadic Second Order Logic

- MSO logic: we add to FO logic
  - set variables $S_1, S_2, \dots$
  - and the $a \in$ predicate.
  - Quantifiers $\forall$, $\exists$
    MSO_1 logic: we can quantify over sets of vertices
    MSO_2 logic: we can quantify over sets of vertices and sets of edges
- Example: 2-coloring

Monadic Second Order Logic

- MSO logic: we add to FO logic
  - set variables $S_1, S_2, \ldots$
  - and the $a \in$ predicate.
  - Quantifiers $\forall$, $\exists$
    MSO$_1$ logic: we can quantify over sets of vertices
    MSO$_2$ logic: we can quantify over sets of vertices and sets of edges
- Example: 2-coloring

$$\exists V_1 \exists V_2 \forall x \forall y \; E(x, y) \to (x \in V_1 \leftrightarrow y \in V_2)$$

Algorithmic meta theorems

Algorithmic meta theorems

- All Monadic Second Order logic (MSO) expressible problems are solvable in linear time on graphs of constant treewidth.

Algorithmic meta theorems

- All Monadic Second Order logic (MSO) expressible problems are solvable in linear time on graphs of constant treewidth.
- All minor closed optimization problems can be solved in sub-exponential time on planar graphs

Algorithmic meta theorems

- All Monadic Second Order logic (MSO) expressible problems are solvable in linear time on graphs of constant treewidth.
- All minor closed optimization problems can be solved in sub-exponential time on planar graphs

Recall: No algorithm is constructed!