Algorithms for Graphs of Bounded Treewidth

Made by Moshe Sebag
CS department, Technion

The material for all parts of this lecture appears in Chapter 7 of the book "Parameterized Algorithms", by Cygan et al.
Table of Content

- Into: explanation of the term "treewidth of a graph"
- Definitions:
  1. Tree decomposition of a graph
  2. Nice tree decomposition
- Dynamic Programming (DP) on graphs of bounded treewidth
- Example of DP-based algorithm for Weighted Independent Set
- Run-time analysis of the algorithm
- Treewidth and Monadic second-order Logic
- Courcelle's theorem
The treewidth of an undirected graph is a number associated with the graph. Very roughly, treewidth captures how similar a graph is to a tree. Treewidth is commonly used as a parameter in the parameterized complexity analysis of graph algorithms. In this lecture, we focus on connections to the idea of dynamic programming on the structure of a graph.
The **treewidth** of an undirected graph is a number associated with the graph.

Very roughly, treewidth captures how similar a graph is to a tree.

Treewidth is commonly used as a parameter in the parameterized complexity analysis of graph algorithms.

In this lecture, we focus on connections to the idea of dynamic programming on the structure of a graph.

But how do you calculate the treewidth of a graph?
Tree Decomposition

- A tree decomposition is a mapping of a graph into a tree that can be used to define the treewidth of the graph.
Tree Decomposition

- Definition: a tree decomposition of a graph $G$ is a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where $T$ is a tree whose every node $t$ is assigned a vertex subset $X_t \subseteq V(G)$.

- The following three conditions hold:
  - (T1) $\bigcup_{t \in V(T)} X_t = V(G)$
  - (T2) For every $uv \in E(G)$, there exists a node $t$ of $T$ such that bag $X_t$ contains both $u$ and $v$.
  - (T3) For every $u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$, induces a connected subtree of $T$. 
Tree Decomposition

- Definition: a tree decomposition of a graph $G$ is a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where $T$ is a tree whose every node $t$ is assigned a vertex subset $X_t \subseteq V(G)$.

- The following three conditions hold:
  1. $\bigcup_{t \in V(T)} X_t = V(G)$
  2. For every $uv \in E(G)$, there exists a node $t$ of $T$ such that bag $X_t$ contains both $u$ and $v$.
  3. For every $u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$, induces a connected subtree of $T$.

I still didn’t get what is the TREEWIDTH?
After we defined what a tree composition is, we can define the treewidth of a graph.

The width of tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ equals $\max_{t \in V(T)} |X_t| - 1$.

The treewidth of a graph $G$, denoted by $tw(G)$, is the minimum possible width of a tree decomposition of $G$. 
Tree Decomposition
Intro to Lemma 1.

- Definition 1: \((A, B)\) is a **separation** of a graph \(G\) if \(A \cup B = V(G)\) and there is no edge between \(A \setminus B\) and \(B \setminus A\).
- Definition 2: Let \((A, B)\) be a separation of a graph, then \(A \cap B\) is a **separator** of this separation, and \(|A \cap B|\) is the **order of the separation**.

\(\{A,B,C,D,E\}, \{B,E,F,G,H\}\) is a separation of the graph.

\(\{B,E\}\) is the separator.
Tree Decomposition
Intro to Lemma 1.

- Definition 3: Let $A$ be a subset of $V(G)$, the **border of $A$**, denoted by $\partial(A)$, is the set of those vertices of $A$ that have a neighbor in $V(G) \setminus A$.

- For us the most crucial property of tree decompositions is that they define a sequence of separators in the graph.

For the subset $\{A,B,C,D,E\}$, $\partial(A)=\{B,E\}$.
**Tree Decomposition**

- **Lemma 1.** Let \((T, \{X_t\}_{t \in V(T)})\) be a tree decomposition of a graph \(G\) and let \(ab\) be an edge of \(T\).

  The forest \(T - ab\) obtained from \(T\) by deleting edge \(ab\) consists of two connected components \(T_a\) (containing \(a\)) and \(T_b\) (containing \(b\)).

  Let \(K = \bigcup_{t \in V(T_a)} X_t\) and \(M = \bigcup_{t \in V(T_b)} X_t\).

  Then \(\partial(K), \partial(M) \subseteq X_a \cap X_b\). Equivalently, \((K, M)\) is a separation of \(G\) with separator \(X_a \cap X_b\).
Lemma 1. Let \((T, \{X_t\}_{t \in V(T)})\) be a tree decomposition of a graph \(G\) and let \(ab\) be an edge of \(T\).

The forest \(T - ab\) obtained from \(T\) by deleting edge \(ab\) consists of two connected components \(T_a\) (containing \(a\)) and \(T_b\) (containing \(b\)).

Let \(K = \bigcup_{t \in V(T_a)} X_t\) and \(M = \bigcup_{t \in V(T_b)} X_t\).

Then \(\partial(K), \partial(M) \subseteq X_a \cap X_b\). Equivalently, \((K, M)\) is a separation of \(G\) with separator \(X_a \cap X_b\).
Lemma 1. Let \((T, \{X_t\}_{t \in V(T)})\) be a tree decomposition of a graph \(G\) and let \(ab\) be an edge of \(T\). The forest \(T - ab\) obtained from \(T\) by deleting edge \(ab\) consists of two connected components \(T_a\) (containing \(a\)) and \(T_b\) (containing \(b\)). Let \(K = \bigcup_{t \in V(T_a)} X_t\) and \(M = \bigcup_{t \in V(T_b)} X_t\). Then \(\partial(K), \partial(M) \subseteq X_a \cap X_b\). Equivalently, \((K, M)\) is a separation of \(G\) with separator \(X_a \cap X_b\).

\[
K = \{A, B, C, D, E\} \\
M = \{B, E, F, G, H\}
\]
Lemma 1. Let \((T, \{X_t\}_{t \in V(T)}) be a tree decomposition of a graph \(G\) and let \(ab\) be an edge of \(T\).

The forest \(T - ab\) obtained from \(T\) by deleting edge \(ab\) consists of two connected components \(T_a\) (containing \(a\)) and \(T_b\) (containing \(b\)).

Let \(K = \bigcup_{t \in V(T_a)} X_t\) and \(M = \bigcup_{t \in V(T_b)} X_t\).

Then \(\partial(K), \partial(M) \subseteq X_a \cap X_b\). Equivalently, \((K, M)\) is a separation of \(G\) with separator \(X_a \cap X_b\).

\[\begin{align*}
K &= \{A, B, C, D, E\} \\
M &= \{B, E, F, G, H\} \\
X_a &= \{C, B, E\} \\
X_b &= \{B, E, G\} \\
X_a \cap X_b &= \{B, E\} \\
\partial(K) &= \{B, E\} \\
\partial(M) &= \{B, E\}
\end{align*}\]
Lemma 1. Let \((T, \{X_t\}_{t \in V(T)})\) be a tree decomposition of a graph \(G\) and let \(ab\) be an edge of \(T\). The forest \(T - ab\) obtained from \(T\) by deleting edge \(ab\) consists of two connected components \(T_a\) (containing \(a\)) and \(T_b\) (containing \(b\)). Let \(K = \bigcup_{t \in V(T_a)} X_t\) and \(M = \bigcup_{t \in V(T_b)} X_t\). Then \(\partial(K), \partial(M) \subseteq X_a \cap X_b\). Equivalently, \((K, M)\) is a separation of \(G\) with separator \(X_a \cap X_b\).

\[
\begin{align*}
K &= \{A,B,C,D,E\} \\
M &= \{B,E,F,G,H\} \\
X_a &= \{C,B,E\} \\
X_b &= \{B,E,G\} \\
X_a \cap X_b &= \{B,E\} \\
\partial(K) &= \{B,E\} \\
\partial(M) &= \{B,E\}
\end{align*}
\]
Nice Tree Decomposition - motivation
We will think of a nice tree decompositions as rooted trees. A (rooted) tree decomposition \((T, \{X_t\}_{t \in V(T)})\) is nice if the following conditions are satisfied:

- \(X_r = \emptyset\) for \(r\) the root of \(T\) and \(X_l = \emptyset\) for every leaf \(l\) of \(T\).
- Every non-leaf node of \(T\) is of one of the following three types:
Nice Tree Decomposition

- We will think of a nice tree decompositions as rooted trees.
- A (rooted) tree decomposition \((T, \{X_t\}_{t \in V(T)})\) is **nice** if the following conditions are satisfied:
  - \(X_r = \emptyset\) for \(r\) the root of \(T\) and \(X_l = \emptyset\) for every leaf \(l\) of \(T\).
  - Every non-leaf node of \(T\) is of one of the following three types:
    1. Introduce node:
      a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{v\}\) for some vertex \(v \notin X_{t'}\) (we say that \(v\) is introduced at \(t\)).
We will think of a nice tree decompositions as rooted trees.

A (rooted) tree decomposition \((T, \{X_t\}_{t \in \mathcal{V}(T)})\) is **nice** if the following conditions are satisfied:

1. **Introduce node:** a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{v\}\) for some vertex \(v \not\in X_{t'}\) (we say that \(v\) is introduced at \(t\)).

2. **Forget node:** a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{w\}\) for some vertex \(w\) (we say that \(w\) is forgotten at \(t\)).
We will think of a nice tree decompositions as rooted trees.

A (rooted) tree decomposition \((T, \{X_t\}_{t \in V(T)})\) is **nice** if the following conditions are satisfied:

1. **Introduce node:**
   - A node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{v\}\) for some vertex \(v \notin X_{t'}\) (we say that \(v\) is introduced at \(t\)).

2. **Forget node:**
   - A node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{w\}\) for some vertex \(w \notin X_{t'}\) (we say that \(w\) is forgotten at \(t\)).

3. **Join node:**
   - A node \(t\) with two children \(t_1, t_2\) such that \(X_t = X_{t_1} = X_{t_2}\).
Nice Tree Decomposition

- We will think of a nice tree decompositions as rooted trees.
- A (rooted) tree decomposition \((T, \{X_t\}_{t \in V(T)})\) is nice if the following conditions are satisfied:
  
  - \(X_r = \emptyset\) for \(r\) the root of \(T\) and \(X_l = \emptyset\) for every leaf \(l\) of \(T\).
  
  - Every non-leaf node of \(T\) is of one of the following three types:
    1. Introduce node: a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{v\}\) for some vertex \(v \not\in X_{t'}\) (we say that \(v\) is introduced at \(t\)).
    2. Forget node: a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{w\}\) for some vertex \(w\) (we say that \(w\) is forgotten at \(t\)).
    3. Join node: a node \(t\) with two children \(t_1, t_2\) such that \(X_t = X_{t_1} = X_{t_2}\).
    4. Introduce edge node*: a node \(t\), labeled with an edge \(uv \in E(G)\) such that \(u, v \in X_t\), and with exactly one child \(t'\) such that \(X_t = X_{t'}\) (We say that edge \(uv\) is introduced at \(t\)).
We will think of a nice tree decompositions as rooted trees.

A (rooted) tree decomposition \((T, \{X_t\}_{t \in V(T)})\) is **nice** if the following conditions are satisfied:

- \(X_r = \emptyset\) for \(r\) the root of \(T\) and \(X_t = \emptyset\) for every leaf \(t\) of \(T\).
- Every non-leaf node of \(T\) is of one of the following three types:
  1. **Introduce node:** a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{v\}\) for some vertex \(v \notin X_{t'}\) (we say that \(v\) is introduced at \(t\)).
  2. **Forget node:** a node \(t\) with exactly one child \(t'\) such that \(X_t = X_{t'} \cup \{w\}\) for some vertex \(w \in X_{t'}\) (we say that \(w\) is forgotten at \(t\)).
  3. **Join node:** a node \(t\) with two children \(t_1, t_2\) such that \(X_t = X_{t_1} = X_{t_2}\).
  4. **Introduce edge node:** a node \(t\), labeled with an edge \(uv \in E(G)\) such that \(u, v \in X_t\), and with exactly one child \(t'\) such that \(X_t = X'_t\). (We say that edge \(uv\) is introduced at \(t\)).

Isn't the nice tree decomposition width bigger then the TW of the graph?
Nice Tree Decomposition

- **Lemma 2.** If a graph $G$ admits a tree decomposition of width at most $k$, then it also admits a nice tree decomposition of width at most $k$.

- Moreover, given a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of $G$ of width at most $k$, one can in time $O(k^2 \cdot \max(|V(T)|, |V(G)|))$ compute a nice tree decomposition of $G$ of width at most $k$ that has at most $O(k|V(G)|)$ nodes.

- Proof of the lemma and the algorithm of computing a (nice) tree decomposition are out of the scope of this lecture. Therefore, we will assume that such a decomposition is provided on the input together with the graph.
Dynamic Programming Reminder

- Dynamic Programming (DP) is a technique to solve problems by breaking them down into overlapping sub-problems which follows the optimal substructure.
- Dynamic programming is a class of problems where it is possible to store results for recurring computations in some lookup so that they can be used when required again by other computations.
- This improves performance at the cost of memory.
- We will focus on dynamic programming for graphs of bounded treewidth.
Dynamic Programming Reminder

- Dynamic Programming (DP) is a technique to solve problems by breaking them down into overlapping sub-problems which follows the optimal substructure.
- Dynamic programming is a class of problems where it is possible to store results for recurring computations so that they can be used when required again by other computations.
- This improves performance at the cost of memory.
- We will focus on dynamic programming on graphs of bounded treewidth.

Which graphs are of bounded treewidth?
Dynamic Programming on graphs of bounded treewidth

- Examples of graphs with bounded treewidth:
- Pseudoforest graph:
  every connected component has at most one cycle.

\[ \text{treewidth} = 2 \]
Dynamic Programming on graphs of bounded treewidth

- Examples of graphs with bounded treewidth:
  - Cactus graph:
    any two simple cycles have at most one vertex in common

  treewidth = 2
Dynamic Programming on graphs of bounded treewidth

- Examples of graphs with bounded treewidth:
  - outerplanar graph:
    - has a planar drawing for which all vertices belong to the outer face of the drawing.
    - $\text{treewidth} = 2$
Dynamic Programming on graphs of bounded treewidth

- Examples of graphs with bounded treewidth:
- Control flow graph (compilation):
  all paths that might be traversed through a program during its execution.

\[ \text{treewidth} \leq 6 \]
Maximum Weighted Independent Set

- **Independent Set:** Given an undirected graph $G = (V, E)$ an independent set (IS) of $G$ is a subset $S \subseteq V$, such that no two of its vertices are adjacent.

- The problem: Given an undirected graph $G = (V, E)$ and a weight function on its vertices $w: V \to \mathbb{R}^+$, find a subset $S \subseteq V$ such that $S \in IS$ and $\forall S' \in IS : w(S) \geq w(S')$.

- The maximum weighted independent set is known to be NP-hard.

- Therefore, it is unlikely that there exists an efficient algorithm for solving it.

- However, we will now see a dynamic-programming-based algorithm that solves it efficiently on graphs of bounded treewidth.
Let $G=(V,E)$ be a graph of $n$-vertex with width of at most $k$.

And let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of $G$.

By applying Lemma 2 we can assume that $\mathcal{T}$ is a nice tree decomposition.
Weighted Independent Set – DP alg.

- Let $G=(V,E)$ be a graph of $n$-vertex with width of at most $k$.
- And let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of $G$.
- By applying Lemma 2 we can assume that $\mathcal{T}$ is a nice tree decomposition.
Weighted Independent Set – DP alg intro.

- Recall that $T$ is rooted at some node $r$.
- For a node $t$ of $T$, let $V_t$ be the union of all the bags present in the subtree of $T$ rooted at $t$, including $X_t$.
- Provided that $t \neq r$ we can apply Lemma 1 to the edge of $T$ between $t$ and its parent, and infer that $\partial(V_t) \subseteq X_t$.
- The same conclusion is trivial when $t = r$. 
Weighted Independent Set – DP alg intro.

Recall that $T$ is rooted at some node $r$.

For a node $t$ of $T$, let $V_t$ be the union of all the bags present in the subtree of $T$ rooted at $t$, including $X_t$.

Provided that $t \neq r$ we can apply Lemma 1 to the edge of $T$ between $t$ and its parent, and infer that $\partial(V_t) \subseteq X_t$.

The same conclusion is trivial when $t = r$.

**Lemma 1.**

$K = \bigcup_{t \in V(T_a)} X_t$, and

$M = \bigcup_{t \in V(T_b)} X_t$, $(K, M)$ is a separation of $G$ with separator $X_a \cap X_b$. 

**Diagram:**

- $V_t$ corresponds to the subtree rooted at node $t$.
- The graph $G$ is represented with nodes $a, b, c, d, e, f$ and edges connecting them.
Among independent sets $I$ satisfying $I \cap X_t = S$ for some fixed $S$, all the maximum-weight solutions have exactly the same weight of the part contained in $V_t$. 
Weighted Independent Set – DP alg.

- Among independent sets $I$ satisfying $I \cap X_t = S$ for some fixed $S$, all the maximum-weight solutions have exactly the same weight of the part contained in $V_t$.
- For every node $t$ and every $S \subseteq X_t$, define the following value:

$$c[t,S] = \text{maximum possible weight of a set } \hat{S} \text{ such that }$$

$$S \subseteq \hat{S} \subseteq V_t, \hat{S} \cap X_t = S, \text{ and } \hat{S} \text{ is independent.}$$
Weighted Independent Set – DP alg.

- Among independent sets $I$ satisfying $I \cap X_t = S$ for some fixed $S$, all the maximum-weight solutions have exactly the same weight of the part contained in $V_t$.
- For every node $t$ and every $S \subseteq X_t$, define the following value:

$$c[t, S] = \text{maximum possible weight of a set } \hat{S} \text{ such that }$$

$$S \subseteq \hat{S} \subseteq V_t, \hat{S} \cap X_t = S, \text{and } \hat{S} \text{ is independent.}$$

- If no such set $\hat{S}$ exists, then we put $c[t, S] = -\infty$ (iff $S$ is not independent)
- Final solution is $c[r, \emptyset]$. 
Weighted Independent Set – DP alg.

- Among independent sets $I$ satisfying $I \cap X_t = S$ for some fixed $S$, all the maximum-weight solutions have exactly the same weight of the part contained in $V_t$.
- For every node $t$ and every $S \subseteq X_t$, define the following value:

$$c[t, S] = \text{maximum possible weight of a set } \hat{S} \text{ such that }$$

$$S \subseteq \hat{S} \subseteq V_t, \hat{S} \cap X_t = S, \text{and } \hat{S} \text{ is independent.}$$

- If no such set $\hat{S}$ exists, then we put $c[t, S] = -\infty$ (iff $S$ is not independent)
- Final solution is $c[r, \emptyset]$.
- Now all we have to do is defining our recursive formulas for bottom-up DP.
Weighted Independent Set
- DP alg.

- Thanks to the definition of nice tree decomposition we have only few cases of how a bag relates to its children.
- The computation $C[t, S]$ for each node is based only on the values that were computed already for the children of this node.
- **Leaf node.** If $t$ is a leaf node, then we have only one value $c[t, \emptyset] = 0$. 

Leaf node
Weighted Independent Set – DP alg.

- **Introduce node.** Suppose $t$ is an introduce node with child $t'$ such that $X_t = X'_t \cup \{v\}$ for some $v \notin X'_t$. Let $S$ be any subset of $X_t$. If $S$ is not independent, then we can immediately put $c[t, S] = -\infty$; hence assume otherwise.

Then we claim that the following formula holds:

$$c[t, S] = \begin{cases} c[t', S] & \text{if } v \notin S; \\ c[t', S \setminus \{v\}] + w(v) & \text{otherwise.} \end{cases}$$
Weighted Independent Set – DP alg.

Proof of *:

- $v \in S$. Let $\hat{S}$ be the set that maximize $c[t, S]$.
- $\hat{S} \setminus \{v\}$ was considered in the calculation of $c[t', S \setminus \{v\}]$ (no other vertices were added, here nice tree helps us)
- $\Rightarrow c[t', S \setminus \{v\}] \geq w(\hat{S} \setminus \{v\}) = w(\hat{S}) - w(v) = c[t, S] - w(v)$.
- $\Rightarrow c[t, S] \leq c[t', S \setminus \{v\}] + w(v)$. 

$$c[t, S] = \begin{cases} 
    c[t', S] & \text{if } v \notin S; \\
    c[t', S \setminus \{v\}] + w(v) & \text{otherwise.}
\end{cases}$$
Weighted Independent Set – DP alg.

Proof of \(*\) (cont.):

- \(v \in S\). Let \(\hat{S}'\) be the set that maximize \(c[t', S \setminus \{v\}]\).
- \(S\) is independent (as we assume before) so \(v\) doesn’t have neighbors in \(S \setminus \{v\} = \hat{S}' \cap X_t\),
- Moreover, by lemma 1, \(v\) doesn’t have any neighbor in \(V_t' \setminus X_t \supseteq \hat{S}' \setminus X_t\),
- \(\Rightarrow v\) Doesn’t have neighbor in \(\hat{S}'\)
- \(\Rightarrow \hat{S}' \cup \{v\}\) is independent Set.
- \(\hat{S}' \cup \{v\}\) intersects with \(X_t\) only at \(S\) so this set was consider for \(c[t, S]\).
Proof of * (cont.):

Now we can conclude:

\[ c[t, S] \geq w(\hat{S}' \cup \{v\}) = w(\hat{S}') + w(v) = c[t', S \setminus \{v\}] + w(v). \]

And from both conclusions we get:

\[ c[t, S] = c[t', S \setminus \{v\}] + w(v) \text{ for the case } v \in S. \]
Weighted Independent Set – DP alg.

- **Forget node.** Suppose \( t \) is a forget node with child \( t' \) such that \( X_t = X_{t'} \setminus \{w\} \) for some \( w \in X_t \). Let \( S \) be any subset of \( X_t \); again we assume that \( S \) is independent, since otherwise we put \( c[t, S] = -\infty \).

We claim that the following formula holds:

\[
{c[t, S]} = \max \left\{ {c[t', S], c[t', S \cup \{w\}]} \right\}.
\]

- Proof: Let \( \hat{S}' \) be a set that maximize \( c[t, S] \).
  If \( w \notin \hat{S}' \), so \( \hat{S}' \) was considered when calculating \( c[t', S] \) and hence \( c[t', S] \geq w(\hat{S}') = c[t, S] \).
  Else, If \( w \in \hat{S}' \) so \( \hat{S}' \) was considered when calculating \( c[t', S \cup \{w\}] \) and hence \( c[t', S \cup \{w\}] \) \( \geq w(\hat{S}') = c[t, S] \).
Weighted Independent Set – DP alg.

- **Forget node.** Suppose $t$ is a forget node with child $t'$ such that $X_t = X_{t'} \setminus \{w\}$ for some $w \in X_t$. Let $S$ be any subset of $X_t$; again we assume that $S$ is independent, since otherwise we put $c[t, S] = -\infty$.

We claim that the following formula holds:

$$c[t, S] = \max \left\{ c[t', S], c[t', S \cup \{w\}] \right\}.$$
Weighted Independent Set – DP alg.

- **Forget node.** Suppose \( t \) is a forget node with child \( t' \) such that \( X_t = X_{t'} \setminus \{w\} \) for some \( w \in X_t \). Let \( S \) be any subset of \( X_t \); again we assume that \( S \) is independent, since otherwise we put \( c[t, S] = -\infty \).

We claim that the following formula holds:

\[
c[t, S] = \max \left\{ c[t', S], c[t', S \cup \{w\}] \right\}.
\]
Weighted Independent Set – DP alg.

- **Join node.** Finally, suppose that $t$ is a join node with children $t_1, t_2$ such that $X_t = X_{t_1} = X_{t_2}$. Let $S$ be any subset of $X_t$; as before, we can assume that $S$ is independent. The recursive formula is as follows:

$$c[t, S] = c[t_1, S] + c[t_2, S] - w(S).$$

- Proof idea: from Lemma 1 we know that each part of $t$’s children is separated and the border is inside $X_t$ vertices. We take all of $S(\subseteq X_t)$ to be in $c[t, S]$. So we will consider the best solution of each child for $S$ and subtract its size once (because we took it twice).
Run-time analysis of the algorithm

- We have treewidth of at most $k$, which means $|X_t| \leq k + 1$ for every node $t$.
- Thus for every node $t$ we compute $2^{|X_t|} \leq 2^{k+1}$ values of $c[t, S]$.
- In naive solution we will say that each $c[t, S]$ computed in $n^{O(1)}$ time. It is possible to construct a data structure that allows performing adjacency queries in time $O(k)$, so computing each $c[t, S]$ will take only $k^{O(1)}$ time.
- We assumed that the number of nodes of the given tree decompositions is $O(kn)$ (Lemma 2).
- The total running time of the algorithm is $2^k \cdot k^{O(1)} \cdot n$.

We’ve got a FPT algorithm for a problem that is known to be NP-hard!
Run-time analysis of the algorithm

We have treewidth of at most $k$, which means $|X_t| \leq k + 1$ for every node $t$. Thus for every node $t$ we compute $2^{|X_t|} \leq 2^{k+1}$ values of $c[t, S]$.

- In naive solution we will say that each $c[t, S]$ computed in $n^{O(1)}$ time. It is possible to construct a data structure that allows performing adjacency queries in time $O(k)$, so computing each $c[t, S]$ will take only $k^{O(1)}$ time.
- we assumed that the number of nodes of the given tree decompositions is $O(kn)$ (Lemma 2)
- the total running time of the algorithm is $2^k \cdot k^{O(1)} \cdot n$

We’ve got a FPT algorithm for a problem that is known to be NP-hard!

Just for graphs of bounded treewidth though
Monadic second-order Logic

- In the course of Logic we saw First-Order Logic, where we can use quantifiers only over variables that range over individuals ($\forall x$ and $\exists x$).
  $$\forall x \forall y (R(x, y) \rightarrow \neg R(y, x))$$  
  (a – symmetric def.)
- **second-order logic** is an extension of it, that allows us to use quantifiers over relations, functions and sets of elements.
- **monadic second order logic (MSO)** is the fragment of second-order logic where the second-order quantification is limited to be only over sets.
- **MSO$_2$** allows quantification over sets of vertices or edges.
- Our main interest here is to use MSO to describe properties of undirected graphs.
- We view an undirected graph as relational structure (i.e. a model as in logic), where the universe is the vertices and there is one binary relation $E(x, y)$ for the edges.
- Example for such **MSO$_2$** formula that express 3-coloring in a graph:

$$\exists X_1 \exists X_2 \exists X_3 \ \forall x \bigvee_{i} X_i \land \ \forall x \forall y \ E(x, y) \rightarrow \bigvee_{i \neq j} X_i(x) \land X_j(x)$$
Courcelle's theorem

- **Courcelle's theorem.** Assume that $\phi$ is a formula of $\text{MSO}_2$ and $G$ is an $n$-vertex graph. Suppose, moreover, that a tree decomposition of $G$ of width $t$ is provided. Then there exists an algorithm that verifies whether $\phi$ is satisfied in $G$ in time $f(||\phi||, t) \cdot n$, for some computable function $f$.

- In other words, every graph property definable in the monadic second-order logic of graphs can be decided in linear time on graphs of bounded treewidth.
Courcelle's theorem

- **Courcelle's theorem.** Assume that $\phi$ is a formula of $MSO_2$ and $G$ is an $n$-vertex graph. Suppose, moreover, that a tree decomposition of $G$ of width $t$ is provided. Then there exists an algorithm that verifies whether $\phi$ is satisfied in $G$ in time $f(||\phi||, t) \cdot n$, for some computable function $f$.

- In other words, every graph property definable in the monadic second-order logic of graphs can be decided in linear time on graphs of bounded treewidth.

**Problems in $G$ in time $k^{O(k)} \cdot n$:**

- **Steiner Tree,**
- **Feedback Vertex Set,**
- **Hamiltonian Path and Longest Path,**
- **Hamiltonian Cycle and Longest Cycle,**
- **Chromatic Number,**
- **Cycle Packing,**
- **Connected Vertex Cover,**
- **Connected Dominating Set,**
- **Connected Feedback Vertex Set.**
Courcelle's theorem

Courcelle's theorem. Assume that $\phi$ is a formula of $MSO_2$ and $G$ is an $n$-vertex graph. Suppose, moreover, that a tree decomposition of $G$ of width $t$ is provided. Then there exists an algorithm that verifies whether $\phi$ is satisfied in $G$ in time $f(||\phi||, t) \cdot n$, for some computable function $f$.

In other words, every graph property definable in the monadic second-order logic of graphs can be decided in linear time on graphs of bounded treewidth.

- **Vertex Cover and Independent Set** in time $2^k \cdot k^{O(1)} \cdot n$,
- **Dominating Set** in time $4^k \cdot k^{O(1)} \cdot n$,
- **Odd Cycle Transversal** in time $3^k \cdot k^{O(1)} \cdot n$,
- **MaxCut** in time $2^k \cdot k^{O(1)} \cdot n$,
- **$q$-Coloring** in time $q^k \cdot k^{O(1)} \cdot n$. 

So which more problems are tractable now?
That’s all, Thank you for listening

Any questions?