KAIST, School of Computing, Spring 2025 | **Scribed By: Eunjung KIM**
Graph classes, algorithms and logic (CS492) | **Tree-decomposition and tree languages**
Lecture: Eunjung KIM | **Week 3: 11, 13 March 2025**

Contents

---

1 Tree-decomposition

We follow the standard graph terminology, e.g. as in Diestel [1].

Definition 1. *Let G be a graph. A* tree decomposition *of G is a pair (T, χ), where T is a tree and $\chi : V(T) \to 2^{V(G)}$ is a mapping associating each tree node t to a vertex subset $\chi(t) \subseteq V(G)$ called a* bag, *which satisfies the following conditions.*

- **Vertex Coverage.** $\bigcup_{t \in V(T)} \chi(t) = V(G)$.
- **Edge Coverage.** *For every edge $e = uv$ of G, there is a tree node t of T such that $u, v \in \chi(t)$.*
- **Connectivity.** *For every vertex v of G, the nodes of T whose bags contain v is connected in T. In other words, the set $\{t \in V(T) : v \in \chi(t)\}$ is a subtree of T for each v of G.*

The width *of a tree decomposition (T, χ), written as* $\mathsf{width}(T, \chi)$, *equals* $\max_{t \in V(T)} |\chi(t)| - 1$. *The* treewidth *of G, denoted as* $\mathsf{tw}(G)$ *is*

$$\mathsf{tw}(G) := \min_{(T,\chi)} \mathsf{width}(T, \chi),$$

where (T, χ) is taken over all tree decompositions of G.

A *rooted* tree decomposition is a tree decomposition (T, χ) where T is a rooted tree. When T is rooted, one can talk about a child/parent/ancestor/descendant of a tree node.

We call the set $\beta(t)$ the *bag* of t. The *adhesion* of a node t, denoted $\mathsf{adh}_{(T,\beta)}(t)$, is the intersection of $\beta(t)$ with the bag of its parent unless t is the root in which case $\mathsf{adh}_{(T,\beta)}(t) = \emptyset$. The *cone* of a node t, denoted $\mathsf{cone}_{(T,\beta)}(t)$, is the set of vertices $\bigcup_{s \in V(T_t)} \beta(s)$. If the tree-decomposition (T, β) is clear from the context we omit the subscript (T, β) from the notation and write $\mathsf{adh}(t)$ and $\mathsf{cone}(t)$.

Let $A, B \subseteq V(G)$ be vertex subsets of G; note that they may intersect. An (A, B)*-path* in G is a path with one endpoint in A and another endpoint in B. If there is a vertex v in the intersection $A \cap B \neq \emptyset$, then v itself forms a trivial (A, B)-path. We say that a vertex set $X \subseteq V(G)$ *separates* A and B, or say that X *is an* (A, B)*-separator in* G if every (A, B)-path contains a vertex of X. Note that any (A, B)-separator must contain $A \cap B$ fully. In the special case when $A = \{a\}$ and $B = \{b\}$ for some $a, b \in V(G)$, we write (a, b)-separator instead of (rather cumbersome) $(\{a\}, \{b\})$-separator. A vertex set X is called a *separator* of G if there $a, b \in V(G) \setminus X$ such that X is an (a, b)-separator.

Here are some basic properties of a tree decomposition and treewidth of a graph.

Lemma 2. *Let (T, χ) be a tree decomposition of G. The following holds.*

1. *For any clique K of G, there is a tree node t of T such that $K \subseteq \chi(t)$.*

2. *Let xy be an edge of T and let T_x (respectively T_y) be the subtree of $T - xy$ containing x (respectively, with y). Then $\chi(x) \cap \chi(y)$ is a separator of $\bigcup_{t \in V(T_x)} \chi(t)$ and $\bigcup_{t \in V(T_y)} \chi(t)$. In particular, there is no edge between $\bigcup_{t \in V(T_x)} \chi(t) \setminus (\chi(x) \cap \chi(y))$ and $\bigcup_{t \in V(T_y)} \chi(t) \setminus (\chi(x) \cap \chi(y))$.*

Proof: We prove the last property only. Consider a mapping $\mathsf{top} : V(G) \to V(T)$ defined as

$$\mathsf{top}(v) := \text{the topmost node of } T \text{ containing } v.$$

In other words, top chooses the node of T whose distance to root is the minimum among all nodes whose bags contain v.

We first argue that top is well-defined, i.e. there is a unique such node achieving the minimum distance to root. Suppose the contrary, i.e. there are two distinct nodes x, y of T with $v \in \chi(x)$ and $v \in \chi(y)$ neither of which is an ancestor of the other. By the connectivity property of tree decomposition, all the nodes on the (x, y)-path P of T contains v in their bags. In particular, the least common ancestor of x and y, which must be distinct from both x and y, lies on P and contains v. This contradicts the choice of x and y.

Next, we claim that top is surjective, i.e. for every tree node t there is a vertex w of G such that $\mathsf{top}(w) = t$. Indeed, the non-redundancy of (T, χ) subsumes $\chi(\mathsf{root}) \neq \emptyset$ and for every vertex $w \in \chi(\mathsf{root})$, $\mathsf{top}(w) = \mathsf{root}$. If t is an internal node with a parent node $p(t)$, the set $\chi(t) \setminus \chi(p(t)) \neq \emptyset$ due to the non-redundancy of the given tree decomposition. For each $w \in \chi(t) \setminus \chi(p(t))$, the mapping top maps w to t. This proves that top is surjective, implying $|V(T)| \leq |V(G)|$. $\square$

Lemma 3. *Any subdivision of any graph G has treewidth at most* $\mathsf{tw}(G)$.

Nice tree decomposition. A nice tree decomposition is a tree decomposition which is tailored to ease the design and description of a dynamic programming algorithm over a tree decomposition.

Definition 4. *Let G be a graph. A* nice tree decomposition *(T, χ) of G is a rooted tree decomposition such that each tree node t of T falls into one of the next four types and satisfies the corresponding property.*

- **Leaf node.** *t is a leaf of T.*

- **Introduce node.** *t has exactly one child, say t', in T and it holds that $\chi(t) = \chi(t') \cup \{v\}$ for some vertex v of G.*

- **Forget node.** *t has exactly one child, say t', in T and it holds that $\chi(t) = \chi(t') \setminus \{v\}$ for some vertex v of G.*

- **Join node.** *t has exactly two children, say t_1 and t_2, in T and it holds that $\chi(t) = \chi(t_1) = \chi(t_2)$.*

Lemma 5. *Let (T, χ) be a tree decomposition of G of width w. In linear time, one can obtain a nice tree decomposition (T', χ') of G of width w. Moreover,*

- *one can further impose the condition that all leaf nodes and the root of T' has empty bags, and*
- *the number of nodes in T' is at most $4wn$, where $n := |V(G)|$.*

Let us fix some notations for a rooted tree decomposition (T, χ) of G. The root of T is written as roote. For a tree node t of T, T_t denotes the subtree of T rooted at t; that is, the set of nodes in T_t is precisely the set of all tree nodes x such that the (unique) path between x and root in T intersects t. We define

$$V_t := \bigcup_{b \in V(T_t)} \chi(b), \qquad\qquad G_t := G[V_t].$$

A useful property of a nice tree decomposition for designing DP algorithm is that the vertex v introduced in an introduce node has all its neighbors in G_t in the bag containing v. This property is a simple corollary of (iii) in Lemma 2 applied for the specific setting of a nice tree decomposition.

Lemma 6. *Let (T, χ) be a nice tree decomposition of G. Let t be an introduce node of T with a child t' and v be the vertex of G such that $\chi(t) = \chi(t') \cup \{v\}$. Then $N_{G_t}(v) \subseteq \chi(t)$.*

A similar consequence of Lemma 2 to join node is stated in the next lemma.

Lemma 7. *Let (T, χ) be a nice tree decomposition of G. Let t be a join node of T with two children t_1 and t_2. Then $V_{t_1} \cap V_{t_2} = \chi(t)$.*

In a nice tree-decomposition, each internal node has at most two children and this makes the presentation of an algorithm over the tree-decomposition convenient. We will use a *binary* tree-decomposition for proving Courcelle's Theorem, that is, a tree-decomposition such that each internal node has exactly two children. A nice tree-decomposition can be easily converted into a binary tree-decomposition of the same width.

Lemma 8. *There is a linear-time algorithm that, given a tree-decomposition of G whose width is at most ω, outputs a binary tree-decomposition of width at most ω.*

There is an efficient algorithm for approximating the treewidth of a graph in the following sense. It is one of the basic algorithms, conceptually simple and elegant. There are many expositions of the algorithm initially proposed by Reed [2], for example see [3] (link).

Theorem 9 (Computing treewidth)**.** *[2] There is an algorithm which, given an input consisting of a graph G and $\omega \in \mathbb{N}$, either outputs a tree-decomposition (T, β) of width at most $4\omega + 1$ or correctly decides that the treewidth of G is bigger than ω in time $27^{\omega} \cdot \omega nm$.*

A modern treatment of algorithm computing treewidth is available thanks to Korhonen, which reduces the dependency on n from cubic in [2] to linear function of n.

Theorem 10 (Computing treewidth)**.** *[4] There is an algorithm which, given an input consisting of a graph G and $\omega \in \mathbb{N}$, either outputs a tree-decomposition (T, β) of width at most $2\omega + 1$ or correctly decides that the treewidth of G is bigger than ω in time $2^{O(\omega)} \cdot n$.*

2 Tree language and tree automata

We want to extend the notion of a language as a set of strings, to the notion of tree language. Many results on strings transfer to trees.

Tree language. We start by introduction a tree-analogue of a string. Let Σ be an alphabet. A string s can be seen as a pair (U, ρ), where U is a ground set equipped with a linear order $<$ (or "successor relation", which are equivalent to linear order for MSO logic) and $\rho : U \to \Sigma$. The mapping ρ simply matches a letter from Σ to each position of the string. Let us extend the notion of string to a tree.

We say that a tree is *rooted* if it has a unique distinguished node called the *root*. For a rooted tree, a child/parent/ancestor/descendant can be defined in the usual way. For now, we only consider a rooted tree where each node has at most two children.

We define a Σ-*tree* as a rooted tree such that each node is labeled by a letter from the alphabet Σ and each internal node has precisely two children[1], one *left* child and one *right* child. Note that if the tree is simply a path, then a Σ-tree is precisely a string. We denote a Σ-tree as a pair (T, ρ) where T is a rooted binary tree with root r, in which each internal node has exactly two children, and a mapping $\rho : N \to \Sigma$. We call each vertex of the tree a *node* and the node set of T is written as $N(T)$ to distinguish them from graphs. A *tree language* (of binary trees) over Σ is a set of Σ-trees.

Tree automaton. Recall that a (deterministic) finite automaton for a usual string language is a 5-tuple in the form $(Q, \Sigma, \delta, q_0, F)$ where the transition function δ maps a pair $(q, a) \in Q \times \Sigma$ to a state in Q.

A *(deterministic) tree automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ and the only difference from the string automaton is the transition function δ. Now it maps $Q \times Q \times \Sigma$ to Q. That is, when the automaton runs on node x of T, it will read the states q_1 and q_2 of at its left and right child, and depending on the letter of Σ assigned to x the state at x will be determined using the transition function δ. When x is a leaf, the state of x is $\delta(q_0, q_0, \rho(x))$.

The *run* of tree automaton M on a Σ-tree is a function $\gamma : N(T) \to Q$ such that

1. for every leaf x of T, $\gamma(x) = \delta(q_0, q_0, \rho(x))$,
2. for every internal node x with a left child y_1 and right child y_2, $\gamma(x) = \delta(\gamma(y_1), \gamma(y_2), \rho(x))$, and
3. at the root node r, it holds that $\gamma(r) \in F$.

Due to the way that the automaton runs, we also call this automaton as *bottom-up tree automaton.*

3 Equivalence of MSO-definability and recognizability

One can easily represent a Σ-tree as a relational structure over τ, where the vocabulary τ consists of

- child_i: a binary predicate. $\mathsf{child}_1(x, y)$ means "x is the left child of y" and $\mathsf{child}_2(x, y)$ means "x is the right child of y".

[1]Tree language with more flexible trees can be defined and most of the results for the restricted tree language generalizes, but in this class we adhere to the basic setting.

- P_a for each letter $a \in \Sigma$: $P_a(x)$ means that “node x is assigned with letter $a \in \Sigma$ in the Σ-tree (T, ρ).

With the vocabulary τ, it is clear how a Σ-tree should be expressed as a τ-structure. Let (T, ρ) be a Σ-tree. Then the corresponding τ-structure has $N(T)$ as the universe, and the binary predicates child_i for $i = 1, 2$ and the unary predicates P_a for $a \in \Sigma$ are interpreted in $N(T)$ in the obvious way.

Theorem 11. *[5] A tree language over Σ is regular if and only if it* MSO*-definable.*

The following is an immediate consequence of Theorem 11.

Corollary 12. *An* MSO*-definable tree language can be recognized in linear time.*

References

[1] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[2] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC ’92, page 221–228, New York, NY, USA, 1992. Association for Computing Machinery.

[3] Kim Eun Jung. Lecture note for algorithms for NP-hard problems, CS492 spring semester, KAIST.

[4] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–192, 2022.

[5] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory*, 2:57–81, 1968.