Treewidth

Sven Driessen

January 13, 2022

Introduction

While many important algorithmic problems on graphs are NP-hard and therefore not computationally tractable, instances that arise in real world problems often have a certain structure. For example, the vertices in street maps mostly have a relatively small degree. This structure may be used to build algorithms with an improved running time. A particularly well-structured class of graphs are *trees*. Therefore, it may be desirable to find a way of measuring how much a graph is structered "like" a tree. One approach is to *decompose* a given graph into a tree, where the nodes of the tree are sets of graph vertices that fulfill certain properties. The "tree-ness" of the graph can then be measured by the minimum size of the nodes of its tree-decompositions, which we call its *tree width*. With these tools, we can now restrict classes of problem-instances in their tree-width, which proves to be very beneficial to the complexity of the given problem, while at the same time not restricting the problem instances so much that the resulting algorithms lose their usefulness in real-world applications. The main result here is Courcelle's theorem, which can be found e.g. in [Flum and Grohe(2006)]. It roughly states that problems on graphs with bounded treewidth can be solved in polynomial time. Of course, in order for the tool of treewidth and tree-decompositions to be useful in practice, we need to find a method of computing them efficiently. In the following sections we formally introduce the notion of tree-decompositions and treewidth, give some important examples and characterizations, and finally present the basics of a well-known linear-time algorithm for computing treewidth, which is based on reductions to smaller graphs using contraction along maximum matchings and deletion of suitable vertices.

1 Tree-Decompositions

Definition 1.1 (tree decomposition)**.** Let $G = (V, E)$ be a graph. A *tree-decomposition* of G is a pair $\mathcal{T} = (T, X)$, where $T = (I, F)$ is a tree and $X = (X_i)_{i \in I} \subseteq \mathcal{P}(V)$ is a family of subsets X_i of V (also called *bags*), such that the following properties hold:

(i) $\bigcup_{i \in I} X_i = V$.

(ii) For all $\{u, v\} \in E$, there exists an $i \in I$ with $\{u, v\} \subseteq X_i$.

(iii) For all $v \in V$, $X^{-1}(v) := \{i \in I | v \in X_i\}$ is connected in T.

Note that for any two nodes $i, j \in I$, the last property is equivalent to $X_i \cap X_j \subseteq X_k$ for all k on the unique path between i and j in T. Intuitively, vertices of G correspond to subtrees in $\mathcal{T}$, which intersect if (but not only if) the corresponding vertices are adjacent in G. In this way, the structure of G is "preserved" in the tree-decomposition. The size of the bags X_i in $\mathcal{T}$ are an indication of the structural complexity of G, although they only provide an upper bound, since tree decompositions can contain arbitrarily large (up to $|V|$) bags, even if the graph G has a very low complexity. For example, there is always the trivial tree-decomposition, consisting of only one bag that contains all vertices of G. To formalize this idea, we intruduce the notion of *treewidth*:

Definition 1.2 (treewidth)**.** Let G be a graph and $\mathcal{T} = (T, X = (X_i)_{i \in I})$ a tree-decomposition of G. The *width* of $\mathcal{T}$ is defined as

$$\text{width}(\mathcal{T}) := \max \{|X_i| \mid i \in I\} - 1.$$

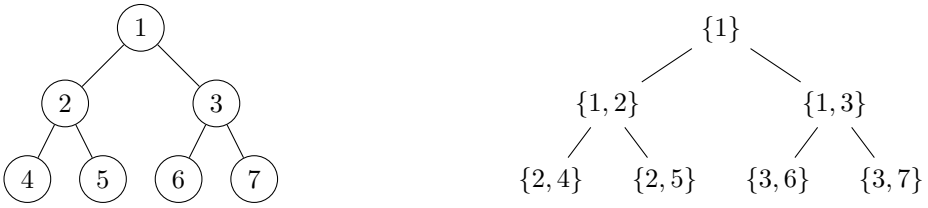
Then we define the *treewidth* of the graph G as

$$\mathrm{tw}(G) := \min\left(\{\mathrm{width}(\mathcal{T}) \mid \mathcal{T} \text{ tree-decomposition of } G\}\right).$$

We will now look at a few important examples. [1]

Example 1.3 (treewidth of trees)**.** A tree T can always be decomposed into a tree-decomposition with width 1 which is isomorphic to T. It is obtained by starting with T itself (i.e. every node of T becomes a bag of size one) and then adding the predecessor of every node to its corresponding bag. This construction is illustrated in the following example:

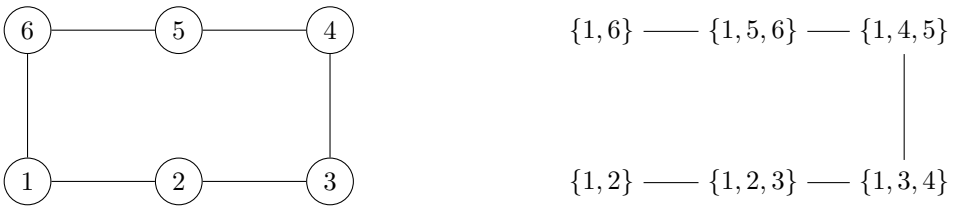
G : (T, X) :



In this sense, every tree "is" its own tree-decomposition. Therefore, the treewidth of a tree is always 1. In fact, the graphs with treewidth 1 are exactly the forests, i.e. the acyclic graphs ([Flum and Grohe(2006)]).

Example 1.4 (treewidth of cycles)**.** The treewidth of cycles is 2.
From the last example we already know that the treewidth of cycles can not be less than two. Conversely, the following example gives an idea how we can always construct a tree-decomposition with width 2:

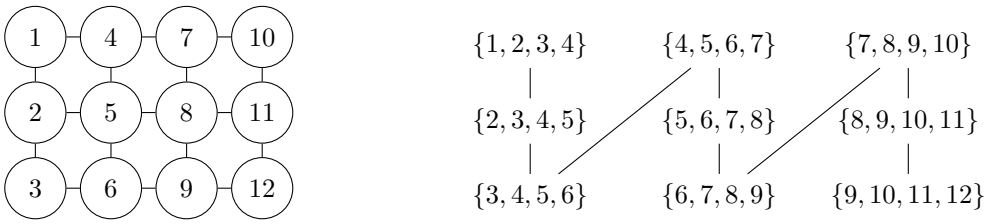
G : (T, X) :



In general, the tree-decomposition is constructed by moving along the cycle and adding a bag for every visited edge as well as an edge connecting the previous and the current bag. Aside from the two vertices of the corresponding edge, every bag also contains the start-vertex (here vertex 1) in order to satisfy condition (3) of tree-decompositions.

Example 1.5 (treewidth of grids)**.** The (m, n)-grid $G_{m,n}$ with m rows and n columns has treewidth $\mathrm{tw}(G_{m,n}) = \min(m, n)$ ([Flum and Grohe(2006)]). To see this, consider the following example:

G : (T, X) :



[1] While always having minimal width, the given tree-decompositions are always chosen such as to best convey the idea of the construction, and are therefore generally not minimal in the number of nodes.

The important oberservation in this tree-decomposition is that it is a single path such that every bag X_k seperates $\bigcup_{i<k} X_i$ from $\bigcup_{j\geq k} X_j$ in G, guaranteeing a certain "independence" of vertices in G whose corresponding subtrees do not intersect in T.

Example 1.5 hints at a connection between tree-decompositions and seperators in G. This connection is formulated in the following theorem. For a tree-decomposition $\mathcal{T} = (T = (I, F), X)$ of a graph G and a subset $J \subseteq I$, we define $X(J) := \bigcup_{j\in J} X_j$.

Theorem 1.6 ([Flum and Grohe(2006)])**.** *Let $G = (V, E)$ be a graph with a tree-decomposition $\mathcal{T} = (T, X)$ and let $\{i, j\}$ be an edge in T. Deleting the edge $\{i, j\}$ splits T into two disconnected subtrees, which we denote by T_i and T_j. Then the set $S = X_i \cap X_j$ seperates $X(T_i)$ from $X(T_j)$ in G, i.e. any path from $X(T_i)$ to $X(T_j)$ leads through S.*

Proof. Let p be a path from $X(T_i)$ to $X(T_j)$. Then p includes an edge $\{u, v\}$ with $u \in X(T_i)$ and $v \in X(T_j)$. We have $\{u, v\} \subseteq X_k$ for some bag X_k, and let w.l.o.g. $k \in T_j$. But then $u \in X(T_j)$, and therefore we have both $X^{-1}(u) \cap T_i \neq \emptyset$ and $X^{-1}(u) \cap T_j \neq \emptyset$. Since $X^{-1}(u)$ is connected, it follows that $\{i, j\} \subseteq X^{-1}(u)$ and thus $u \in X(T_i) \cap X(T_j) = S$. □

Using this theorem, we can prove the following result about cliques, which will be useful in section 3.2:

Corollary 1.7. *Let $\mathcal{T} = (T, X)$ be a tree-decomposition of a graph $G = (V, E)$ and $C \subseteq V$ be a Clique in G. Then there exists an $i \in I$ with $C \subseteq X_i$. In particular, if G contains a Clique of size k, then* $\mathrm{tw}(G) \geq k - 1$.

Proof. For $|C| = 1$ the statement clearly holds. Now let $|C| = n$ and assume per induction that the statement holds for all Cliques C' with $|C'| \leq n - 1$. In particular, for any fixed $v \in C$ there exist an $i \in I$ with $C' = C \setminus \{v\} \subseteq X_i$. Therefore, $T_{C'} := \{i \in I | C' \subseteq X_i\}$ is a nonempty subtree.
Assume $X^{-1}(v) \cap T_{C'} = \emptyset$, i.e. no bag in $T_{C'}$ contains v.
There is a unique edge $\{i, j\}$ in T connecting $T_{C'}$ and the connected component in $T \setminus T_{C'}$ containing $X^{-1}(v)$. Theorem 1.6 yields that $S = X_i \cap X_j$ seperates $X(T_i) \supseteq C'$ and $X(T_j) \supseteq \{v\}$ (where T_i and T_j denote the corresponding subtrees as in the theorem). But since C is a clique, for every $w \in C'$ there is an edge $\{w, v\}$ connecting $X(T_i)$ and $X(T_j)$, so either $w \in S$ or $v \in S$. Since $i \in T_{C'}$, per assumption we have $v \notin S \subseteq X_i$, so we get $C' \subseteq S = X_i \cap X_j \subseteq X^{-1}(v)$ and therefore $X^{-1}(v) \cap T_{C'} \neq \emptyset$, contradicting our assumption that this is false. Therefore $X^{-1}(v) \cap T_{C'} \neq \emptyset$, i.e. there exist an $i \in I$ with $X_i \supseteq C' \cup \{v\} = C$. □

Finally, the following similar result will also be important in 3.2:

Corollary 1.8 ([Bodlaender(1992)])**.** *Let $\mathcal{T} = (T, X)$ be a tree-decomposition of a graph $G = (V, E)$. If $W_1, W_2 \subseteq V$ induce a complete bipartite subgraph in G, i.e. $W_1 \times W_2 \subseteq E$ then there exists an $i \in I$ with either $W_1 \subseteq X_i$ or $W_2 \subseteq X_i$.*

2 A Game-Characterization of Treewidth

In example 1.5 we constructed a tree-decomposition by (intuitively speaking) moving along the graph and seperating the already visited vertices from the unvisited ones. This idea can be further developed into a characterization of treewidth by a Cops-and-Robber game:

There are k cops searching the graph. Every cop either occupies a vertex of the graph or is in a helicopter (and therefore currently removed from the graph.) In every step, cops on the graph can choose to enter a helicopter, and cops in a helicopter can choose to fly to an arbitrary vertex of the graph, leave the helicopter and occupy that vertex (in particular, they can only move in a helicopter.) The robber can move arbitrarily fast along the edges of the graph (in particular he can elude cops before they land on the graph,) but he cannot pass vertices occupied by a cop. The cops are always aware of the current position of the robber. The goal of the cops is then to catch the robber, i.e. to corner him such that a cop can be placed on his vertex without him being able to elude.
More formally, a *state* of the game is a tuple $(C_i, R_i), i \in \mathbb{N}$, where $C_i \subseteq V$ with $|C_i| \leq k$ (representing the placement of the cops) and $R_i \subseteq V$ is a connected component of the graph $G \setminus C_i$

obtained by deleting C_i from G (representing the location of the robber, whose exact position in the component is not relevant.) The starting state is $(C_0, R_0) = (\emptyset, V)$. To ensure correct transitions, for any $i \in \mathbb{N}$, we require $C_i \subseteq C_{i+1}$ or $C_{i+1} \subseteq C_i$ and accordingly $R_{i+1} \subseteq R_i$ or $R_i \subseteq R_{i+1}$. Then the cops have won if $R_i \subseteq C_{i+1}$ or $R_{i+1} = \emptyset$. We say that a graph can be searched by k cops if they have a winning strategy for the game.
The following result shows that the game is indeed a characterization of treewidth:

Theorem 2.1 ([Seymour and Thomas(1993)])**.** *Let G be a graph and $k \in \mathbb{N}$. Then the following are equivalent:*

- $\mathrm{tw}(G) \leq k - 1$
- *G can be searched by k cops*

The following example gives an idea for the strategy of the cops if $\mathrm{tw}(G) \leq k - 1$:

Example 2.2 (Trees)**.** Let $T = (V, E)$ be a tree. Then T can be searched by two cops, but not by one. If we have two cops, we can always leave one on the graph and move the other one edge into the component occupied by the robber, then remove the first cop and repeat until the robber is caught. If we only have one cop, the robber can always escape while the cop is removed from graph.

In genereal, if $\mathcal{T} = (T, X)$ is a tree-decomposition of G with width $\leq k - 1$, we can use the same strategy as in Example 2.2 using $\mathcal{T}$ to show that k cops can search G: First place cops on the vertices of a bag X_i, i.e. $C_1 = X_i$. Then there is a connected component J of $T \setminus \{i\}$ containing the robber, i.e. $R_1 \subseteq X(J)$. Let $\{i, j\}$ be the edge from i to the component J. By Theorem 1.6, we know that $X_i \cap X_j$ seperates $R_1 \subseteq X(J)$ from $G \setminus X(J)$. Therefore, if we choose $C_2 = X_i \cap X_j$ and $C_3 = X_j \supseteq X_i \cap X_j$, the robber is forced to choose $R_2 \subseteq X(J)$ and $R_3 \subseteq X(J)$ (and is therefore being "cornered" in $X(J)$.) Now it is easy to see that if we move along $\mathcal{T}$, repeating this strategy, the cops will eventually win: If the robber reaches a leaf j of $\mathcal{T}$, i.e. $R_i \subseteq X_j$, the cops win by setting $C_{i+1} = X_j \supseteq R_i$ (which is possible because $\mathrm{width}(\mathcal{T}) \leq k - 1$).

This proves one direction of Theorem 2.1. The second, more difficult direction can be found in [Seymour and Thomas(1993)].

3 Computing Treewidth

As mentioned in the Introduction, our goal is to utilize the notion of treewidth to develop efficient algorithms for certain classes of structures. Therefore is desirable to have an algorithm that computes the treewidth and a corresponding tree-decomposition of a given graph in polynomial time. While the problem in general is NP-complete ([Flum and Grohe(2006)]), it becomes tractable if we fix the treewidth k. In this section, we present the basics of one such algorithm, proposed by Hans Bodlaender in 1992. Bodlaenders main result states:

Theorem 3.1 ([Bodlaender(1992)])**.** *For all $k \in \mathbb{N}$, there exists a linear time algorithm that tests whether a given Graph $G = (V, E)$ has treewidth at most k, and if so, outputs a tree-decomposition of G with width at most k.*

The main idea of the algorithm is to recursively reduce the problem to smaller graphs. In each step, they are either constructed by contracting the graph along the edges of a maximal matching, or by deleting certain vertices. In this section we will develop these two sub-algorithms, which are both already correct algorithms for our problem. To achieve a linear running time, they will finally be combined into the full algorithm by Bodlaender.

From here on, let always $G = (V, E)$ be a graph and $k \in \mathbb{N}$.

The following observation shows that graphs with bounded tree-width are "sparse" in the sense that the number of edges is linearly bounded by the number of vertices, already hinting at the linear running time of the final algortihm. Furthermore, it provides a first necessary condition that allows us to easily identify graphs that are "too dense" to have treewidth at most k.

Lemma 3.2. *If* $\mathrm{tw}(G) \leq k$*, then* $|E| \leq k|V| - \frac{1}{2}k(k+1)$.

Proof. W.l.o.g., let $\mathrm{tw}(G) = k$. Every graph of treewidth k can be expanded (by adding edges) into a *k-tree*, which is a graph constructed by starting with a $k+1$-Clique and then adding new vertices that are made adjacent to an already existing k-Clique (and therefore forming a $(k+1)$-Clique with their neighbourhood.) This can be seen as follows:
First, given a tree-decomposition $\mathcal{T}'$ of G with width k, we can expand $\mathcal{T}$ to a *smooth* tree-decomposition $\mathcal{T} = (T, X)$ of the same width with the properties that $|X_i| = k+1$ for all $i \in I$ and $|X_i \cap X_j| = k$ for all edges $\{i, j\}$ in T (see [Bodlaender(1992)]). Given the smooth tree-decomposition $\mathcal{T}$, the graph G' obtained by adding edges $\{u, v\}$ to G for all $u, v \in V$ with $\{u, v\} \subseteq X_i$ for some $i \in I$ (i.e. by turning the vertices of every bag X_i into a clique in G') is a k-tree.
Therefore, the k-trees are exactly the edge-maximal graphs (with n vertices) of treewidth k, and their number of edges is given by

$$|E| = \underbrace{\binom{k+1}{2}}_{\text{edges of the first clique}} + \underbrace{k(n-(k+1))}_{\text{remaining edges}} = \frac{1}{2}k(k+1) + kn - k(k+1) = k|V| - \frac{1}{2}k(k+1)$$

as desired. □

3.1 Reduction by Contraction

Recall that a *matching* is a set $M \subseteq E$ of pairwise disjoint edges. Given a matching M in G, we obtain a new graph G/M by contracting G along the edges in M, i.e. merging all pairs u, v of vertices that are matched by M (i.e. $e = \{u, v\} \in M$) into a single new vertex w_e that is incident to all vertices that were incident to u or v. G/M is called the *contraction* of G along M. Given a matching M, we obtain a natural surjection $f_M : G \to G/M$ defined by $f_M(u) = f_M(v) = w_e$ if $e = \{u, v\} \in M$ and $f_M(v) = v$ otherwise.
The following observation gives a lower and upper bound for the treewidth of G given the treewidth of G/M:

Lemma 3.3. $\mathrm{tw}(G/M) \leq \mathrm{tw}(G) \leq 2\,\mathrm{tw}(G/M) + 1$

Proof. Every tree-decomposition $\mathcal{T} = (T, X)$ of G yields a tree-decomposition $\mathcal{T}' = (T, X' = f_M(X))$ of G/M which satisfies $\mathrm{width}(\mathcal{T}') \leq \mathrm{width}(\mathcal{T})$. Likewise, if $\mathcal{T}' = (T, X')$ is a tree-decomposition of G/M with width k, then $\mathcal{T} = (T, X = f_M^{-1}(X'))$ is a tree-decomposition of G, and for every $i \in I$,

$$|X_i| = |f_M^{-1}(X_i')| \leq \sum_{v' \in X_i'} |f^{-1}(v')| \leq \sum_{v' \in X_i'} 2 \leq 2(k+1).$$

Therefore, the width of $\mathcal{T}$ is at most $2(k+1) - 1 = 2k + 1$. □

As it turns out, having an upper bound for the treewidth of G as provided in Lemma 3.3 is enough to allow us to decide in linear time if the treewidth of G is at most k:

Theorem 3.4 ([Bodlaender and Kloks(1991)])**.** *Given $k \in \mathbb{N}$ and an upper bound $l \in \mathbb{N}$, there exist a linear time algorithm that, provided a graph G and a tree-decomposition $\mathcal{T}$ of G with width at most l, determines whether the treewidth of G is at most k and if so, finds a tree-decomposition of G with width at most k.*

The results of this chapter finally allow us to formulate the first sub-algorithm for our problem:

Algorithm 1. *Input:* $G = (V, E)$, $k \in \mathbb{N}$

- Find a maximal matching M and compute G/M.
- Compute the treewidth of the smaller graph G/M.
- If $\mathrm{tw}(G/M) > k$, **return** $\mathrm{tw}(G) > k$.
- If $\mathcal{T}'$ is a tree-decomposition of G/M with width at most k, compute a tree-decomposition $\widetilde{\mathcal{T}}$ of G with width at most $2k+1$ as in the proof of Lemma 3.3 and use the algorithm from Theorem 3.4 (with $l = 2k+1$) to **return** either $\mathrm{tw}(G) > k$ or a tree-decomposition $\mathcal{T}$ of G with width at most k.

The algorithm is correct if we assume correctness of the recursive call. A maximal matching can be computed in time $O(|V| + |E|)$ using a greedy algorithm. The contraction $G/_M$ and the tree-decompositions $\widetilde{\mathcal{T}}$ and $\mathcal{T}$ (by Theorem 3.4) can be computed in $O(|V|)$, so the overall running time of this sub-algorithm (excluding the recursive call) is $O(|V| + |E|)$.

3.2 Reduction by Deletion

The second approach to reduce our problem to a smaller instance is to delete vertices that do not affect the treewidth of G. First, we have to introduce some terminology.

Definition 3.5 (improved graph)**.** The *improved* graph of G is the graph $G_I = (V, E')$ obtained by adding an edge between every pair of vertices with at least $k + 1$ common neighbours, i.e. $E' = E \cup \{\{u, v\} |\ |N_G(u) \cap N_G(v)| \geq k + 1\}$.

In terms of treewidth, G and G_I are equivalent:

Lemma 3.6. $\mathrm{tw}(G) \leq k$ *if and only if* $\mathrm{tw}(G_I) \leq k$*, and every tree-decomposition* $\mathcal{T}$ *of* G *with width at most* k *is a tree-decomposition of* G_I *with width at most* k *and vice versa.*

Proof. Let $\mathcal{T} = (T, X)$ be a tree-decomposition of G with width at most k and let $u, v \in V$ have at least $k+1$ common neighbours. Then $\{u, v\}$ and $W := N_G(u) \cap N_G(v)$ induce a complete bipartite subgraph in G. By Corollary 1.8 there exists an $i \in I$ with $\{u, v\} \subseteq X_i$ or $W \subseteq X_i$. First *assume* $W \subseteq X_i$. Then $\mathcal{T}$ is still a tree-decomposition after adding an edge $\{w_1, w_2\}$ for every $w_1, w_2 \in W$ to G. But then $W \cup \{v\}$ forms a Clique of size at least $k + 2$, and therefore $\mathcal{T}$ has to contain a bag of size at least $k + 2$, contradicting $\mathrm{width}(\mathcal{T}) \leq k$. Therefore $\{u, v\} \subseteq X_i$, so $\mathcal{T}$ is still a tree-decomposition of G if we add an edge between u and v.
Furthermore, any tree-decomposition of G_I is also a tree-decomposition of G, since G is a subgraph of G_I. □

We will now define the vertices that will be deleted in the algorithm:

Definition 3.7 (simplicial, I-simplicial)**.** A *simplicial vertex* is a vertex v whose neighbours $N_G(v)$ form a Clique in G. An *I-simplicial* vertex is a vertex v that is simplicial in the improved graph G_I.

Note that if there exists an I-simplicial vertex v with $\deg(v) \geq k + 1$, then by Lemma 3.6 and Corollary 1.7 it follows that $\mathrm{tw}(G) \geq k + 1$.

Lemma 3.8. *Let* S *be a set of pairwise non-adjacent I-simplicial vertices with degree* $\leq k$ *in* G *and let* G' *be the graph obtained by deleting* S *from the improved graph* G_I*. Then* $\mathrm{tw}(G) = \mathrm{tw}(G')$*, and from a given tree-decomposition* $\mathcal{T}'$ *of* G'*, we can compute a tree-decomposition* $\mathcal{T}$ *of* G *with equal width.*

Proof. Let $\mathcal{T}' = (T', X')$ with $T' = (I', F')$ be a tree-decomposition of G' with width k. For every $s \in S$, $N_G(s)$ forms a clique in G', so by Corollary 1.7 there exists an $i_s \in I'$ such that $N_G(s) \subseteq X'_{i_s}$. Since $X_{j_s} := N_G(s) \cup \{s\}$ forms a clique of size at most $k+1$ in G, we can construct a tree-decomposition $\mathcal{T}_S = (T, X)$ of G with width k, where $T = (I' \cup \{j_s | s \in S\}, F' \cup \{\{i_s, j_s\} | s \in S\})$ and $X := X' \cup \{X_{j_s} | s \in S\}$.
Therefore, $\mathrm{tw}(G) \leq \mathrm{tw}(G')$. Since we also have $\mathrm{tw}(G') \leq \mathrm{tw}(G_I) = \mathrm{tw}(G)$, it follows that $\mathrm{tw}(G) = \mathrm{tw}(G')$. □

We obtain the second sub-algorithm for our problem:

Algorithm 2. *Input:* $G = (V, E)$, $k \in \mathbb{N}$

- Compute the improved graph G'.
- If G contains an I-simplicial vertex with degree at least $k + 1$, **return** $\mathrm{tw}(G) > k$.
- Find a maximal set S of pairwise non-adjacent I-simplicial vertices
- Compute the reduced graph G' by deleting all vertices in S from G
- Compute the treewidth of the smaller graph G'.
- If $\mathrm{tw}(G') > k$, **return** $\mathrm{tw}(G) > k$.

- If $\mathcal{T}'$ is a tree-decomposition of G' with width at most k, compute the tree-decomposition $\mathcal{T}_S$ of G as in the proof of Lemma 3.8 and **return** $\mathcal{T}_S$.

The algorithm is correct if we assume correctness of the recursive call. All steps can be computed in $O(|V|)$ (see [Bodlaender(1992)]), therefore the sub-algorithm has an overall linear running time.

3.3 Linear running time

Finally, we want to combine the algorithms 1 and 2 in a way as to achieve a sufficiently large reduction in every step. The main idea is to classify the vertices of G into low- and high degree vertices:

Definition 3.9 (low/high degree vertices, friendly vertices)**.** For a fixed constant $d \in \mathbb{N}$, a vertex $v \in V$ is called a *low degree vertex* if $\deg(v) \leq d$. Otherwise, v is called a *high degree vertex*. A low degree vertex that is adjacent to at least one other low degree vertex is called a *friendly vertex*.

It can be shown that for the right choice of d and a factor $\alpha \in (0, 1)$ (which both depend on k), if there are "enough" friendly vertices in G (i.e. at least $\alpha|V|$), then we can find a "large" matching to contract, and otherwise there are "many" I-simplicial vertice to delete.
Consider the first case: By definition, if we have a maximal matching M, then every friendly vertex needs to be either matched by M or adjacent to a friendly vertex that is matched by M. Since friendly vertices are of low degree, at most $2d$ friendly vertices can be related to a given matching edge in this way, effectively giving us a lower bound on the size of the maximal matching that is proportional to the number of friendly vertices. Therefore, every contraction reduces the size of the graph by factor of at least $\frac{\alpha}{2}$. A similar result can be shown for the second case (all these results can be found in [Bodlaender(1992)].)
This ensures a reduction in the size of G by a *constant* factor in every step. By using Lemma 3.2 we can further ensure that $|E| \in O(|V|)$, so both sub-algorithms 1 and 2 have a linear running time. Therefore, we obtain an overall linear running time of the final algorithm:

Algorithm 3. *Input:* $G = (V, E)$, $k \in \mathbb{N}$.

- If $|V|$ is small, we can use brute force. **Return** either $\mathrm{tw}(G) > k$ or a corresponding tree-composition with width at most k.
- If $|E| > k|V| - \frac{1}{2}k(k+1)$, **return** $\mathrm{tw}(G) > k$ (see Lemma 3.2).
- If there are at least $\alpha|V|$ friendly vertices, use Algorithm 1.
- Otherwise, use Algorithm 2.

This concludes the chapter on computing treewidth. As mentioned in the Introduction, by Courcelle's Theorem we know that problems on classes of graphs with bounded treewidth can be solved in polynomial time. If we have such a class of graphs with an upper bound k, we can now use Bodlaenders algorithm to efficiently find a tree-decomposition of every instance. For further reading on how we can use tree-decompositions to solve specific problems, see e.g. [Flum and Grohe(2006)].

References

[Bodlaender and Kloks(1991)] Hans L. Bodlaender and Ton Kloks. Better algorithms for the pathwidth and treewidth of graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 544–555. Springer, 1991. doi: 10.1007/3-540-54233-7_162.

[Seymour and Thomas(1993)] Paul D. Seymour and Robin Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993. doi: 10.1006/jctb.1993.1027.

[Bodlaender(1992)] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1992. doi: 10.1145/167088.167161.

[Flum and Grohe(2006)] J. Flum and M. Grohe. *Parameterized complexity theory.* Springer, 2006. doi: 10.1007/3-540-29953-X.