eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

Bachelor's Degree in Computer Engineering
Computing

Final Year Project

Treewidth

Theory and applications to Computer Science

Author:
Iván Matellanes Pastoriza

informatika fakultatea | facultad de informática

2015

Preface

This bachelor's thesis is the result of a study of the concept of treewidth. I was delighted by the proposal of carrying out some research in theoretical computer science. A few courses and extra activities had already given me the opportunity to taste some of the topics in this field, but the thesis was the real chance to get involved and immerse myself in it. Besides, I was willing to begin in research and do a project out of the ordinary.

The study has been accomplished inside the LoRea group from the Faculty of Computer Science at Donostia/San Sebastián, supervised by Prof Hubert Chen and with the collaboration of Prof Montse Hermo. The main goal of this project has been to produce a self-contained report about treewidth, with a focus on its applications to computer science and on the current line of research. It should be as clear and concise as possible for those that have never studied this concept before, and it contains practically everything needed to understand the definitions and theorems presented. The proofs in this report are self-written with an emphasis in making them easy to understand, although in some cases they are based in work by other authors.

The methodology consisted on reading articles from different publications, books and technical reports and giving presentations every two weeks about the studied topics. Feedback and comments were received in the talks and then the subjects were incorporated to this report.

Abstract

This report is an introduction to the concept of treewidth, a property of graphs that has important implications in algorithms. Some basic concepts of graph theory are presented in the first chapter for those readers that are not familiar with the notation. In Chapter 2, the definition of treewidth and some different ways of characterizing it are explained. The last two chapters focus on the algorithmic implications of treewidth, which are very relevant in Computer Science. An algorithm to compute the treewidth of a graph is presented and its result can be later applied to many other problems in graph theory, like those introduced in the last chapter.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Hubie, for encouraging me to carry out a project in theoretical computer science, and for giving me advice and guidance throughout the year.

I want to thank Montse and Asier too for bearing with my presentations and for their support and comments.

Finally, I cannot forget about Paqui and the rest of the members of the LoRea group for giving me the opportunity to work in their team.

Contents

List of Figures

Introduction

The *treewidth* is a numeric property of a graph that measures how close is the graph from being a tree. It was introduced for the first time in 1972 by Umberto Bertelé and Francesco Brioschi [3], and later rediscovered by Neil Robertson and Paul Seymour in 1984 [24]. Its important algorithmic implications have led to many authors developing a strong interest in the topic.

Treewidth can be defined in many different ways, as we shall see in Chapter 2, but the canonical characterization comes from the structure of tree decomposition. A *tree decomposition* is a representation of a graph in a tree-like structure, which gives rise to possibly the most important application of treewidth to computer science. Graphs that allow relatively small tree decompositions are said to have bounded treewidth, which effectively means that tree decompositions allow polynomial time algorithms for problems that are usually NP-hard, and therefore, difficult to compute for large inputs (unless $\mathsf{P} = \mathsf{NP}$).

Determining the treewidth of an arbitrary graph is an NP-hard problem itself [2]. However, if the treewidth of the graph is bounded by a small constant, we can find a small tree decomposition for that graph in linear time using the algorithm in Chapter 3. Furthermore, we know that for some particular families of graphs the treewidth can be computed in constant or polynomial time [4]. As an example, trees always have treewidth 1 and complete graphs on n vertices, treewidth $n-1$.

The applications of treewidth and tree decompositions to algorithms belong to a recent branch in computational complexity theory known as parameterized complexity. This field introduces a refined analysis of hard computational problems, classifying them with respect to some parameters of the input, not just the usual size of the input. In the case concerning us, the complexity of a problem is measured in terms of the size and the treewidth of the input graph. More precisely, when the complexity of an algorithm is exponential in the parameter but polynomial in the size of the input, the algorithm is called *fixed-parameter tractable*, as it allows efficient solutions for small values of the parameter even if the size of the input is big. Such problems belong to the class FPT. We present some examples of these algorithms in Chapter 4, but there is an endless number of applications of FPT problems out of the scope of this project, including parameters other than treewidth.

Chapter 1

Preliminaries

Graphs have been widely studied in the past decades for their ability to model many types of processes and relations in diverse fields. This chapter presents some basic concepts in graph theory.

As the report aims to be self-contained, we explain all terms and notation that are used later in this paper. Although not strictly necessary, some elementary knowledge in graph theory is advised. A couple of recommended readings to introduce in these topics are [13] and [28]. Those that are already familiar with the terminology can choose to skip some sections or even the whole chapter.

1.1 Graphs

An undirected graph is an ordered pair $G = (V, E)$ formed by a non-empty set of *vertices* or *nodes* V and a set of *edges* E, where an edge is a 2-element subset of V denoted by $\{a, b\}$ or simply ab. In a directed graph, edges are ordered pairs of elements of V, written as $(a, b) \neq (b, a)$. In both cases we will require the graphs to be loopless, this is, for every edge $ab \in E$, $a \neq b$.

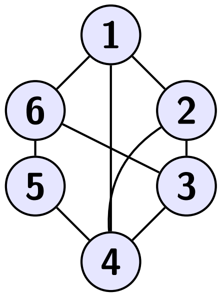
The two vertices that form an edge are its *ends* or *endpoints*. An edge $e = uv$ is *incident* to u and v. Two vertices u, v are *adjacent* or *neighbours* if uv is an edge in G. The neighbourhood of a vertex v, $N(v)$, is the set of all vertices $u \in V$ such that $uv \in E$. It is common to use $V(G)$ and $E(G)$ to refer to the vertex and edge sets of a graph G, respectively.

We usually represent a graph by drawing a dot or a circle for each vertex and a line joining two vertices if they form an edge.

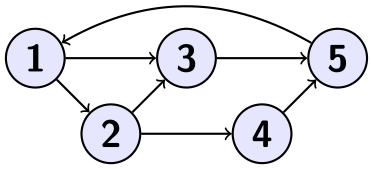
The *order* of a graph $G = (V, E)$ is its number of vertices, $|V|$ or $|G|$, and the *size* corresponds to its number of edges, $|E|$. Depending on its order, a graph can be *finite* or *infinite*. Unless stated otherwise, we can safely assume that the graphs in this report are finite and undirected.

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We define the following operations:

- Union: $G \cup G' = (V \cup V', E \cup E')$
- Intersection: $G \cap G' = (V \cap V', E \cap E')$
- Vertex deletion: If U is a set of vertices of G, $G \backslash U$ (or alternatively $G - U$) is the graph that results after deleting from G all the vertices in U and their incident edges. If U contains a single vertex u, we write $G - u$ instead.
- Edge addition or deletion: For a set of edges F on V, we write $G + F = (V, E \cup F)$ and $G - F = (V, E \setminus F)$. The notation $G + e$ and $G - e$ is also used as before for single edges.



$$V = \{1,2,3,4,5,6\}$$
$$E = \{\{1,2\}, \{1,4\}, \{1,6\}, \{2,3\}, \{2,4\}, \{3,4\}, \{3,6\}, \{4,5\}, \{5,6\}\}$$



$$V = \{1,2,3,4,5\}$$
$$E = \{(1,2), (1,3), (2,3), (2,4), (3,5), (4,5), (5,1)\}$$

Figure 1.1: Examples of undirected (top) and directed (bottom) graphs.

- Edge contraction: Given an edge $e = uv \in E$, the contraction of e produces the graph $G \bullet e = (V_e, E_e)$ where $V_e = (V \setminus \{u,v\}) \cup \{w\}$ (with $w \notin V$) and $E_e = \{xy \in E : \{x,y\} \cap \{u,v\} = \emptyset\} \cup \{xw : xu \in E \setminus \{e\} \text{ or } xv \in E \setminus \{e\}\}$. In other words, u and v are replaced with a new vertex w and edges incident to w are the edges other than e that were incident to u or v.

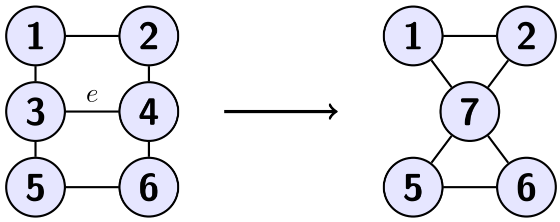


Figure 1.2: Contraction of the edge e.

If $V' \subseteq V$ and $E' \subseteq E$, we say that G' is a *subgraph* of G and write $G' \subseteq G$. In particular, when $G' \subseteq G$ and E' consists of all edges in E that are subsets of V', G' is the *induced subgraph* of G on V', and V' *induces* G' in G. This induced subgraph is denoted by $G' = G[V']$.

1.2 Paths and connectivity

A *path* is a special graph $P = (V, E)$ where $V = \{v_0, v_1, ..., v_n\}$ and $E = \{v_0v_1, v_1v_2, ..., v_{n-1}v_n\}$, with $n \geq 0$ and all v_i distinct. The vertices v_0 and v_n are its *ends*. The *length* of the path is its number of edges.

A path is often represented as the sequence of its vertices, $P = v_0v_1...v_n$. We equally say:

- P is a path between v_0 and v_n.
- P is a path from v_0 to v_n.

- P is a $v_0 - v_n$ path.
- v_0 and v_n are *connected* by the path P.

Let A, B be sets of vertices. P is an *$A - B$ path* if $v_0 \in A$, $v_n \in B$ and $v_1, ..., v_{n-1} \notin A \cup B$.

Given a path $P = v_0 v_1 ... v_n$ such that $n \geq 2$, the graph $P + v_n v_0$ is called a *cycle*. The *length* of a cycle is also its number of edges. As with the paths, cycles can be written as $C = v_0 v_1 ... v_n v_0$.

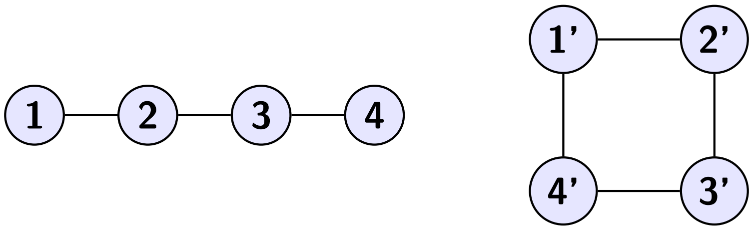


Figure 1.3: Examples of undirected path (left) and cycle (right).

A non-empty graph G is *connected* if there exists a path between any two vertices in the graph. Otherwise, the graph is *disconnected*. A *connected component*, or just *component*, of G is a connected subgraph whose vertices are not connected to any other vertex outside the component.

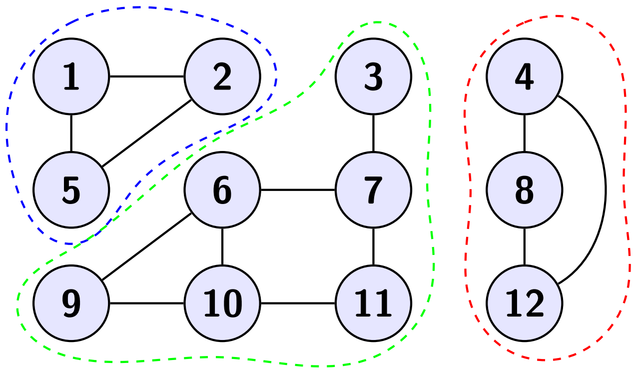


Figure 1.4: Components of a disconnected graph.

When every pair of distinct vertices in a graph G are connected by a single edge, we say that G is *complete*, and denote by K_n a complete graph of n vertices. A *clique* C is a subset of $V(G)$ such that for all pairs $u, v \in C$, $u \neq v$, then $uv \in E(G)$. In other words, $G[C]$ is complete. Sometimes we also call this induced subgraph a clique.

Given sets of vertices $A, B \subseteq V(G)$, a set $X \subseteq V(G)$ *separates* A and B in G if every $A - B$ path contains some vertex from X. Observe that $A \cap B \subseteq X$, and also that the deletion of X leaves what remains of A and B in distinct connected components.

The following theorem is one of the most important ones in graph theory, and it is widely known as Menger's theorem. See [26] for a proof.

Theorem 1.1 (Menger 1927)**.** *Let $G = (V, E)$ be a graph. The minimum number of vertices needed to separate $A \subseteq V$ and $B \subseteq V$ is equal to the maximum number of vertex-disjoint $A - B$ paths in G.*

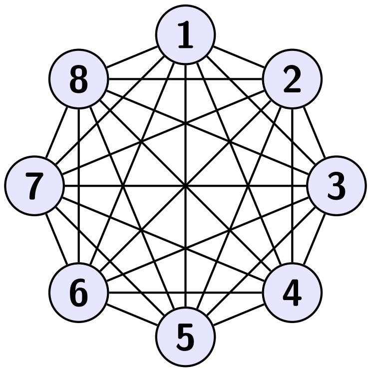


Figure 1.5: Complete graph of 8 vertices.

1.3 Trees

A *tree* is a connected graph that contains no cycles. A *leaf* is a single-neighbour node in a tree. Nodes that are not leaves are *internal nodes*. With a slight abuse of notation, we shall use $t \in T$ to refer to the nodes in a tree T in the same sense we would use $t \in V(T)$. The same applies for edges, writing $e \in T$ instead of $e \in E(T)$.

Theorem 1.2. *The following are equivalent definitions of a tree T:*

- *T is connected and acyclic.*
- *Each pair of nodes of T is connected by a unique path in T.*
- *T is minimally connected: for any edge $e \in T$, $T - e$ is disconnected.*
- *T is maximally acyclic: for any pair of non-adjacent nodes $u, v \in T$, $T + uv$ has a cycle.*

The proof of theorem 1.2 is simple, refer to [28] for further information.

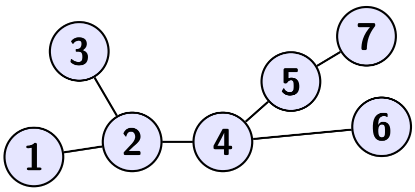


Figure 1.6: An example of tree.

One of the nodes in a tree T can be fixed and considered special for convenience. This node is the *root* of the tree, making T a *rooted tree*. Then we define the *height* of a node $t \in T$ as the length of the unique path that goes from the root to t.

Similar to the concept of subgraph, T' is a *subtree* of a tree T if $T' \subseteq T$ and T' is also a tree.

1.4 Flow networks

A *flow network* or simply *network* is a directed graph $G = (V, E)$ where each edge $e \in E$ has an associated non-negative real value $c(e)$ called its *capacity*. Two of the vertices are distinguished from the rest: the *source* and the *sink*, denoted by s and t respectively.

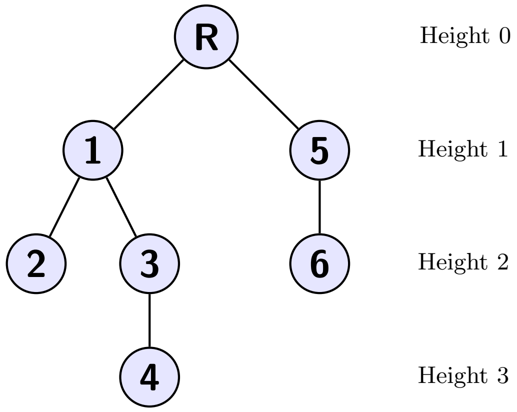


Figure 1.7: Usual representation of a rooted tree.

The *flow* in a network is a function $f : E \to \mathbb{R}$ subject to these constraints:

- The flow along an edge cannot exceed its capacity:
  For each $e \in E$, $f(e) \leq c(e)$.
- Incoming flow is equal to outgoing flow, except in s and t:
  For each $v \in V \setminus \{s, t\}$, $\sum\limits_{(u,v) \in E} f((u, v)) = \sum\limits_{(v,x) \in E} f((v, x))$.

The source only produces flow and the sink only consumes flow. The value of the flow is given by $\sum_{(s,v) \in E} f((s, v))$, and it represents the amount of flow from s to t.

The problem that is usually presented on a flow network involves finding a flow from s to t of maximum value, this is, routing as much flow as possible. We refer to this problem as the *maximum flow* problem.

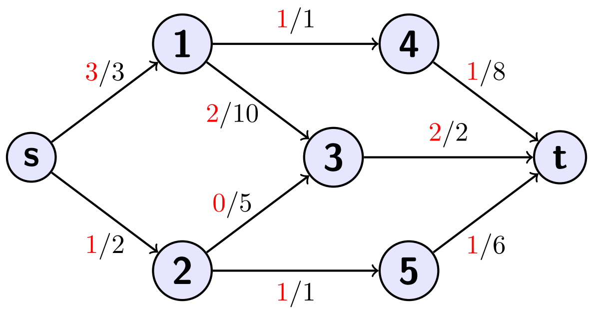


Figure 1.8: Flow network showing capacities (black) and a maximum flow (red).

Networks arise in many contexts and can be used to model countless scenarios: pipes, road systems, computer networks, electrical current distribution... Well known problems such as bipartite matching and the assignment problem can be solved using flow networks. These and other applications, along with efficient algorithms, are discussed in [1].

Chapter 2

Characterizations

The concept of treewidth can be defined using different approaches. The traditional definition comes from the structure of tree decomposition, but many other have been proposed. Elimination orderings, brambles and game theoretical models are the ones presented here along with tree decompositions. As the main goal of this document is not to study the concept itself but its applications to computer science, we do not put the focus on equivalent definitions. Other ways to characterize the treewidth of a graph can be found in the literature and are equally valid (cf. [6]).

2.1 Tree Decompositions

Definition 2.1. A *tree decomposition* of a graph $G = (V, E)$ consists of a tree T and, for each node $t \in T$, an associated *bag* $V_t \subseteq V$ such that:

(i) $V = \bigcup_{t \in T} V_t$

(ii) For each edge $uv \in E$, there exists $t \in T$ such that $u, v \in V_t$

(iii) For each $v \in V$, the set $S_v = \{t \in T \mid v \in V_t\}$ induces a non-empty subtree of T.

The width of a tree decomposition is the size of its largest bag minus one:

$$\max\{|V_t| - 1 : t \in T\}$$

The treewidth of G, $tw(G)$, is defined as the minimum width among all possible tree decompositions of G.

Definition 2.2. A tree decomposition $(T, (V_t)_{t \in T})$ is *nonredundant* if there is no edge $xy \in T$ such that $V_x \subseteq V_y$.

Any redundant tree decomposition can be transformed into a nonredundant one by simply contracting every edge $xy \in T$ such that $V_x \subseteq V_y$ and joining the two bags together. Moreover, we can notice that the number of bags in a nonredundant tree decomposition is bounded by the order of the graph.

Lemma 2.3. *Let G be a graph of n vertices. Any nonredundant tree decomposition of G has at most n bags.*

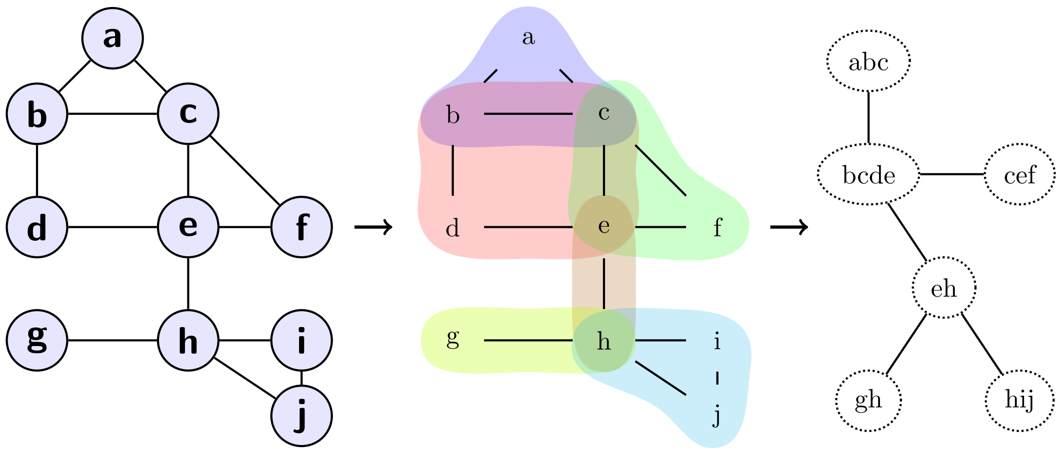


Figure 2.1: A graph and a possible tree decomposition of width 3.

Proof. This can be proved by induction on n. The case $n = 1$ is clear.

If $n > 1$, let $(T, (V_t)_{t\in T})$ be a nonredundant tree decomposition of G and let t be a leaf in T. Since it is nonredundant, one or more vertices in V_t are not in its neighbouring bag, and hence by (iii) in definition 2.1 they are not in any other bag. Let U be those vertices. By deleting t and V_t we obtain a nonredundant tree decomposition of $G \setminus U$. By the induction hypothesis, this tree decomposition has at most $n - |U| \leq n - 1$ bags, so $(T, (V_t)_{t\in T})$ has at most n bags, which completes the proof. □

Notice that tree decompositions are passed on to subgraphs, as follows from this lemma:

Lemma 2.4. *For every $H \subseteq G$, the pair $(T, (W_t)_{t\in T})$ where $W_t = V_t \cap V(H)$ is a tree decomposition of H.*

Proof. We can easily see that the three points in definition 2.1 still hold for $(T, (W_t)_{t\in T})$ and H, as T has not been modified and possibly having some empty bags is not a problem. □

Corollary 2.5. *For every $H \subseteq G$, $tw(H) \leq tw(G)$ as a consequence of lemma 2.4.*

Conversely, a subtree of a tree decomposition is a valid tree decomposition of the graph induced by the vertices contained in its bags, as follows from applying the next lemma recursively.

Lemma 2.6. *Let $(T, (V_t)_{t\in T})$ be a tree decomposition of width k of a graph $G = (V, E)$, and let l be a leaf in T. Let l' be the unique neighbour of l and set $U = V_l \setminus V_{l'}$. Then $(T - l, (V_t)_{t\in(T-l)})$ is a tree decomposition of width $\leq k$ of $G \setminus U$.*

Proof. First of all, for each edge $uv \in E(G \setminus U)$, there must be some $x \in T$ such that $u, v \in V_x$. Notice that $u, v \notin U$ or otherwise the edge would not be part of $G \setminus U$. If $x \neq l$ then $x \in T'$; if $x = l$, then $u, v \in V_{l'}$ as well since we know $u, v \notin U$, so $l' \in T'$ and (ii) in definition 2.1 is satisfied.

Secondly, for each vertex $v \in V(G \setminus U)$ the set S_v in $T - l$ induces a non-empty subtree of $T - l$ because $v \notin U$ and the deleted node was a leaf (observe that deleting a leaf from a tree gives a connected subtree), so (iii) in definition 2.1 is also satisfied.

Finally, the width of a tree decomposition clearly cannot increase when deleting a node. □

2.1.1 Connectivity and separation

Some interesting properties of the tree decomposition deal with its connectivity and separation attributes. For the following lemmas, let $(T, (V_t)_{t \in T})$ be a given tree decomposition of a graph $G = (V, E)$. We will see that the separation of the nodes in T is somehow related to the separation of the vertices in G.

Lemma 2.7. *Let $t \in T$ and $v \in V$ such that $v \notin V_t$. Then v is contained in bags from only one of the components of $T - t$.*

Proof. Denote by T_1, T_2 any two components of $T - t$. Suppose there are $t_1 \in T_1$ and $t_2 \in T_2$ such that $v \in V_{t_1}$ and $v \in V_{t_2}$. Observe that t is in the path from t_1 to t_2. By (iii) in definition 2.1 $v \in V_t$, a contradiction. □

Lemma 2.8. *Let $t \in T$ and let $T_1, T_2, ...T_r$ be the components of $T - t$. Set $U_i = \bigcup_{s \in T_i} V_s$ for each T_i, $1 \leq i \leq r$. Then the subgraphs $G[U_1 \setminus V_t], G[U_2 \setminus V_t], ..., G[U_r \setminus V_t]$ have neither vertices in common nor edges between them.*

Proof. It is clear from lemma 2.7 that a vertex not belonging to V_t cannot be in bags from two different components of $T - t$.

Similarly, if an edge $uv \in E$ has $u \in U_i \setminus V_t$ and $v \in U_j \setminus V_t$ for some $i \neq j$, then $u \in V_x$ and $v \in V_y$ for some $x \in T_i$ and $y \in T_j$. Additionally, some bag V_z must contain both u and v by (ii) in definition 2.1, and $z \neq t$ by the choice of u and v. Supposing without loss of generality that $z \in T_k$ for some $k \neq i$, V_x and V_z contain u and lie in different components of $T - t$, contradicting lemma 2.7. □

Lemma 2.9. *Let t_1, t_2, t_3 be nodes of T such that t_2 is in the path from t_1 to t_3. Then V_{t_2} separates $V_{t_1} \setminus V_{t_2}$ and $V_{t_3} \setminus V_{t_2}$ in G.*

Proof. Observe that t_1 and t_3 lie in different components of $T - t_2$. By lemma 2.8, there is no path between $V_{t_1} \setminus V_{t_2}$ and $V_{t_3} \setminus V_{t_2}$ in $G \setminus V_{t_2}$, which proves the lemma. □

Lemma 2.10. *Let $e = xy$ be an edge in T, and let T_1, T_2 be the two components of $T - e$. Set U_1 and U_2 as in lemma 2.8. Then the set $V_x \cap V_y$ separates U_1 from U_2 in G.*

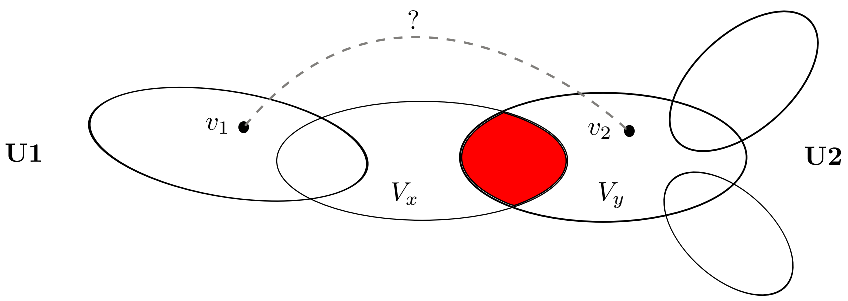


Figure 2.2: $V_x \cap V_y$ separates U_1 from U_2 in G.

Proof. In the first place, for any $u \in U_1 \cap U_2$, it follows from (iii) in definition 2.1 that u is also in V_x and in V_y, so $u \in V_x \cap V_y$.

Secondly, in the case where $u_1 \in U_1 \setminus U_2$ and $u_2 \in U_2 \setminus U_1$, we have to show that all $u_1 - u_2$ paths in G have some vertex in $V_x \cap V_y$. For this purpose, it is enough to see that there is no edge $v_1v_2 \in E$ with $v_1 \in U_1 \setminus (V_x \cap V_y)$ and $v_2 \in U_2 \setminus (V_x \cap V_y)$. If there was, some V_z would contain v_1 and v_2 by (ii) in definition 2.1. Assume w.l.o.g. that $z \in T_1$, and by the choice of v_2 we know that it belongs to some bag V_w where $w \in T_2$. Then z and w are linked by e, and by (iii) in definition 2.1 $v_2 \in V_x \cap V_y$, a contradiction. □

Another useful property of tree decompositions deals with complete subgraphs:

Lemma 2.11. *For any clique $W \subseteq V$, there is a node $t \in T$ such that $W \subseteq V_t$.*

Proof. Let t be any node in T. If $W \subseteq V_t$ we are done, otherwise there is some $w \in W$ such that $w \notin V_t$. This w is in only one of the components of $T - t$, say T_1, by lemma 2.7. Since w has an edge with every other vertex in W, the whole W has to be in bags from T_1 by (ii) in definition 2.1.

Now consider only the nodes and bags from T_1 and repeat the process. At some point we will get to a node $t \in T$ such that $W \subseteq V_t$ as the number of nodes we are considering decreases at each step. □

This implies the following fact of complete graphs, since any tree decomposition has a bag with all the vertices:

Corollary 2.12. *The treewidth of a complete graph of n vertices is $n - 1$.*

2.2 Elimination Orderings

Definition 2.13. Let $G = (V, E)$ be a graph. An elimination ordering is a pair $(\mathcal{V}, E')$ such that:

- $\mathcal{V} = (v_1, v_2, ..., v_n)$ is an ordering of the vertices in V.
- $E \subseteq E'$.
- For $i < j < k$, if $v_iv_k \in E'$ and $v_jv_k \in E'$, then $v_iv_j \in E'$.

The lower neighbours of a vertex v_j in the elimination ordering are the vertices v_i such that $i < j$ and $v_iv_j \in E'$. The width of an elimination ordering is the maximum number of lower neighbours among all vertices in the graph.

Elimination orderings are just another way of defining treewidth, as the following theorem proves.

Theorem 2.14. *Let G be a graph and let $k > 1$ be an integer. There exists a tree decomposition of G of width $< k$ if and only if there exists an elimination ordering of G with width $< k$.*

Proof. Given a subset $X \subseteq V$, $K(X)$ refers to the set of all possible edges between the vertices in X.

We start proving a stronger statement for the forward implication: if there exists a tree decomposition of G of width $< k$ then there exists an elimination ordering $(\mathcal{V}, E')$ of G of width $< k$ where E' contains $K(V_t)$ for every bag V_t of the tree decomposition. For this purpose, assume that we have a tree decomposition $(T, (V_t)_{t\in T})$ of G where each bag has size $\leq k$ and $|T| = n$. We apply induction on n.

For the base case, $n = 1$, let t be the single node in T. We can construct an elimination ordering of G formed by any ordering of the $\leq k$ vertices and the set of all possible edges in V_t, $K(V_t)$. The width of this elimination ordering is clearly $< k$ since no vertex can have $\geq k$ neighbours.

Now suppose that our tree decomposition of G has two or more bags. Let t be a leaf of T, and t' its unique neighbour. Set $U = V_t \setminus V_t'$. Then the deletion of t and its bag V_t yields a tree decomposition of $G \setminus U$ by lemma 2.6. By induction we have an elimination ordering of $G \setminus U$ of width $< k$, say $(\mathcal{W}, F)$, such that $K(V_s) \subseteq F$ for all $s \neq t$.

Denote by $u_1, ..., u_m$ the elements in U. The pair $(\mathcal{V}, F \cup K(V_t))$, where $\mathcal{V} = (\mathcal{W}, u_1, ..., u_m)$, is an elimination ordering of G: for any $uv \in E(G)$, if $uv \in E(G \setminus U)$ then $uv \in F$, otherwise $u \in U$ or $v \in U$ and by (ii) in definition 2.1 $u, v \in V_t$, hence $uv \in K(V_t)$.

The width of this new ordering is still $< k$ because any new u_i has $< k$ lower neighbours: at most the $k - 1$ other vertices in V_t (recall that $|V_t| \leq k$). In addition, the lower neighbours of the remaining vertices have not changed: for any two $v_1, v_2 \in V_t \setminus U$, observe that $v_1, v_2 \in V_{t'}$ and hence $v_1 v_2 \in K(V_{t'}) \subseteq F$ by the induction hypothesis.

For the backward implication, given an elimination ordering $(\mathcal{V}, E')$ of G of width $< k$, we can construct a tree decomposition of the graph $(V(G), E')$ where each bag has size $\leq k$. This tree decomposition will be valid also for the graph G as $E(G) \subseteq E'$. This time we use induction on the order of G, $|V(G)| = n$. The base case, $n = 1$, is trivial.

Now let $\mathcal{V} = (v_1, v_2, ..., v_n)$. Consider $(\mathcal{W} = (v_1, v_2, ..., v_{n-1}), F)$ where $F = E' \setminus \{v_i v_n \mid 1 \leq i < n\}$. We claim that $(\mathcal{W}, F)$ is an elimination ordering of $G' = G - v_n$ of width $< k$. By induction, there exists a tree decomposition $(T', (W_t)_{t \in T'})$ of $(V(G'), F)$ where each bag has size $\leq k$.

Define $(T, (V_t)_{t \in T})$ to be a tree decomposition of $(V(G), E')$ in the following way:

- $V(T) = V(T') \cup \{u\}$ where $u \notin T'$.
- $V_u = N(v_n) \cup \{v_n\}$, so the size of V_u is $\leq k$ since v_n has $< k$ neighbours.
- For the rest $t \in T'$, $t \neq u$, $V_t = W_t$.
- By lemma 2.11, there is some $t \in T'$ such that $N(v_n) \subseteq V_t$ because $N(v_n)$ is a clique in $(V(G), E')$. Set $E(T) = E(T') + tu$.

□

2.3 Brambles

Definition 2.15. Two subsets $X, Y \subseteq V$ *touch* if either $X \cap Y \neq \emptyset$ or some vertex in X has a neighbour in Y.

Definition 2.16. A *bramble* $\mathcal{B}$ for a graph G is a set of connected subsets of $V(G)$ that touch each other.

The order of a bramble is the size of the smallest $X \subseteq V(G)$ that covers all the subsets in the bramble, i.e., for all $B \in \mathcal{B}$, $X \cap B \neq \emptyset$.

In [25] brambles are called *screens* and their order, *thickness*.

We shall see that treewidth is closely related to brambles and the games presented in the next section. In this section in particular, we want to prove that if a graph has treewidth $\geq k$, then it has a bramble of order $> k$. The following lemmas will help us in this task.

Lemma 2.17. *Any set of vertices separating two covers of a bramble also covers that bramble.*

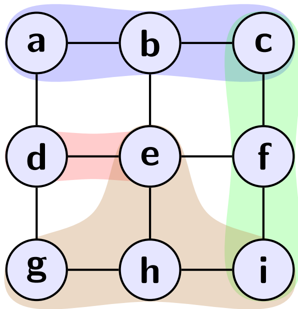


Figure 2.3: Example of a bramble on the 3x3 grid.

Proof. Let $\mathcal{B}$ be a bramble and C_1, C_2 two covers of $\mathcal{B}$ separated by the set S.

For every $B \in \mathcal{B}$, there are some $u \in C_1$ and $v \in C_2$ such that u and v meet B, this is, $u \in B$ and $v \in B$. Since B is connected, there exists some $u - v$ path that is inside B. This path must contain some vertex $w \in S$ because S separates the covers, so $w \in B$ as well and S covers the bramble. $\square$

Definition 2.18. Fix $k \geq 1$. Given a bramble $\mathcal{B}$ for a graph G, a tree decomposition of G is $\mathcal{B}$*-admissible* if every bag of size $> k$ is a leaf and fails to cover $\mathcal{B}$.

Lemma 2.19. *Let G be a graph with no bramble of order $> k$. For every bramble $\mathcal{B}$ there exists a $\mathcal{B}$-admissible tree decomposition T.*

Proof. We may assume that $|V(G)| > k$, otherwise the tree decomposition with a single bag $X = V(G)$ satisfies the lemma. Assume this induction hypothesis: for every bramble $\mathcal{B}'$ containing more sets than $\mathcal{B}$, there is a $\mathcal{B}'$-admissible tree decomposition of G. This holds for the base case where $\mathcal{B}$ has the maximum possible amount of sets, no greater than $2^{|V(G)|}$.

Let $\mathcal{B}$ be a bramble of order $\leq k$ for G, and let $W \subseteq V(G)$ be a cover of $\mathcal{B}$ of minimum size $|W| \leq k$. Denote the components of $G \setminus W$ by $C_1, C_2, ..., C_r$. Since $|V(G)| > k$, there exists at least one of these components. We shall prove this statement: for every component C_i there exists a $\mathcal{B}$-admissible tree decomposition of $G[W \cup C_i]$, T_i, where W is one of its bags. Joining the tree decompositions $T_1, T_2, ..., T_r$ from the bag W gives the $\mathcal{B}$-admissible tree decomposition we want (see fig. 2.4).

To prove the previous statement, let $H = G[W \cup C_i]$ and $\mathcal{B}' = \mathcal{B} \cup \{C_i\}$. If C_i does not touch some element in $\mathcal{B}$, then neither C_i nor its neighbours intersect that element, and hence $C_i \cup N(C_i)$ fails to cover $\mathcal{B}$. Thus, T_i is the tree decomposition consisting of two bags: W and $C_i \cup N(C_i)$.

Otherwise, $\mathcal{B}'$ is a bramble. In fact, $C_i \notin \mathcal{B}$ since W covers $\mathcal{B}$ and $W \cap C_i = \emptyset$. Therefore, we have that $|\mathcal{B}'| > |\mathcal{B}|$ and by induction there is a $\mathcal{B}'$-admissible tree decomposition of G, say T'.

If T' is also $\mathcal{B}$-admissible this is the tree decomposition that satisfies the lemma. If not, T' contains a leaf node x whose bag V_x has size $> k$ and covers $\mathcal{B}$ but not $\mathcal{B}'$, so V_x is disjoint with C_i and therefore it lies on $G \setminus C_i$. By lemma 2.17 any set separating W and V_x (both cover $\mathcal{B}$) also covers $\mathcal{B}$, so no such set can be of size $< |W|$ because we selected W to be a cover of minimum size for $\mathcal{B}$. By theorem 1.1 there exist $|W|$ vertex-disjoint paths $P_1, P_2, ..., P_{|W|}$ which connect W and V_x (fig. 2.5). Observe that the paths P_i meet W, and hence H, only in their ends.

We transform T' into T_i appropriately:

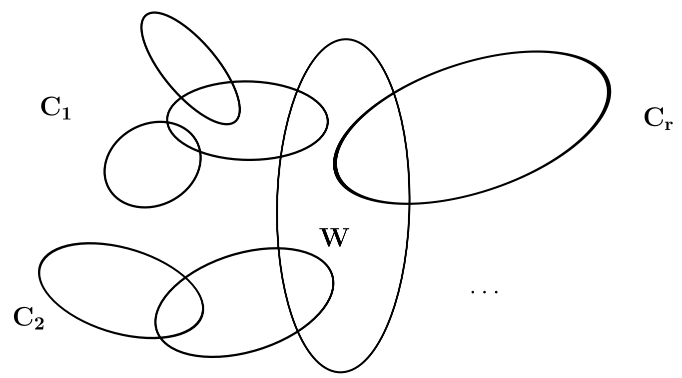


Figure 2.4: Joining T_i for every component C_i of $G \setminus W$ gives T.

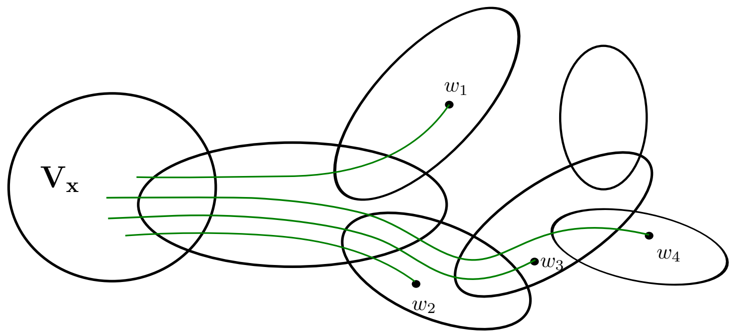


Figure 2.5: $|W|$ vertex-disjoint paths $P_1, P_2, ..., P_{|W|}$ which connect W and V_x.

1. All nodes that are not in H are deleted from all bags.
2. For every $w \in W$, pick a node $y \in T'$ whose bag contains w. Insert w in every bag V_t such that t is on the path from x to y.

Note that the size of any bag does not increase. At least a node from a path P_i is deleted for every $w \in W$ that gets inserted in a bag V_t. The reason is that V_t separates $V_y \setminus V_t$ and $V_x \setminus V_t$ in G by lemma 2.9, thus every path in G from w to a vertex in V_x contains some vertex in V_t. Such a path is P_i, and we know that all its vertices except w are not in H, so the one in V_t gets deleted.

Moreover, T_i is still a tree decomposition after 1 by lemma 2.4, and since each w is inserted in every bag along some path leading to a bag with w, it satisfies (iii) from definition 2.1 and is a tree decomposition after 2. Observe that insertions only happen in internal nodes of T' and in V_x. The new content of the bag V_x is exactly W, so we only have to show that T_i is $\mathcal{B}$-admissible.

Any bag V_z from T_i of size $> k$ is a leaf and contains at least one vertex from C_i because $V_z \subseteq (W \cup C_i)$ and $|W| \leq k$. That vertex was already there before the transformation because no such vertex has been inserted, and V_z failed covering $\mathcal{B}'$, so it had to fail covering some $B \in \mathcal{B}$. Thus, it still does not cover $\mathcal{B}$ since no vertex has been inserted into the leaves (other than V_x). This shows that T_i is the tree decomposition we were looking for and completes the proof. $\square$

Now we are ready to provide a proof for the theorem we were interested in.

Theorem 2.20 (Seymour and Thomas [25]). *Let $k \geq 1$ be an integer. If a graph G has treewidth $\geq k$ then G has a bramble of order $> k$.*

Proof. Assume that G contains no bramble of order $> k$. By lemma 2.19, we can find a $\mathcal{B}$-admissible tree decomposition of G for every $\mathcal{B}$. For $\mathcal{B} = \{V(G)\}$ this implies that the tree decomposition has no bag of size $> k$ since any bag would cover that $\mathcal{B}$. Hence, $tw(G) < k$. □

2.4 Pursuit-evasion games

We present a cops-and-robber game, played on a finite and undirected graph $G = (V, E)$ by $k \geq 1$ cops and a robber that must elude them.

Cops can move anywhere in the graph by helicopter, but they require two turns to perform the movement (a turn to get in the helicopter and take off, and a second turn to land). Therefore, at any time, a cop either stands on a vertex of the graph or is moving on a helicopter. The robber can move arbitrarily fast from the vertex he is to any other reachable vertex, i.e., any vertex along a path of the graph that is not blocked by a cop. Assuming that the cops always know where the robber is, the objective of the game is to corner him somewhere and land a helicopter in the vertex he is. The robber wins if he can find a strategy to avoid the cops *ad infinitum*.

Definition 2.21. We call an *X-flap* of G the vertex set of a component of $G \setminus X$.

Let us denote by $[V]^{\leq k}$ the set of all subsets of V of cardinality $\leq k$. Then the game can be formally defined as follows.

Definition 2.22. A state of the game is a pair (X, R) where $X \in [V]^{\leq k}$ and R is an X-flap. X is the set of vertices occupied by cops, and R represents the position of the robber.

The initial state is (X_0, R_0) with $X_0 = \emptyset$ and R_0 being any component of G. At any step of the game, the current state is (X_{i-1}, R_{i-1}). The cop player chooses $X_i \in [V]^{\leq k}$ such that $X_{i-1} \subseteq X_i$ or $X_i \subseteq X_{i-1}$. Then, the robber chooses an X_i-flap R_i such that $R_i \subseteq R_{i-1}$ or $R_{i-1} \subseteq R_i$, respectively. If no such choice is available for the robber, then cops have won, otherwise the game continues with another step.

Furthermore, if the sequence $X_0, X_1, ...$ satisfies $X_i \cap X_k \subseteq X_j$ for $i \leq j \leq k$, then "$\leq k$ cops can *monotonically* search G". Intuitively, this means that once cops leave a vertex it is not visited again during the game.

A similar game is called *jump-searching*, where the state is represented as in the search game but follows a slightly different rule. Set (X_0, R_0) as before. From the state (X_{i-1}, R_{i-1}), the cop player chooses a new X_i but this time with no restriction. Then the robber chooses an X_i-flap R_i that touches R_{i-1}.

Lemma 2.23. *If $\leq k$ cops cannot jump-search G, then $\leq k$ cops cannot search G.*

Proof. Let (X_{i-1}, R_{i-1}) be the current state of the search game. The cops move to position X_i following the rules of the search game. If $X_i \subseteq X_{i-1}$ there are fewer cops in the graph after the movement, so the X_{i-1}-flaps either remain the same or get bigger, $R_{i-1} \subseteq R_i$. Otherwise $X_{i-1} \subseteq X_i$, there are more cops in the graph now so the X_{i-1}-flaps are unchanged or become smaller, $R_i \subseteq R_{i-1}$.

Assuming that $\leq k$ cops cannot jump-search G, the robber has an X_i-flap R_i available that touches R_{i-1}. In fact, this R_i must follow one of the two cases presented above, giving a valid movement for the robber. □

The strategy for the robber in the jump-search game is given by a type of function called *haven*.

Definition 2.24. A *haven* in G of order k is a function β that assigns an X-flap $\beta(X)$ to each $X \in [V]^{<k}$ such that $\beta(X)$ touches $\beta(Y)$ for all $X, Y \in [V]^{<k}$.

Lemma 2.25. *G cannot be jump-searched by $\leq k$ cops if and only if there exists a haven in G of order $> k$.*

Proof. Suppose that $\leq k$ cops cannot jump-search G. Then for each $X \in [V]^{<k}$, let $\sigma(X)$ be an X-flap R such that from the state (X, R) the cop cannot guarantee to win. Then σ is a haven in G of order $> k$.

On the other hand, let β be a haven in G of order $> k$. At any step i the cops make their move to the position X_i, then the robber can choose $R_i \in \beta(X_i)$ to avoid them. □

Now we are ready to see the relationship between these games, the brambles and the concept of treewidth.

Theorem 2.26 (Seymour and Thomas [25])**.** *The next are equivalent:*

(1) G has a bramble of order $> k$.

(2) G has a haven of order $> k$.

(3) $\leq k$ cops cannot jump-search G.

(4) $\leq k$ cops cannot search G.

(5) $\leq k$ cops cannot monotonically search G.

(6) G has treewidth $\geq k$.

Proof. (1) $\rightarrow$ (2) is proven in the next lemma.
(2) $\rightarrow$ (3) follows from lemma 2.25.
(3) $\rightarrow$ (4) is proved by lemma 2.23.
(4) $\rightarrow$ (5) is an immediate consequence of the definition of monotonic search. If the graph cannot be searched at all, a more restricted search is also impossible.
(5) $\rightarrow$ (6) will come from lemma 2.28.
Finally, (6) $\rightarrow$ (1) was shown in theorem 2.20. □

Lemma 2.27. *If G has a bramble of order $> k$ then G has a haven of order $> k$.*

Proof. Let $\mathcal{B}$ be a bramble for G of order $> k$. For each $X \in [V]^{\leq k}$ there exists some connected $B \in \mathcal{B}$ with $X \cap B = \emptyset$, so let $\beta(X)$ be the X-flap containing B. Since B touches every other subset in $\mathcal{B}$, so does $\beta(X)$. Therefore, all $\beta(X)$ where $X \in [V]^{\leq k}$ touch each other and β is a haven in G of order $> k$. □

Lemma 2.28. *If $\leq k$ cops cannot monotonically search G, then G has treewidth $\geq k$.*

Proof. Assume for contrapositive that $tw(G) < k$, and let $(T, (V_t)_{t \in T})$ be a tree decomposition of G where all bags have size $\leq k$. Place the cops in the vertices from any bag $X = V_t$; that will require at most k cops. As follows from lemma 2.8, the robber stands on an X-flap in one of the components of $T - t$. Let t' be the neighbour of t in that component. Then the set $V_t \cap V_{t'}$ separates U_1 and U_2 by lemma 2.10, so we can safely move the cops in two turns to the vertices in $V_{t'}$ without the robber being able to escape, because the cops in $V_t \cap V_{t'}$ block his way out of the component. Repeat these steps until the robber is cornered.

Since at any step the vertices occupied by cops correspond to a bag from the tree decomposition, $\leq k$ cops will suffice. Moreover, by (iii) in definition 2.1 the search is monotonic: once the cops leave a vertex v it is not visited again because the bags that contain v induce a connected subgraph of T. $\square$

Chapter 3

Constructing a tree decomposition

Now that we know what treewidth is, it is reasonable to ask how could we actually get a low-width tree decomposition of a given graph. Having in mind that determining the treewidth of a graph is NP-hard, this task does not seem easy. However, the problem becomes tractable for graphs of small treewidth where we fix an upper bound.

For this reason, we will see an algorithm presented in [21] that given a graph and a fixed parameter, constructs a tree decomposition in reasonable time provided that the treewidth of the graph is smaller than the parameter. Otherwise, the parameter would turn out to be a lower bound for the treewidth.

Before presenting the actual algorithm, we need to find a way to detect if the treewidth of a graph is possibly large. The w-linked sets introduced in the first section will serve the purpose.

3.1 Strongly interlaced sets

The following structure can be used to identify whether the treewidth of a graph G is large.

Definition 3.1. Two sets $X, Y \subseteq V(G)$, $|X| = |Y|$, are *separable* if some strictly smaller set S separates them, this is, X and Y are disconnected in $G \setminus S$.

Definition 3.2. A set $X \subseteq V(G)$ is *w-linked* if $|X| \geq w$ and X does not contain separable subsets Y and Z such that $|Y| = |Z| \leq w$.

For the reason that a tree decomposition splits the graph in parts of possibly small size that separate it (as we have seen in the previous chapter), we can think of a w-linked set as an obstacle to construct a low-width tree decomposition, since such a set is hard to separate. Our intuition is confirmed by this theorem, based on the work by Kleinberg and Tardos [21]:

Theorem 3.3. *If a graph G contains a $(w+1)$-linked set of size $\geq 3w$, then $tw(G) \geq w$.*

Proof. Suppose for a contradiction that G has a $(w+1)$-linked set X of size $\geq 3w$, and that $(T, (V_t)_{t \in T})$ is a tree decomposition of G of width $< w$. The size of each bag V_t is $\leq w$. Assume also that this tree decomposition is nonredundant.

Our goal is to find a bag V_t such that when some $S \subseteq V_t$ is deleted from G, two small subsets of X are separated from each other. Since $|V_t| \leq w$, this will contradict the assumption that X is $(w+1)$-linked.

To begin with, root the tree T at a node r. Let T_t denote the subtree rooted at a node t, and G_t the graph induced by the union of bags from T_t. Now set t to be a node as far from the root as possible such that G_t contains $> 2w$ nodes of X. Such a node exists because G_r itself contains all nodes of X. Observe that t cannot be a leaf, because $|G_t| \leq w$ in that case, so let $t_1, t_2, ..., t_d$ be its children. Each G_{t_i} contains at most $2w$ nodes of X, by our choice of t being as far from the root as possible. We now consider two possible scenarios.

If there is a child t_i such that G_{t_i} contains at least w nodes of X, then we define Y to be w nodes of X from G_{t_i}, and Z to be w nodes of X from $G \setminus G_{t_i}$. Since the tree decomposition is nonredundant, $V_t \neq V_{t_i}$ and hence $S = V_t \cap V_{t_i}$ has at most $w - 1$ nodes. By lemma 2.10 S separates Y and Z, contradicting our assumption.

In the case where there is no child t_i so that G_{t_i} contains at least w nodes of X, we will combine several G_{t_i} to get to a similar situation. Beginning with G_{t_1}, combine it with G_{t_2}, then G_{t_3} and so on, until we first get a subgraph containing $> w$ nodes of X. This will happen after adding some G_{t_i} because G_t contains $> 2w$ nodes of X and at most w of them can be in V_t. Let W be the set of nodes in the subgraphs $G_{t_1}, G_{t_2}, ..., G_{t_i}$. We have that $w < |W \cap X| < 2w$ by the choice of W: more than w or we would have continued combining $G_{t_{i+1}}$, and fewer than $2w$ because combining $G_{t_1}, G_{t_2}, ..., G_{t_{i-1}}$ we had $\leq w$ nodes of X and G_{t_i} contains $< w$ in this case we are studying (fig. 3.1). This time we define Y to be $w + 1$ nodes of X from W, and Z to be $w + 1$ nodes of X not in W. By lemma 2.8, V_t is a set of size $\leq w$ that separates Y from Z, contradicting again that X is $(w + 1)$-linked. □

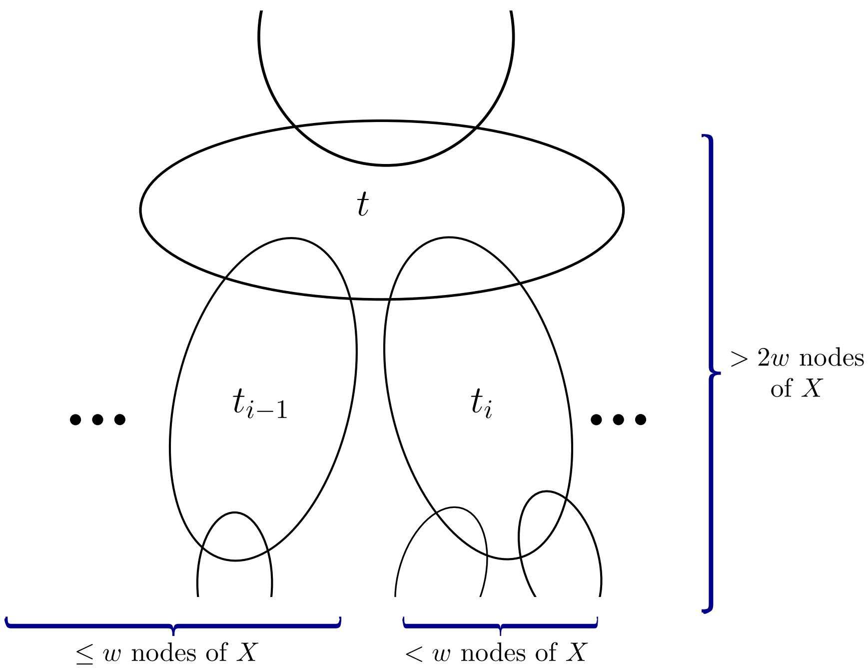


Figure 3.1: Combining subtrees until $> w$ nodes of X are obtained.

Moreover, the following theorem guarantees that a set can be tested for w-linkedness in reasonable time.

Theorem 3.4 (based on Kleinberg and Tardos [21]). *Let $G = (V, E)$ be a graph, let $X \subseteq V$ be a set of k vertices, and let $w \leq k$ be a given parameter. We can determine whether X is w-linked in time $\mathcal{O}(f(k) \cdot |E|)$. If it is not, we can give sets $Y, Z \subseteq X$ and $S \subseteq V$ that confirm it.*

Proof. Enumerate all pairs of subsets $Y, Z \subseteq X$ satisfying $|Y| = |Z| \leq w$. X has 2^k subsets, so there are $\leq 4^k$ such pairs.

For each pair of subsets, let $\ell = |Y| = |Z| \leq w$. We need to check if some set S of size $< \ell$ separates Y and Z. By theorem 1.1, the size of the smallest S that separates them is exactly the maximum number of vertex-disjoint paths from Y to Z, therefore if this number of paths is $< \ell$ then Y and Z are separable.

To compute these paths, we construct a flow network from the graph with unit capacity edges as follows:

1. Each node $v \in V$ is replaced with two nodes v_{in} and v_{out}.
2. An edge (v_{in}, v_{out}) is added for each pair of new nodes. This effectively restricts each node in the original graph to be used just once, as only one unit of flow can go through the edge.
3. For each undirected edge $uv \in E$, we add edges (u_{out}, v_{in}) and (v_{out}, u_{in}) to the network.
4. A source s is introduced and an edge (s, v_{in}) inserted for each $v \in Y$.
5. Similarly, a new node t is created and edges (v_{out}, t) inserted for nodes $v \in Z$.

We can check that the maximum flow from s to t gives us the number of vertex-disjoint paths from Y to Z. An algorithm like Ford-Fulkerson's computes this max-flow in time $\mathcal{O}(\ell \cdot |E|)$.

After checking all pairs, the total running time is $\mathcal{O}(f(k) \cdot |E|)$ where f is a function that only depends on k. □

3.2 Description of the algorithm

Given a graph $G = (V, E)$ and some fixed parameter w, following these steps will lead us to either a tree decomposition of G of width $< 4w$ or a $(w+1)$-linked set of size $\geq 3w$, which would mean that the treewidth of G is $\geq w$ by theorem 3.3. The running time for the algorithm will be $\mathcal{O}(f(w) \cdot |E| \cdot |V|)$, where f is an exponential function that depends only on the parameter w.

The algorithm works iteratively in a greedy fashion. We start choosing any subset $V_t \subseteq V$ such that $|V_t| \leq 3w$ as the first bag of the tree decomposition $(T, (V_t)_{t \in T})$. Then we proceed to expand the tree decomposition step by step (if possible) until it covers the whole graph.

At any iteration of the algorithm two invariants must hold. Let $U = \bigcup_{t \in T} V_t$:

$\boxed{\text{I1}}$ We have a partial tree decomposition:
$(T, (V_t)_{t \in T})$ is a tree decomposition of $G[U]$ of width $< 4w$.

$\boxed{\text{I2}}$ Each component C of $G \setminus U$ has $\leq 3w$ neighbours in U and some bag V_t contains all of them:
This invariant ensures that we can grow the tree decomposition adding a new bag from C.

We now describe the iterative step, and we will see that it maintains both invariants and U grows strictly larger.

Let C be a component of $G \setminus U$, let $X \subseteq U$ be the set of neighbours of C in U and let V_t be a bag that contains X as guaranteed by $\boxed{\text{I2}}$. If $|X| < 3w$, then pick any $v \in C$, set $V_s = X \cup \{v\}$ and make s a leaf of t (fig. 3.2). Since $|X \cup \{v\}| \leq 3w$ and for all edges (v, u) where $u \in U$ we

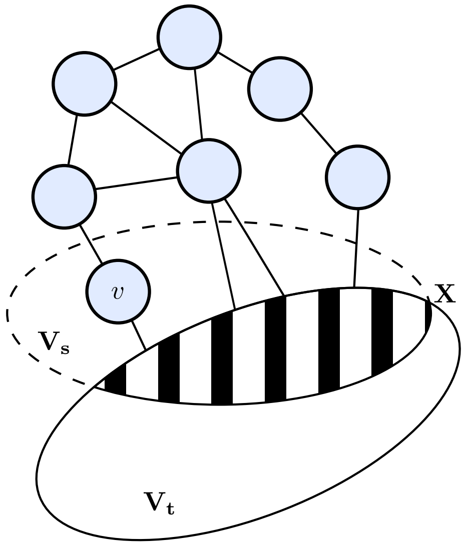


Figure 3.2: New bag V_s when $|X| < 3w$.

have that $u \in X$, both $\boxed{\text{I1}}$ and $\boxed{\text{I2}}$ are maintained. Furthermore, U has grown in one vertex so the step is valid.

In case $|X| = 3w$, it might be the case that G has no low-width tree decomposition, so first of all we will check if X is $(w+1)$-linked. By theorem 3.4 this can be done in time $\mathcal{O}(f(w) \cdot |E|)$. If the outcome is positive, then we can stop the algorithm and output that $tw(G) \geq w$. Otherwise, we now have sets $Y, Z \subseteq X$ and $S \subseteq V$ such that $|S| < |Y| = |Z| \leq w + 1$ and S separates Y and Z in G, which we will use to extend the tree decomposition.

Set $S' = S \cap C$. Observe that $|S'| \leq |S| \leq w$, and also note that $S' \neq \emptyset$ because in that case, as Y and Z have edges into C, there would exist some path starting in Y that jumps to C, travels though C, and jumps back to Z contradicting the fact that S separates Y and Z (fig. 3.3). Our new bag will be $V_s = X \cup S'$, being s a leaf of t. $\boxed{\text{I1}}$ holds since all edges from S' into U have their ends in X and $|X \cup S'| \leq 3w + w = 4w$.

To see that $\boxed{\text{I2}}$ still holds, let $C' \subset C$ be any component of $G \setminus (U \cup S')$. C' clearly has all of its neighbours in $X \cup S'$, but we have to make sure that there are $\leq 3w$ of them. We claim that all of them belong to one of the two subsets $(X \setminus Z) \cup S'$ or $(X \setminus Y) \cup S'$, both of them having size $< 3w$ as $|X| = 3w$ and $|S'| < |Y| = |Z|$. If this was not true, there would be two neighbours of C' one in Y and the other in Z, making a path through C' from Y to Z which has already been proved impossible. Therefore, the invariant holds, and to complete the argument we must see that the new U is strictly larger than the previous, because it now covers $U \cup S'$ where $S' \neq \emptyset$.

Finally, the most time-expensive operation for adding a new bag to the partial tree decomposition is to check whether the set X is $(w+1)$-linked, with a running time of $\mathcal{O}(f(w) \cdot |E|)$. In the worst case scenario, this operation is repeated $|V|$ times, as the number of vertices covered by the tree decomposition increases in each iteration. Hence the total running time is $\mathcal{O}(f(w) \cdot |E| \cdot |V|)$.

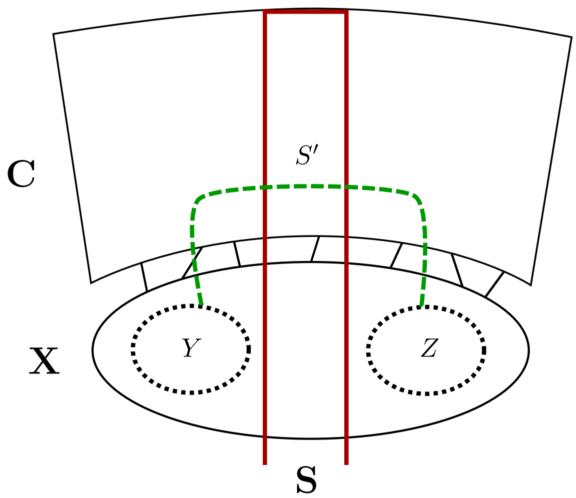


Figure 3.3: $S' \neq \emptyset$ as the path in green from Y to Z must traverse S.

Chapter 4

Algorithms for graphs of bounded treewidth

One of the most well-known applications of the treewidth is, as suggested in the abstract, to efficiently solve problems on graphs where the treewidth is low, problems that would be intractable for arbitrary graphs.

In this chapter we will study some dynamic programming algorithms that find optimal solutions in time $\mathcal{O}(f(w) \cdot p(n))$, where f is a function only depending on the treewidth w of the input graph and p is a polynomial on the size n of the graph. These distinctive running times make all these problems *fixed-parameter tractable*: if the treewidth can be fixed to a relatively small value, then the problem can be solved in reasonable time.

Although the structure of tree decomposition is the classical characterization of the treewidth of a graph, many problems are easier to describe and solve using dynamic programming if we work with similar but more restricted forms of tree decompositions. For convenience, nice path decompositions and nice tree decompositions are presented and used in this chapter, even though regular tree decompositions could be used as well.

4.1 Path decompositions

The structure of path decomposition is a more restricted version of the tree decomposition that simplifies the definition of some dynamic programming algorithms. We will use it in the following sections.

Definition 4.1. A *path decomposition* of a graph G is a tree decomposition of G with the underlying tree T being a path. It is usually denoted as the list of the bags that conform it, $(V_1, V_2, ..., V_r)$.

The width of a path decomposition is defined in the same way that the width of a tree decomposition:

$$\max\{|V_t| - 1 : t \in T\}$$

Analogous to the treewidth, the pathwidth of G, $pw(G)$, corresponds to the minimum width among all possible path decompositions of G. For any graph G, clearly $tw(G) \leq pw(G)$ since any path decomposition can be viewed as a tree decomposition. The properties of tree decompositions seen in section 2.1 also apply to path decompositions.

As seen in [22] and [5], the pathwidth of a graph is directly related to its treewidth and its number of vertices. Refer to the former for the proof of the next lemma.

Lemma 4.2. *For every forest F on n vertices, $pw(F) = \mathcal{O}(\log n)$.*

Consequently:

Theorem 4.3. *For every graph G on n vertices, $pw(G) \leq c \cdot tw(G) \cdot \log n$ for some constant c.*

Proof. Let $(T, (V_t)_{t \in T})$ be a nonredundant tree decomposition of G of width $tw(G)$, having $\leq n$ bags by lemma 2.3. Find a path decomposition of T of width $c \cdot \log n$ for some constant c, $(W_1, W_2, ..., W_k)$, whose existence is proved in lemma 4.2. Then define a path decomposition of G, $(X_1, X_2, ..., X_k)$, where $X_i = \bigcup_{j \in W_i} V_j$.

This is a valid path decomposition since each edge is present in some V_i, hence in some X_i; and for any $v \in V(G)$ the bags in (T, V_t) containing them form a connected subtree, then the nodes of those bags also form a connected subpath in the path decomposition of T, leading to a legal path decomposition of G. The size of each X_i is at most $c \cdot tw(G) \cdot \log n$ thus the theorem holds. $\square$

4.1.1 Nice path decompositions

Furthermore, we say that a path decomposition $(V_1, V_2, ..., V_r)$ is *nice* if it satisfies these properties:

1. $|V_1| = |V_r| = 1$.
2. For every $1 \leq i < r$, there is a vertex $v \in V(G)$ such that $V_{i+1} = V_i \cup \{v\}$, $v \notin V_i$, or $V_{i+1} = V_i \setminus \{v\}$, $v \in V_i$.

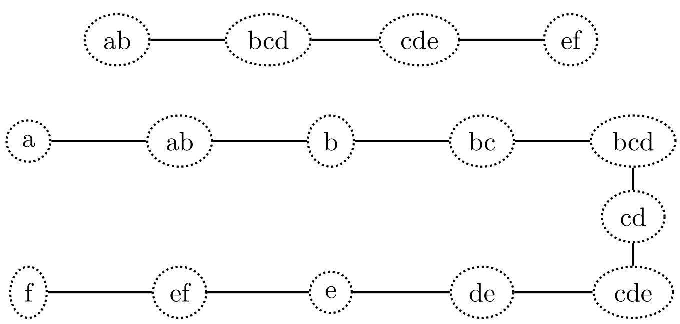


Figure 4.1: A path decomposition and its nice counterpart.

Nice path decompositions can be obtained from regular path decompositions as the following lemma shows.

Lemma 4.4. *Let $P = (V_1, V_2, ..., V_r)$ be a path decomposition of a graph G of width w. Then G has a nice path decomposition of width w and it can be constructed from P in linear time on $|V(G)|$.*

Proof. The procedure to transform the path decomposition works by adding new bags between every two V_i and V_{i+1}. Bags are inserted to the right of V_i following the recurrence $V_{j+1} = V_j \setminus \{v\}$ where $v \in V_j \setminus V_{i+1}$, until a bag containing exactly $V_i \cap V_{i+1}$ is achieved. From that point on, the new bags added will follow the recurrence $V_{j+1} = V_j \cup \{v\}$ where $v \in V_{i+1} \setminus V_j$ one at a time, until V_{i+1} is reached.

For the bags on the ends, V_1 and V_r, it is enough to keep adding bags to the left and to the right, respectively, removing one vertex at a time until single-vertex bags are achieved.

These steps lead us to a nice path decomposition of width w as one can easily check. The running time is linear in $|V(G)|$ since the number of new bags is at most twice the number of vertices in G, because each vertex is introduced and removed by the recurrences not more than once. □

4.2 Maximum Cut

Definition 4.5. Let $G = (V, E)$ be a graph and let $A, B \subseteq V$ be sets of vertices. We define $\mathrm{CUT}(A, B)$ to be the number of edges from E that have one end in A and the other in B.

The problem of finding the maximum cut on a graph $G = (V, E)$ consists in finding a subset $X \subseteq V$ such that the value of $\mathrm{CUT}(X, V \setminus X)$ is maximum. We refer to this value as the *size* of the cut.

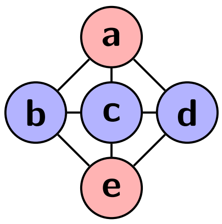


Figure 4.2: A maximum cut (size 6) represented in two colours.

The associated decision problem, i.e., given G and k determine if there is a cut of size $\geq k$ in G, is known to be NP-complete [17]. However, a nice path decomposition makes it relatively easy to compute for graphs of low pathwidth as seen in [16].

Theorem 4.6. *Let $G = (V, E)$ be a graph on n vertices. With a given path decomposition of width $\leq w$, the maximum cut problem on G can be solved in time $\mathcal{O}(2^w \cdot w \cdot n)$.*

Proof. Using lemma 4.4 we transform the path decomposition into a nice path decomposition $(V_1, V_2, ..., V_r)$ in linear time. Then set

$$W_i = \bigcup_{j=1}^{i} V_j$$

for every $1 \leq i \leq r$.

For a given i, let (A, B) be a partition of V_i. We define $c_i(A, B)$ to be the maximum size of a cut on the graph $G[W_i]$, taken over all partitions (X, Y) of W_i that preserve the partition (A, B), i.e., $A \subseteq X$ and $B \subseteq Y$. The values of $c_i(A, B)$ can be computed using dynamic programming.

Computing the values of c_1 is trivial. Let $V_1 = \{v\}$, there are only two possible partitions, $(\{v\}, \emptyset)$ and $(\emptyset, \{v\})$. In any case, $c_1(\{v\}, \emptyset) = c_1(\emptyset, \{v\}) = 0$ because $G[W_1]$ has a single vertex and no edge.

For $i > 1$, we will consider two scenarios:

- $V_i = V_{i-1} \cup \{v\}$ for some $v \notin V_{i-1}$. Observe that $v \notin W_{i-1}$ by (iii) in definition 2.1, therefore $W_i = W_{i-1} \cup \{v\}$. Also notice that all neighbours of v in $G[W_i]$ must be in V_i to satisfy (ii) in definition 2.1. Then for every partition (A, B) of V_i:

$$c_i(A,B) = \begin{cases} c_{i-1}(A \setminus \{v\}, B) + \mathrm{CUT}(\{v\}, B) & \text{if } v \in A \\ c_{i-1}(A, B \setminus \{v\}) + \mathrm{CUT}(\{v\}, A) & \text{if } v \notin A \end{cases}$$

The recurrence above works because the new v has neighbours only in V_i so introducing v does not affect $W_i \setminus V_i$, whose maximum cut is already computed. Furthermore, given a partition (A, B) of V_i (recall that we compute each possible partition), either $v \in A$ or $v \in B$, so the size of the cut increases by the number of edges between v and the vertices in the opposite set of the partition (fig. 4.3).

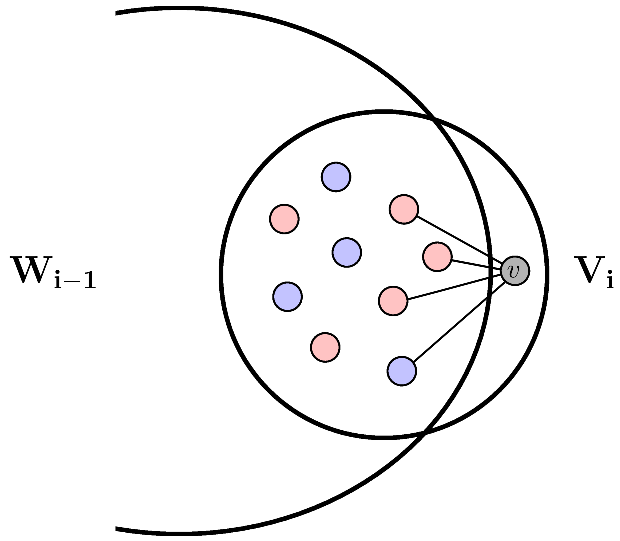


Figure 4.3: The size of the cut depends on the colour of v, which is given by (A, B).

- $V_i = V_{i-1} \setminus \{v\}$ for some $v \in V_{i-1}$. Considering that $W_i = W_{i-1}$, for every partition (A, B) of V_i:

$$c_i(A,B) = \max\{c_{i-1}(A \cup \{v\}, B), c_{i-1}(A, B \cup \{v\})\}$$

With this definition, the maximum size of a cut on G is:

$$\max\{c_r(A,B) : (A,B) \text{ is a partition of } V_r\}$$

Since $|V_r| = 1$ there are just two possible partitions, so this is equivalent to:

$$\max\{c_r(V_r, \emptyset), c_r(\emptyset, V_r)\}$$

The actual vertices of the maximum cut can be obtained by tracking back the choices made.

Computing the value of $c_i(A, B)$ for each of the $2^{|V_i|}$ partitions requires at most $|V_i|$ operations: $\mathrm{CUT}(\{v\}, A)$ or $\mathrm{CUT}(\{v\}, B)$ takes linear time. Recall that $|V_i| \leq w + 1$. The nice path decomposition has a number of bags linear in n, thus the total running time is

$$\mathcal{O}(\sum_{i=1}^{r} 2^{|V_i|} \cdot |V_i|) = \mathcal{O}(2^w \cdot w \cdot n)$$

□

4.3 Minimum Bisection

Definition 4.7. Let $G = (V, E)$ be a graph on n vertices. The minimum bisection problem consists in finding a partition of V into two sets (A, B) of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, such that $\mathrm{CUT}(A, B)$ is minimized.

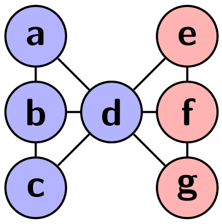


Figure 4.4: A minimum bisection (size 3) represented in two colours.

The minimum bisection problem is a classical NP-hard problem [18] that has been widely studied in the past. Several approximations and polynomial algorithms for special graph classes exist (cf. [27]), but we put the focus on graphs with bounded pathwidth.

Theorem 4.8. *Let $G = (V, E)$ be a graph on n vertices. With a given path decomposition of width $\leq w$, the minimum bisection problem on G can be solved in time $\mathcal{O}(2^w \cdot w \cdot n^2)$.*

Proof. The proof of this theorem is similar to that of theorem 4.6. Using lemma 4.4 we transform the path decomposition into a nice path decomposition $(V_1, V_2, ..., V_r)$ in linear time. Then set

$$W_i = \bigcup_{j=1}^{i} V_j$$

for every $1 \leq i \leq r$.

Now for a given i, let (A, B) be a partition of V_i and let ℓ be an integer $0 \leq \ell \leq n$. Define $b_i(A, B, \ell)$ to be the minimum cut size over partitions (X, Y) of the graph $G[W_i]$ that preserve the partition (A, B) (in other words, $A \subseteq X$ and $B \subseteq Y$) and $|X| = \ell$. The values of $b_i(A, B, \ell)$ can be computed and stored in a table as described below. A value of ∞ means that no partition is possible for the given parameters.

If $i = 1$, then $|V_i| = 1$ and the values of b_1 are:

$$b_1(V_1, \emptyset, 1) = b_1(\emptyset, V_1, 0) = 0$$
$$b_1(V_1, \emptyset, \ell) = b_1(\emptyset, V_1, \ell) = \infty \text{ for the remaining values of } \ell$$

If $i > 1$, there are two possible cases depending on the type of node:

- $V_i = V_{i-1} \cup \{v\}$ for some $v \notin V_{i-1}$. As in theorem 4.6, observe that all neighbours of v in $G[W_i]$ are in V_i. For every partition (A, B) of V_i and for every $0 \leq \ell \leq n$,

$$b_i(A, B, \ell) = \begin{cases} \infty & \text{if } \ell < |A| \text{ or } \ell > |W_i| \\ b_{i-1}(A \setminus \{v\}, B, \ell - 1) + \mathrm{CUT}(\{v\}, B) & \text{if } |A| \leq \ell \leq |W_i| \text{ and } v \in A \\ b_{i-1}(A, B \setminus \{v\}, \ell) + \mathrm{CUT}(\{v\}, A) & \text{if } |A| \leq \ell \leq |W_i| \text{ and } v \notin A \end{cases}$$

  The value of b_i is set to ∞ if the looked up b_{i-1} is ∞.

  The reasoning behind this step is analogous to the one in theorem 4.6, with the introduction of the parameter ℓ that restricts the size of the sets in the partition. The bounds of ℓ

are $|A|$ and $|W_i|$ because a partition (X, Y) of $G[W_i]$ that preserves (A, B) clearly has $|A| \leq |X| \leq |W_i|$.

- $V_i = V_{i-1} \setminus \{v\}$ for some $v \in V_{i-1}$. In this case $W_i = W_{i-1}$, then for every partition (A, B) of V_i and for every $0 \leq \ell \leq n$,

$$b_i(A, B, \ell) = \min\{b_{i-1}(A \cup \{v\}, B, \ell), b_{i-1}(A, B \cup \{v\}, \ell)\}$$

  As in the previous case, b_i is set to ∞ if both b_{i-1} values are ∞.

The result of the minimum bisection problem is the minimum value among these four:

$$\min\{b_r(V_r, \emptyset, \lceil n/2 \rceil), b_r(V_r, \emptyset, \lfloor n/2 \rfloor), b_r(\emptyset, V_r, \lceil n/2 \rceil), b_r(\emptyset, V_r, \lfloor n/2 \rfloor)\}$$

which will be only two different values if n is even. Recall that $|V_r| = 1$ so there are just two possible partitions of V_r. The actual partition can be computed by tracking back the choices of the algorithm.

The running time of the algorithm is given by the time needed to fill the $b_i(A, B, \ell)$ table. For each i, there are at most $2^{|V_i|}$ partitions (A, B) of V_i and n different values of ℓ. Each value can be computed in $\mathcal{O}(|V_i|)$ time, determined by the cost of the most time-expensive operation: $\text{CUT}(\{v\}, A)$ or $\text{CUT}(\{v\}, B)$. Since the nice path decomposition has $\mathcal{O}(n)$ bags, the total running time is

$$\mathcal{O}(\sum_{i=1}^{r} 2^{|V_i|} \cdot |V_i| \cdot n) = \mathcal{O}(2^w \cdot w \cdot n^2)$$

□

4.4 Counting homomorphisms

We introduced the concept of nice path decomposition before because it was useful to perform dynamic programming over it. Now we will use a similar strategy for tree decompositions.

A tree decomposition $(T, (V_t)_{t \in T})$ is *nice* if

1. T is rooted.
2. Every node $t \in T$ is one of the following:
   - *Join* node: Has two children $t_1, t_2 \in T$ and $V_t = V_{t_1} = V_{t_2}$.
   - *Introduce* node: Has one child $t' \in T$ and $V_t = V_{t'} \cup \{v\}$ for some $v \notin V_{t'}$.
   - *Forget* node: Has one child $t' \in T$ and $V_t = V_{t'} \setminus \{v\}$ for some $v \in V_{t'}$.
   - Leaf node: Has no child and contains a single vertex.

A lemma analogous to lemma 4.4 can be proved for nice tree decompositions.

Lemma 4.9. *Let G be a graph on n vertices. Given a tree decomposition of G of width k, it can be transformed in time $\mathcal{O}(n)$ into a nice tree decomposition of G of width k and with at most $(k+3)n$ nodes.*

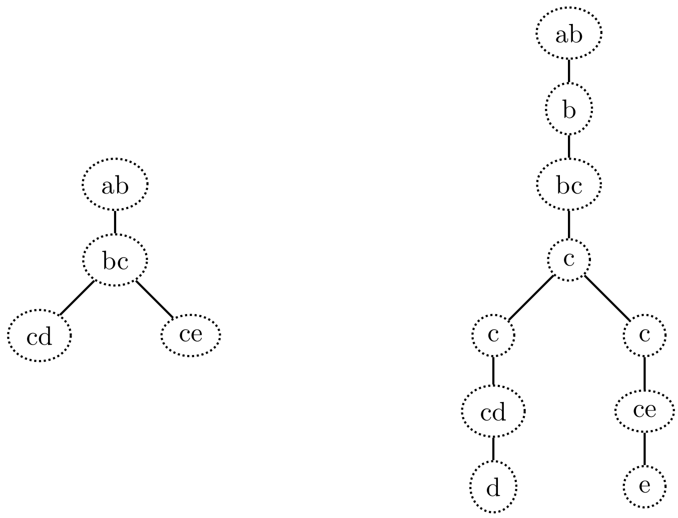


Figure 4.5: A tree decomposition and its nice counterpart.

Proof. Transform the given tree decomposition into a nonredundant one, $(T, (V_t)_{t\in T})$, as shown in section 2.1. The new tree decomposition has $\leq n$ bags by lemma 2.3.

Let $|T| = m$. We will show inductively that a tree decomposition of G of width k and m nodes can be transformed in linear time into a nice tree decomposition of G of width k with at most $(k+3)n$ nodes preserving the bags of the original tree decomposition, i. e., all the bags in $(T, (V_t)_{t\in T})$ are present in the nice tree decomposition.

If $m = 1$, root the tree T at its single node and add child nodes forming a path where each child has one vertex less than its parent, until a leaf is achieved. This tree decomposition clearly has n nodes, the original single bag is still present, the width has not changed and time $\mathcal{O}(n)$ is needed.

For the case $m > 1$, let t be a leaf of T and t' its neighbour. Set $U = V_t \setminus V_{t'}$. Then the deletion of t and its bag V_t yields a tree decomposition of $G \setminus U$ of width $\leq k$ as seen in lemma 2.6. By the induction hypothesis, a nice tree decomposition can be obtained in time $\mathcal{O}(n)$ from the previous one maintaining the width and the original bags, and consisting of at most $(k+3)(n-|U|)$ nodes.

Let x be a node in the nice tree decomposition such that $V_x = V_{t'}$.

- If x is not a leaf, transform it into a join node by inserting two children x_1 and x_2 with $V_x = V_{x_1} = V_{x_2}$. Set the original children of x as children of x_1 and then insert a new path of nodes under x_2, with an introduce node for each $v \in V_{t'} \setminus V_t$ and a forget node for each $u \in U$.

- If x is a leaf, insert a new path as in the previous case directly beneath x.

Notice that the last inserted bag is exactly V_t. If $|V_t| > 1$, then insert new nodes as in the case $m = 1$ until a leaf of size 1 is achieved.

We have got a nice tree decomposition of G, of width k since the original bags are preserved and the new bags are always smaller. The number of new bags in the worst case (x not being a leaf) is $2 + |V_{t'} \setminus V_t| + |U| + (|V_t| - 1) \leq 3k + 1$ considering that $|U| \leq k$. This makes a total number of nodes of at most $(k+3)(n-|U|) + 3k + 1 \leq (k+3)n - k^2 + 1 \leq (k+3)n$ since $-k^2 + 1 \leq 0$ for any $k > 0$. Finally, the process takes time $\mathcal{O}(n)$, which completes the proof. $\square$

Definition 4.10. A homomorphism is a mapping h between two graphs F and G that preserves adjacency, i.e., $uv \in E(F) \Rightarrow (h(u), h(v)) \in E(G)$.

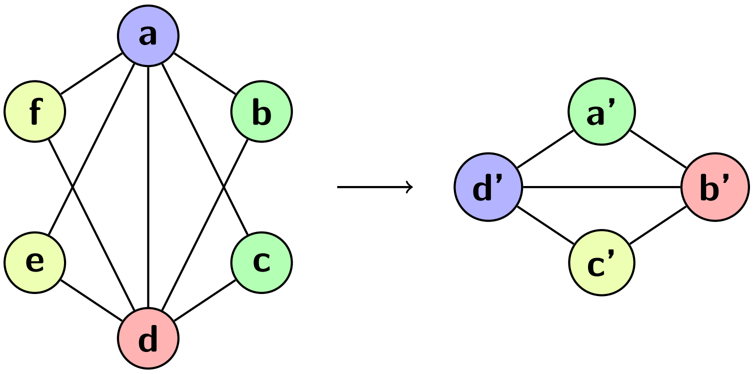


Figure 4.6: An homomorphism between two graphs, nodes of the same colour are mapped.

In general, counting how many homomorphisms there are from one arbitrary graph to another is #P-complete [14], but we can do better if the source graph has bounded treewidth. The following theorem shows how, based on the proof presented in [16].

Theorem 4.11. *Let F and G be graphs on m and n vertices, respectively. Given a tree decomposition $(T, (V_t)_{t \in T})$ of F of width w, the number of homomorphisms $hom(F, G)$ from F to G can be computed in time $\mathcal{O}(w \cdot m \cdot n^{w+1} \cdot \max\{w, n\})$ and space $\mathcal{O}(w \cdot m \cdot n^{w+1})$.*

Proof. Transform the tree decomposition into a nice one by lemma 4.9, and let r be the root of T. For each node $i \in T$, set

$$W_i = \bigcup_j V_j$$

where j runs through all nodes in T that lie below i plus i itself, this is, i is on the path from j to r. Also set $F_i = F[W_i]$.

For every node i and for every mapping $\Phi : V_i \to V(G)$, we define $hom(F_i, G, \Phi)$ to be the number of homomorphisms h from F_i to G that are an extension of Φ, or in other words, for every $v \in V_i$, $h(v) = \Phi(v)$. The values of $hom(F_i, G, \Phi)$ can be computed using dynamic programming, starting from the leaves and according to the type of each node i:

- Leaf node: $V_i = W_i$ and there is only one vertex in V_i, so for any mapping $\Phi : V_i \to V(G)$, $hom(F_i, G, \Phi) = 1$.

  Time to compute for all Φ: $\mathcal{O}(n^{|V_i|})$.

- Introduce node: Let j be its child and let $v = V_i \setminus V_j$. Clearly $W_i = W_j \cup \{v\}$ and $v \notin W_j$, so F_i results from adding v and some edges incident to v to F_j. Also observe that all neighbours of v in F_i are in V_i. This means that homomorphisms from F_i to G are extensions of those from F_j to G that preserve the new edges.

  More precisely, for any mapping $\Phi : V_i \to V(G)$ where the neighbours of v in F_i are mapped to neighbours of $\Phi(v)$ in G, $hom(F_i, G, \Phi) = hom(F_j, G, \Psi)$ where $\Psi : V_j \to V(G)$ is the mapping such that $\Phi(u) = \Psi(u)$ for any $u \in V_j$. If the mapping Φ does not preserve the edges of v, $hom(F_i, G, \Phi) = 0$.

  Time to compute for all Φ: $\mathcal{O}(n^{|V_i|} \cdot |V_i|)$ because for each mapping Φ all neighbours of v have to be checked, which are at most $|V_i| - 1$.

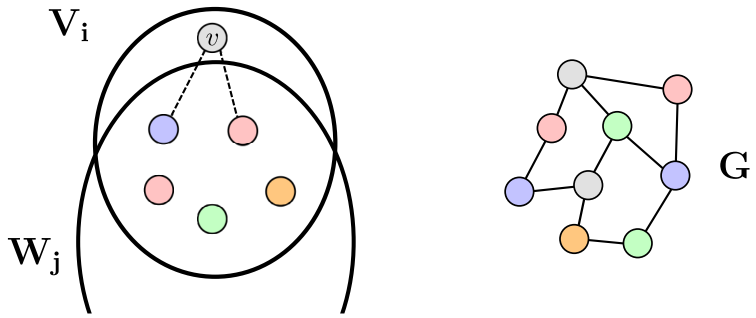


Figure 4.7: The mapping of v is given by Φ, but adjacency preservation has to be checked.

- Join node: Let j and k be its children. By lemma 2.8, there are no edges between $F_j \setminus V_i$ and $F_k \setminus V_i$. Then for every $\Phi : V_i \to V(G)$, $hom(F_i, G, \Phi) = hom(F_j, G, \Phi) \cdot hom(F_k, G, \Phi)$.

  Time to compute for all Φ: $\mathcal{O}(n^{|V_i|})$.

- Forget node: Let j be its child and let $v = V_j \setminus V_i$. Notice that $F_i = F_j$, thus homomorphisms from F_i to G are the same as homomorphisms from F_j to G.

  Then for every $\Phi : V_i \to V(G)$, $hom(F_i, G, \Phi) = \sum hom(F_j, G, \Psi)$ over all mappings $\Psi : V_j \to V(G)$ such that $\Phi(u) = \Psi(u)$ for any $u \in V_i$, this is, we add up the number of homomorphisms for every possible mapping of the removed vertex v.

  Time to compute for all Φ: $\mathcal{O}(n^{|V_i|} \cdot n)$ because for each mapping Φ the vertex v can be mapped to any of the n vertices in G.

The overall number of homomorphisms from F to G is

$$hom(F, G) = \sum_{\Phi : V_r \to V(G)} hom(F_r, G, \Phi)$$

T has at most $(w + 3)m$ nodes, so in the worst case, computing $hom(F, G)$ requires $\mathcal{O}(w \cdot m \cdot n^{w+1} \cdot \max\{w, n\})$. For each node and for each mapping Φ, the number of homomorphisms has to be stored, taking $\mathcal{O}(w \cdot m \cdot n^{w+1})$ space. □

4.5 Maximum-Weight Independent Set

Definition 4.12. Given a graph $G = (V, E)$ with each vertex v assigned a weight w_v, a Maximum-Weight Independent Set of G is a subset of the vertices whose weights sum as much as possible and no two of them are adjacent.

The problem of finding such a set in an arbitrary graph is NP-hard, but there exist efficient algorithms for special graph classes like trees [23, 11]. In this case, we will follow the idea from the linear-time algorithm for trees and apply it to tree decompositions, hopefully achieving a reasonable running time.

The previously presented algorithms take advantage of structures related to tree decompositions that allow an easier definition of the solution, like path decompositions or nice tree decompositions. This time we will use regular tree decompositions to demonstrate that algorithms can be presented as well without any additional structure, like Kleinberg and Tardos did

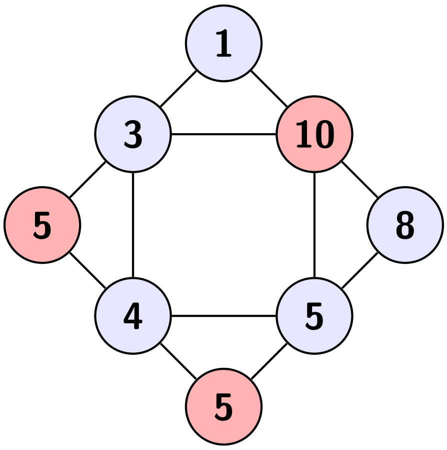


Figure 4.8: A maximum weight (20) independent set, in red.

[21]. One can notice that the complexity of this solution is higher than before and the reasoning is more difficult to follow.

Theorem 4.13. *Let $G = (V, E)$ be a graph on n vertices. Given a tree decomposition $(T, (V_t)_{t \in T})$ of G of width w, the maximum weight independent set problem on G can be solved in time $\mathcal{O}(4^w \cdot w \cdot n)$.*

Proof. We can safely assume that the tree decomposition is nonredundant, as it can be transformed into one in linear time.

Roughly, we root the tree T and build the independent set from the bags in the leaves upwards. For a bag V_t, whose size is at most $w + 1$, we consider all possible 2^{w+1} subsets to be part of the optimal solution, like we did in the previous problems. Once one of the subsets is fixed, we will see that the maximum weight independent sets on the subtrees below t can be used to get the solution for the whole subtree rooted at t.

More precisely, root the tree T at a node r and let $t \in T$ be a node. W_t denotes the union of the bags beneath t and V_t itself, and set $G_t = G[W_t]$. For a subset $U \subseteq V$, let $w(U)$ be the total weight of the vertices in U, $w(U) = \sum_{u \in U} w_u$.

For each $U \subseteq V_t$, define $f_t(U)$ to be the maximum weight of an independent set S in G_t subject to $S \cap V_t = U$, this is, an independent set whose vertices in V_t are exactly U. The values of $f_t(U)$ are computed using dynamic programming and filling a table as usual, and once again, we will take into account two different situations.

- If t is a leaf, then for each independent set $U \subseteq V_t$, $f_t(U) = w(U)$. If U is not an independent set, then put $f_t(U) = -\infty$.
- Otherwise, t has children $t_1, t_2, ..., t_d$ with $d \geq 1$ and we may assume that the values of $f_{t_i}(U_i)$ for all t_i and $U_i \subseteq V_{t_i}$ are already computed. For each $U \subseteq V_t$, the recurrence to compute $f_t(U)$ is

$$f_t(U) = w(U) + \sum_{i=1}^{d} \max\{f_{t_i}(U_i) - w(U_i \cap U) : U_i \subseteq V_{t_i} \text{ and } U_i \cap V_t = U \cap V_{t_i}\}$$

In other words, the recurrence checks if each subset $U_i \subseteq V_{t_i}$ is an independent set and satisfies $U_i \cap V_t = U \cap V_{t_i}$, which is the condition needed to build the solution from the subproblems. If positive, the weight of the nodes in $U_i \cap U$ is subtracted to $f_{t_i}(U_i)$ to avoid counting the nodes in U more than once. The maximum of this values over all possible U_i is taken, and the process is repeated for every child of t. Finally, all the weights are added to $w(U)$ to get the value of $f_t(U)$.

In order to understand why this recurrence works, one has to observe how an optimal independent set S of G_t such that $S \cap V_t = U$ is related to the children of t. Let t_i be any child of t, and set S_i to the part of S that lies in G_{t_i}, i.e., $S_i = S \cap W_{t_i}$. It is easy to check that $S_i \cap V_t = W_{t_i} \cap U$, and $W_{t_i} \cap U = V_{t_i} \cap U$ because a vertex in both W_{t_i} and in V_t has to be in V_{t_i} too by the definition of tree decomposition, so we have that $S_i \cap V_t = V_{t_i} \cap U$ (see fig. 4.9). Thus, when looking at the subproblems, we consider just those $U_i \subseteq V_{t_i}$ that satisfy $U_i \cap V_t = U \cap V_{t_i}$ to guarantee that S can be built from S_i. Moreover, lemma 4.14 (see below) ensures that S_i is an optimal solution to the subproblem satisfying $S_i \cap V_t = V_{t_i} \cap U$, so its weight has already been computed and we can look it up in the table.

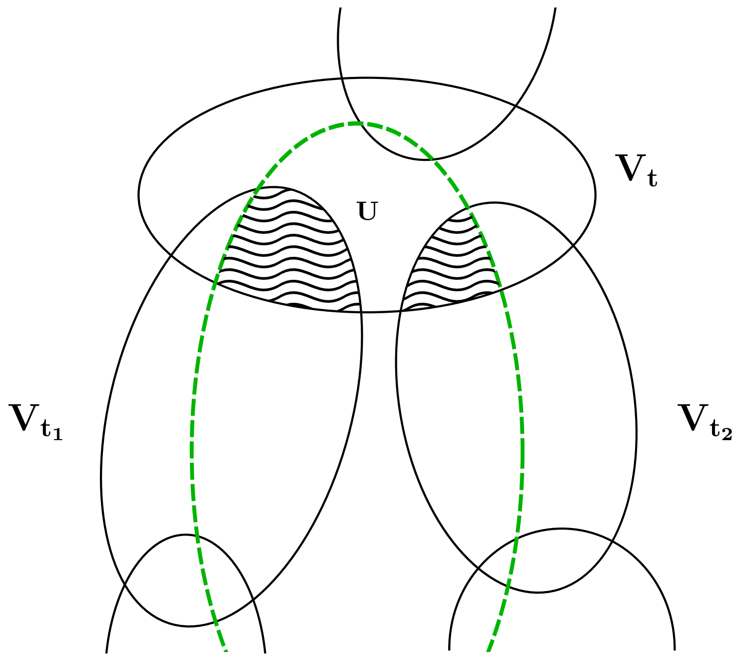


Figure 4.9: The green dashes delimit S, and the areas in waves are $S_i \cap V_t = V_{t_i} \cap U$.

The final solution comes from the root of the tree decomposition. We take the maximum $f_r(U)$ over all independent sets $U \subseteq V_r$. This gives the maximum weight, but if we need the independent set itself we can track back through the execution as usual.

The time required to compute a single $f_t(U)$ in the worst case is $\mathcal{O}(2^w \cdot w \cdot d)$: for each of the d children, 2^{w+1} sets U_i have to be considered, and checking the condition $U_i \cap V_t = U \cap V_{t_i}$ takes time $\mathcal{O}(w)$. There are 2^{w+1} sets U, which make a running time of $\mathcal{O}(4^w \cdot w \cdot d)$ for each f_t. As each node is counted as a child once and there are $\mathcal{O}(n)$ nodes in the tree decomposition by lemma 2.3, the total running time of the algorithm is $\mathcal{O}(4^w \cdot w \cdot n)$. □

Lemma 4.14. *S_i is a maximum weight independent set of G_{t_i} subject to $S_i \cap V_t = V_{t_i} \cap U$.*

Proof. Suppose for a contradiction that there is an independent set S'_i of G_{t_i} such that $S'_i \cap V_t = V_{t_i} \cap U$ and $w(S'_i) > w(S_i)$. Set $S' = (S \setminus S_i) \cup S'_i$. Clearly $w(S') > w(S)$ and $S' \cap V_t = U$. If S' was also an independent set, it would contradict the choice of S as the maximum weight independent set of G_t such that $S \cap V_t = U$, hence such S' could not exist and the lemma would hold.

So let us show that S' is an independent set. Again, suppose that it is not and let uv be an edge with $u, v \in S'$. By the choice of S and S'_i they are independent sets, so it cannot be that $u, v \in S$ or $u, v \in S'_i$. Thus, $u \in S \setminus S'_i$ and $v \in S'_i \setminus S$, and therefore u is not in G_{t_i} and v is in $G_{t_i} \setminus (V_{t_i} \cap V_t)$. This contradicts lemma 2.10 as there cannot be an edge joining u and v, then S' must be an independent set. $\square$

An algorithm for this problem in terms of a nice tree decomposition is easy to obtain by following the same idea we have explained, and it is likely easier to understand. Refer to [8] for such an algorithm. However, the purpose of this section was to show that regular tree decompositions can be used to formulate dynamic programming algorithms too.

Appendix A

A note on computing the treewidth

The algorithm seen in chapter 3 allows to compute the treewidth of an arbitrary graph by brute-force searching on the input parameter w. However, as the running time is exponential in w, in practice this is only feasible for low values of the treewidth. Since computing the treewidth is NP-hard, it is unlikely that a polynomial time algorithm is found, so the efforts are being directed towards finding heuristics that bound the treewidth and using state space search algorithms to find the exact value.

Several upper bound heuristics are given in [9] to construct elimination orderings, that can be later transformed efficiently into tree decompositions (cf. Lemma 8 in [9]). Their experiments show that these heuristics work reasonably well, refer to their article for more detailed results. The same authors studied lower bound heuristics in [10], which are useful for their use in branch-and-bound algorithms and to discard some approaches in graphs with high lower bounds on the treewidth. A web platform developed by one of the authors is available [1] where some of these heuristics are implemented.

In order to find the actual treewidth of the graph, we need an exact algorithm. One of the most famous ones is *QuickBB*, a branch and bound algorithm that searches in the space of elimination orderings of the graph [19]. Other recent algorithms [2012] are analysed and experimental results given in [7]. Similar experiments have been carried out recently [2014] for pathwidth [12].

There are few implementations of these algorithms publicly available. The most relevant one is probably LibTW[2], a library written in Java that implements the heuristics mentioned above and a couple of exact algorithms (branch and bound and a dynamic programming approach), available under the GNU LGPL. The library uses its own internal representation of graph, but can read them from input files in the DIMACS format. An analysis and comparison of the algorithms implemented is also available.

Another implementation is included in the SageMath mathematics software[3]. This software is standalone but open-source, so the code, written in Python, is accessible. It includes a method to compute the pathwidth of a graph too, using three different approaches.

[1] `http://www.math2.rwth-aachen.de/de/mitarbeiter/koster/ComputeTW/home`
[2] `http://www.treewidth.com/`
[3] `http://www.sagemath.org/`

Bibliography

[1] Ahuja, R. K., Orlin, J. B., and Magnanti, T. L. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] Arnborg, S., Corneil, D. G., and Proskurowski, A. Complexity of finding embeddings in a k-tree. *SIAM. J. on Algebraic and Discrete Methods 8*, 2 (1987), 277–284.

[3] Bertelé, U., and Brioschi, F. *Nonserial Dynamic Programming*. Academic Press, 1972.

[4] Bodlaender, H. L. A tourist guide through treewidth. Tech. rep., Utrecht University, 1993.

[5] Bodlaender, H. L. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science 209* (1998), 1–45.

[6] Bodlaender, H. L. Treewidth: Characterizations, applications, and computations. *Lecture Notes in Computer Science 4271* (2006), 1–14.

[7] Bodlaender, H. L., Fomin, F. V., Koster, A. M. C. A., Kratsch, D., and Thilikos, D. M. On exact algorithms for treewidth. *ACM Transactions on Algorithms 9*, 1 (2012).

[8] Bodlaender, H. L., and Koster, A. M. C. A. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal 51*, 3 (2008), 255–269.

[9] Bodlaender, H. L., and Koster, A. M. C. A. Treewidth computations i. upper bounds. *Information and Computation 208*, 3 (2010), 259–275.

[10] Bodlaender, H. L., and Koster, A. M. C. A. Treewidth computations ii. lower bounds. *Information and Computation 209*, 7 (2011), 1103–1119.

[11] Chen, G. H., Kuo, M. T., and Sheu, J. P. An optimal time algorithm for finding a maximum weight independent set in a tree. *BIT Numerical Mathematics 28*, 2 (1988), 353–356.

[12] Coudert, D., Mazauric, D., and Nisse, N. Experimental evaluation of a branch and bound algorithm for computing pathwidth. *Experimental Algorithms. Lecture Notes in Computer Science Volume 8504*, 1 (2014), 46–58.

[13] Diestel, R. *Graph Theory*, 4th ed. Springer-Verlag, 2010.

[14] Dyer, M., and Greenhill, C. The complexity of counting graph homomorphisms. *Random Structures and Algorithms 17*, 3-4 (2000), 260–289.

[15] Fiala, J. Graph minors, decompositions and algorithms, June 2014.

[16] Fomin, F. V., and Kratsch, D. *Exact Exponential Algorithms*. Springer, 2010.

[17] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[18] Garey, M. R., Johnson, D. S., and Stockmeyer, L. Some simplified np-complete graph problems. *Theoretical Computer Science 1*, 3 (1976), 237–267.

[19] Gogate, V., and Dechter, R. A complete anytime algorithm for treewidth. *Proceedings of the 20th conference on Uncertainty in Artificial Intelligence* (2004), 201–208.

[20] Heinz, M. Tree-decomposition. graph minor theory and algorithmic implications. Master's thesis, Technischen Universität Wien, 2013.

[21] Kleinberg, J., and Tardos, É. *Algorithm Design*. Addison-Wesley, 2005.

[22] Korach, E., and Solel, N. Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics 43*, 1 (1993), 97–101.

[23] Pawagi, S. Maximum weight independent set in trees. *BIT Numerical Mathematics 27*, 2 (1987), 170–180.

[24] Robertson, N., and Seymour, P. D. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B 36* (1984), 49–64.

[25] Seymour, P. D., and Thomas, R. Graph searching, and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B 58* (1993), 22–33.

[26] Thulasiraman, K., and Swamy, M. N. S. *Graphs: Theory and Algorithms*. John Wiley & Sons, 1992.

[27] van Bevern, R., Feldmann, A. E., Sorge, M., and Suchý, O. On the parameterized complexity of computing graph bisections. *Lecture Notes in Computer Science 8165* (2013), 76–87.

[28] Wilson, R. J. *Introduction to Graph Theory*, 4th ed. Longman, 1996.