Tree decompositions and linear time algorithms

Zhentao Li

Doctor of Philosophy

School of Computer Science

McGill University

Montreal,Quebec

2011-12-08

A thesis submitted to McGill University in partial fulfillment
of the requirements of the degree of Doctor of Philosophy

Dedicated to my parents.

ACKNOWLEDGEMENTS

First and foremost, I'd like to thank my supervisor, Bruce Reed, and co-supervisor, Adrian Vetta, whose support, help and guidance were essential to the success of this thesis. I would like to thank my officemates and colleagues, Rohan, Sean, Andrew, Omar and Jamie, as well as other professors at McGill for making my stay enjoyable. Special thanks also goes to Sean Kennedy for proofreading part of this thesis. I'd also like to thank the staff at the School of Computer Science for their help. I'd like to thank Ken-ichi Kawarabayashi, who hosted me at the National Institute of Informatics in summer 2009, and Anna Gallucio for hosting me at CNR IASI in summer 2011. Finally, I would like to thank NSERC for financial support.

PREFACE

Contributions of Authors: The main results in this thesis are joint work with Bruce Reed. The result of Chapter 4 is also joint work with Rohan Kapadia. In all cases, all authors contributed equally.

ABSTRACT

This thesis concerns tree decompositions. Trees are one of the simplest and most well understood class of graphs. A tree decomposition of a graph improves our understanding of the graph in a similar way. For example, as a consequence of Robertson and Seymour's groundbreaking work in the theory of graph minors, there are linear time algorithms for NP-hard problem on graphs that admit a tree decomposition of a certain type. We classify existing tree decompositions and examine what makes a tree decomposition unique.

The first result of this thesis is a linear time algorithm for building a tree decomposition for the class of graphs that exclude K_5 as a minor. The second result is a significant modification to this algorithm which results in a linear time algorithm to construct the tree decomposition for graphs which exclude a special set of paths. These are vertex disjoint paths between two pairs of input vertices $(s_1, t_1), (s_2, t_2)$, one from s_1 to t_1 and the other from s_2 to t_2.

We then use these tree decompositions to improve the running time of existing algorithms and extend the allowed input of other algorithms from planar graphs to graphs that exclude K_5 as a minor.

ABRÉGÉ

Cette thèse traite de décompositions arborescentes. Les arbres font partie des classes de graphes les mieux comprises. La décomposition arborescente d'un graphe améliore notre compréhension de ce dernier. Par exemple, grâce aux travaux de Robertson et Seymour sur les mineurs d'un graphe, nous savons qu'il existe, pour des problèmes qui sont en général NP-difficiles, un algorithme linéaire pour les graphes admettant une certaine décomposition arborescente. Nous classons les décompositions arborescentes connues et déterminons les propiétés qui rendent cette décomposition unique.

Comme premier résultat, nous donnons un algorithme linéaire pour construire une décomposition arborescente d'un graphe sans mineur du graphe complet K_5. Notre deuxième resultat repose sur une modification de cet algorithme afin d'obtenir un autre algorithme linéaire. Ce dernier permet la construction d'une décomposition arborescente d'un graphe qui ne contient pas deux chemins à sommets disjoints entre deux paires de sommets données (s_1, t_1) et (s_2, t_2).

Nous utilisons ces deux décompositions pour améliorer le temps de calcul des algorithmes existants et modifions des algorithmes pour graphes planaires pour leur permettre de prendre comme donnée des graphes sans mineur K_5.

TABLE OF CONTENTS

. ii

ACKNOWLEDGEMENTS iii

PREFACE iv

ABSTRACT v

ABRÉGÉ vi

LIST OF TABLES x

LIST OF FIGURES xi

1 Introduction 1

2 Tree decompositions and excluded structure 6

2.1 Block trees 7
2.1.1 Algorithm 8
2.2 Tree of 3-connected components 12
2.2.1 Algorithms 16
2.2.2 Other variants of the tree of triconnected components 16
2.2.3 Application: Planarity Testing 18
2.3 Laminar cutsets and component minimality 25
2.3.1 $\mathcal{F}$-block trees 28
2.3.2 Uniqueness of $\mathcal{F}$-block tree 30
2.4 Tree decomposition types 33
2.4.1 Recursive trees 33
2.4.2 Intersection trees 34
2.4.3 Tree type equivalence 36
2.5 Clique cutset trees 40
2.5.1 Building a clique cutset tree 43
2.5.2 Application: Colouring chordal graphs 46
2.6 Robertson-Seymour tree decomposition 49
2.6.1 Treewidth 49
2.6.2 Building bounded treewidth decompositions 51
2.6.3 Graph minors 51

2.6.4 Complete graph minors . . . 55
2.6.5 Wagner's theorem . . . 58
2.6.6 Treewidth and excluding planar graphs . . . 61
2.6.7 Structure theorem for H-minor free graphs . . . 61
2.6.8 Recognizing an H-minor free graphs . . . 62
2.6.9 Algorithmic proof of Wagner's theorem . . . 62
2.6.10 Bounded treewidth and linear time algorithms . . . 69

3 Recognizing K_5-minor free graphs . . . 76

3.1 $(3,3)$-block trees for K_5-minor free graphs . . . 78
3.2 Formal description . . . 80
3.3 The Structure of 2-cuts and 3-cuts . . . 85
3.3.1 The $(3,3)$-block tree is unique . . . 85
3.3.2 The Structure of 2-cuts . . . 87
3.3.3 Cutset Structure Relative to a Contracted Matching . . . 92
3.4 Reductions . . . 93
3.4.1 Finding a Matching or a Set of Special Degree 3 Vertices . . . 93
3.4.2 Massaging the Matching . . . 95
3.4.3 Decomposing . . . 100
3.5 New Solutions from Old . . . 102
3.5.1 Adding degree 3 vertices . . . 102
3.5.2 Uncontracting a Matching: Easy Case . . . 103
3.5.3 Combining $(3,3)$-Block Trees . . . 106
3.5.4 Uncontracting a Matching: Hard Case . . . 108

4 2-Disjoint Rooted Paths . . . 123

4.1 2-DRP and Attached K_5-Minors . . . 123
4.2 Previous work . . . 129
4.3 Closest Detachers and Laminarity . . . 130
4.4 Pruned K_5-tree . . . 132
4.5 Pruned $(3,3)$-tree . . . 133
4.6 Main algorithm . . . 134
4.6.1 Overview . . . 134
4.6.2 Formal description . . . 137
4.7 Finding the reductions . . . 144
4.8 Pre-recursion: Deleting vertices . . . 146
4.9 Pre-recursion: Contracting stars . . . 146
4.9.1 Maintaining Connectivity . . . 146
4.9.2 Small dense minors . . . 150
4.10 Pre-recursion: Contracting a matching . . . 151
4.11 Post-recursion: Adding vertices of degree at least 5 . . . 154
4.12 Post-recursion: Adding degree 3 vertices . . . 156

4.13 Locally bounded treewidth . . . 159
4.13.1 Previous work . . . 160
4.13.2 Locally bounded treewidth for face-vertex distance and K_5-minor free graphs . . . 161
4.13.3 Definitions . . . 162
4.13.4 Layers in 3-connected K_5-minor free graphs . . . 163
4.13.5 Extending layers to non-3-connected K_5-minor free graphs . . . 166
4.13.6 Red-blue colouring . . . 166
4.14 Post-recursion: Adding degree 4 vertices . . . 171
4.15 Post-recursion: Adding $(3,3)$-connectivity preserving edges . . . 174
4.15.1 Adding planar edges (F_1) . . . 176
4.15.2 Adding far non-planar edges (F_2) . . . 178
4.15.3 Adding close non-planar edges (F_3) . . . 180
4.16 Post-recursion: Uncontracting a matching . . . 181
4.16.1 Overview . . . 181
4.16.2 Details . . . 182
4.17 Post-recursion: Adding vertices in K_5-model nodes . . . 183
4.18 Post-recursion: Adding small components . . . 184

5 Applications . . . 185

5.1 Maximum independent set . . . 185
5.2 Subgraph isomorphism . . . 186
5.3 Improving linear time PTASs . . . 189
5.4 Bounded intersection tree decomposition . . . 189
5.5 k-Realizations . . . 191
5.5.1 Previous work and connection to graph minors . . . 191
5.5.2 Solving k-realizations on K_5-minor free graphs . . . 193

6 Concluding remarks . . . 197

REFERENCES . . . 198

LIST OF TABLES

Table page

2–1 Table of different trees introduced in this section 75

LIST OF FIGURES

Figure page

2–1 An example of a block tree. 7

2–2 An example of a palm tree, numberings num and low, and cutvertices (in white). 9

2–3 Two cutsets X in Y in P_5 and their decomponents. X is in different decomponents of $G - Y$ and Y is no longer a cut in the decomponent of $G - X$ containing Y. This is an example of the first two ways of losing a cut. 26

2–4 Three component minimal cuts $\mathcal{F} = \{\{a\}, \{a, b\}, \{a, c\}\}$ in a graph and three of the possible $\mathcal{F}$-block trees. 31

2–5 The three different types of trees where $\mathcal{F}$ is the family of all cutvertices of G. 35

2–6 A K_4-minor in a graph G and the corresponding model. 53

2–7 H is a minor of G (obtained by contracting the highlighted edge) but G has no H-subdivision as one center of such a subdivision needs to have degree at least 4 and G has maximum degree 3. 59

2–8 The special graph L with 8 vertices and 12 edges. It is also known as V_8 in the literature. 60

2–9 K_5-models obtained from a $K_{3,3}$-model. 64

2–10 K_5-model obtained from L with an added edge. 67

3–1 A K_5-minor obtained from two P_3's with both endpoints on F and distinct midpoints not on F. The added vertex is white. 111

4–1 A planar graph G with vertices s_1, t_1, s_2, t_2 on a face, two curves inside this face intersecting and the auxiliary graph G^* obtained by adding edges $s_1t_1, t_1s_2, s_2t_2, t_2s_1$ and a vertex x^* adjacent to s_1, t_1, s_2, t_2. 124

4–2 The 4-wheel, W_4 125

4–3 (a) A graph G with a set $A = \{s_1, t_1, s_2, t_2, x^*\}$ of vertices (b) A K_5(-model) in G with closest detacher $\{v_3, v_4, v_5\}$ (c) Another K_5(-model) in G, this one with closest detacher $\{v_1, v_2, v_3\}$ which is also a top detacher (d) A pruned K_5-tree of G (d) a red-blue colouring of G 132

4–4 L and $FV(L)$ 163

CHAPTER 1
Introduction

Trees are one of the most well understood class of graphs and often occur naturally, for example in divide-and-conquer algorithms. Sometimes when a recursive algorithm is used to divide a graph, it produces a recursion tree where each node is labelled by its input graph. Such recursion trees are an example of a tree decomposition, the topic of this thesis.

A tree decomposition for a graph is a tree with each node labelled by some (likely different) graph with restrictions on these labels (and the tree) depending on the specific type of tree decomposition we have. They allow us to solve hard problems on graphs for which we can build such a decomposition. This is not too surprising as these hard problems are easy to solve on trees (for example using dynamic programming). Frequently, tree decompositions express how tree-like a graph is and allow us to use similar techniques on graphs which are not trees. To benefit from these faster algorithms, we need to be able to quickly build a tree decomposition of interest in the first place.

Of course, a graph does not have a tree decomposition of every type. Many results in this area are of the form: every graph in which substructures of type A are excluded has a tree decomposition of type B. In order to be able to exploit such a result algorithmically, we need to be able to efficiently construct a tree decomposition of this type (or show that this is impossible because the input graph does not permit such a decomposition). The two main contributions in this thesis are algorithms of this type.

First, we present a linear time algorithm for building a tree decomposition called a $(3, 3)$-block tree, for graphs which exclude a clique of size 5 as a minor. Second, we modify this algorithm into a linear time algorithm for building a tree decomposition, called a pruned

K_5-tree, for graphs which exclude a minor of the clique of size 5 that is not separated by small cuts from a set of input vertices.

The problems our algorithms solve arise in the theory of graph minors (we recall that, as discussed in detail below, a graph H is a minor of a graph G if it can be obtained from a subgraph of G by contracting some edges). They relate to seminal work of Robertson and Seymour, motivated by a theorem and a conjecture of Wagner.

Wagner proved that every graph G from which we exclude a clique of size 5 as a minor has a special kind of tree decomposition. He conjectured that in any infinite family of graphs, there must be one graph containing another member of the family as a minor. In their seminal work, Robertson and Seymour show that for every graph H, those graphs not containing H as a minor permit a special type of tree decomposition (the type of tree decomposition depends on H). This allowed them to prove Wagner's conjecture.

Robertson and Seymour's result implicitly contains an $O(n^3)$ algorithm to construct such a tree decomposition. An explicit algorithm (whose running time is a function of H) was given by Demaine et al. [22]. This structural decomposition result can be used to obtain algorithms. For example Robertson and Seymour used it to provide $O(n^3)$ algorithms to determine if a graph has a specific graph H as a minor. They also used it to obtain an $O(n^3)$ algorithm for the k-DRP problem defined as follows: given a graph G and k pairs of vertices $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$, determine if there are k vertex disjoint paths, one from s_i to t_i for each i. Actually they provided an $O(n^3)$ algorithm for a common generalization of H-minor containment and k-DRP, the labelled minor containment problem (see [58]). We will consider H-minor containment for a specific H: K_5, and k-DRP for a specific k:2.

One of the main contributions of this thesis is providing a linear time algorithm which allows us to exploit Wagner's proof that K_5-minor free graphs have a special type of tree decomposition. Given an input graph it either constructs such a special tree decomposition or determines that G has a K_5-minor. This improves on the previously best known quadratic

algorithm of Kézdy and McGuinness [44]. Having obtained this tree decomposition we show that we can exploit it to obtain linear time algorithms for various optimization problems if we restrict our attention to K_5-minor free graphs.

A second major contribution is an algorithm which allows us to solve the 2-DRP problem in linear time. It improves on the previous best $O(n\alpha(m,n))$ algorithm of Tholey [70] (where α is the inverse Ackermann function).

Our algorithm exploits the characterization of feasible instances of 2-DRP obtained independently by Seymour, Thomassen and Shiloach (see [62, 71, 64]). The algorithm works with an auxiliary graph obtained from the input graph. The characterization implies that if the auxiliary graph contains a K_5-minor then this minor is not "attached" to the set $A = \{s_1, s_2, t_1, t_2\}$ but rather can be "separated" from this set of vertices by a cutset of size at most three. Furthermore, given such a 3-cut it is easy to reduce to a smaller graph in which we add edges to make the cutset a triangle and a component of the auxiliary graph disjoint from A is deleted.

Our algorithm repeatedly applies such reductions. We can think of this algorithm as building a tree decomposition of a graph from which we have excluded an "attached" K_5-minor. The tree we build is a star, with the K_5-minor free graph we have reduced to being at the center of the star, and the leaves containing the pieces we deleted. We call this a pruned K_5-tree decomposition.

The thesis is organized as follows. We begin, in Chapter 2, with an overview of (some) existing tree decompositions and the algorithms which use them. We consider conditions which ensure that a graph has a unique tree decomposition of a given type. We spend considerable time discussing graph minors and the tree decompositions obtained by excluding minors.

In Chapter 3, we present our linear time algorithm for finding a Wagnerian tree-decomposition of a K_5-minor free graph. We first apply classical linear time algorithms

to obtain the block trees and strong 2-block trees of the input graph. These well-known tree decompositions (the algorithmic theory of which was developed by Hopcroft and Tarjan [67, 39]) split the input graph into its 3-connected components using 1-cuts and 2-cuts.

We then apply the key subroutine of our algorithm: a linear time procedure which decomposes the input graph using those 3-cuts whose deletion leaves at least three components, into pieces which have no such 3-cuts. It turns out that this tree decomposition is Wagnerian. This key subroutine is recursive and in each iteration, our algorithm either finds a K_5-minor or recursively applies itself on a set of smaller 3-connected graphs. Having solved the problem on these smaller graphs, it uses the solution to these new problems to construct the solution to the old problem quickly. Our reductions will reduce the problem so much that the entire algorithm runs in linear time provided the reduction and post-recursion subroutines take linear time.

In Chapter 4, we describe our linear time algorithm which builds a pruned K_5-tree of a 3-connected graph with no attached K_5-minor. We begin by reducing the 2-disjoint rooted paths problem to the problem of finding a pruned K_5-tree in a (auxiliary) 3-connected graph. Again, our algorithm is recursive and in each iteration, our algorithm either determines an attached K_5-minor exists and or recursively applies itself on a set of smaller 3-connected graphs. In order to reconstruct a solution for the input graph from the solutions to the smaller graphs we need to exploit the fact that the graph corresponding to the center of the tree decomposition is K_5-minor free. In particular we will show this implies that it has locally bounded tree width. This allows us to apply, and exploit Bodlaender's algorithm [8] for obtaining bounded width tree decompositions of graphs of bounded tree width to carry out the postprocessing in linear time.

Finally, in Chapter 5, we turn to applications. We improve the running time for existing polynomial time approximation schemes on K_5-minor free graphs. This includes all problems described by Baker [5] and Eppstein [26, 27]. Although their algorithm already run in linear

time, we reduce the dependency on the parameter k from super-exponential to exponential. (In fact, building our decomposition only takes $O(kn)$ and it is the dynamic programming step using our decomposition which takes longer.) We then use our algorithm to solve the k-Realizations problem (for fixed k) in linear time on K_5-minor free graphs. Here, we are given a graph G, and a set X of k vertices of G and asked to find all partitions $\Delta = (D_1, \ldots, D_\ell)$ of X such that there is a set of disjoint trees $\{T_1, \ldots, T_\ell\}$ with $D_i \subseteq V(T_i)$. This problem generalizes many problems such as k-disjoint rooted paths and k-vertex disjoint trees.

CHAPTER 2
Tree decompositions and excluded structure

A tree decomposition breaks a graph up into simple pieces using a restricted family of cuts. We have already mentioned some such tree decompositions in the previous section: specifically block trees and the tree decompositions studied by Robertson and Seymour. In this chapter we will discuss these tree decompositions in more detail and present some other examples of tree decompositions. We discuss the relationship between them and conditions ensuring that a graph has a unique tree decomposition.

We will present each tree decomposition in turn, give examples of algorithms for building these decompositions and give examples of using tree decomposition to solve optimization problems.

In the first section, we present the block tree. In the second section, we introduce several trees of 3-connected components. In the third section, we discuss their generalization, $\mathcal{F}$-block tree. We then discuss properties which make a tree decomposition unique. In the fourth section, we classify tree decompositions into three types and show how we can transform a tree decomposition of one type into that of another type. In the fifth section, we introduce clique cutset trees. Finally in the last section we introduce tree decompositions studied by Robertson and Seymour. We present some theorems which tell us that by excluding a substructure in a graph we can ensure it has a tree decomposition of a certain type. We will focus on excluding minors here, given that our main results involve doing so.

2.1 Block trees

The first and perhaps simplest example of a tree decomposition are *block trees*. These trees encode all cutvertices and maximal 2-connected components of a graph, and are well defined for all connected graphs.

Definition 2.1.1. A *block* of a graph G is a maximal 2-connected subgraph of G or a cut-edge of G (or a single vertex if G is a single vertex).

Definition 2.1.2. The *block tree* of a connected graph G is a bipartite graph T with one part indexed by the cutvertices of G and the other part indexed by the blocks of G. A node indexed by a cutvertex v is adjacent to all nodes indexed by blocks containing v (and there are no other edges in T).

See Figure 2–1 for an example of a graph and its block tree. We start by showing that this block tree is indeed a tree and not any bipartite graph.

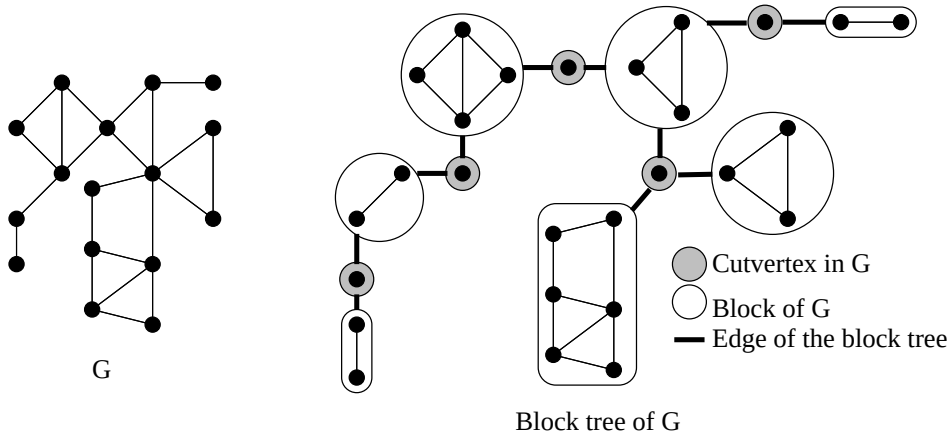


Figure 2–1: An example of a block tree.

To help distinguish between the tree and the graphs indexing its nodes, in the remainder of this thesis, we use "vertices" to refer to vertices of the graph and "nodes" to refer to nodes of the tree.

Theorem 2.1.3. *The block tree of every connected graph is a tree.*

We now present a second recursive definition of the block tree before giving a linear time algorithm to construct one for any connected graph.

From Definition 2.1.2, we see that the block tree of a connected graph is unique (the block tree of a disconnected graph is actually a forest). Not all tree decompositions are unique. This is an important property and we discuss what makes this tree unique later in Section 2.3.

We can instead define block trees recursively, which guarantees it is a tree, and then prove uniqueness.

Definition 2.1.4. 1. If G has no cutvertices, a block tree of G is a single node tree labelled by G.

2. Otherwise, let x be a cutvertex in a connected graph G. Create a node t labelled by $\{x\}$.
3. Obtain the components $U_1, \ldots, U_k$ of $G - x$.
4. Add x back to each component by building the induced subgraphs $C_i = G[U_i \cup \{x\}]$ for $i = 1, \ldots, k$.
5. A block tree of G is obtained from a block tree T_i for each C_i by adding an edge from t to an arbitrary node whose label contains x in each T_i.

A tree obtained recursively in this way can be thought of as a variant of the block tree. Indeed, the final graphs we can no longer recurse on (i.e., those not labelled by a cutvertex) are the blocks of G (this can be shown by induction).

We will come back to the concept of repeatedly breaking a graph on cuts and use similar recursive definitions in the next section. In an even later section, we prove a general theorem that shows this tree is unique and thus, the two definitions are equivalent. We now turn to an algorithm for building a block tree.

2.1.1 Algorithm

We discuss how to build a block tree in linear time in this section.

The first linear time algorithm for building the block tree of a connected graph was developed by Hopcroft and Tarjan [67, 41]. It uses depth-first search (DFS). In fact, many subsequent algorithms for building block trees use DFS (see e.g., [51, 20]).

To perform a depth-first search of a graph,

- we start at an arbitrary vertex and repeat the following,
- move to (and search) an unsearched neighbour (along an edge) if possible, and
- backtrack to the previous vertex if all neighbours have been searched.

Running DFS on a connected graph G gives a natural orientation to the edges by directing them in the first direction in which they are searched. Furthermore, DFS divides the edges of the graph into those that were used in the discovery of a new vertex and those that were not. The first set of edges clearly form a (directed) spanning tree T of G rooted at the starting vertex of DFS. The second set of (directed) edges, called *back edges* (or *fronds* or *cotree edge*), all point from a vertex to one of its ancestors (since it must be from a vertex to a vertex we already visited but not finished visiting at the time). This structure consisting of a directed spanning tree and directed back edges is called a *palm tree* or *trémaux tree*. (The palm tree as a graph is not a tree due to back edges.) This is the only tree that is not a decomposition tree we will see.

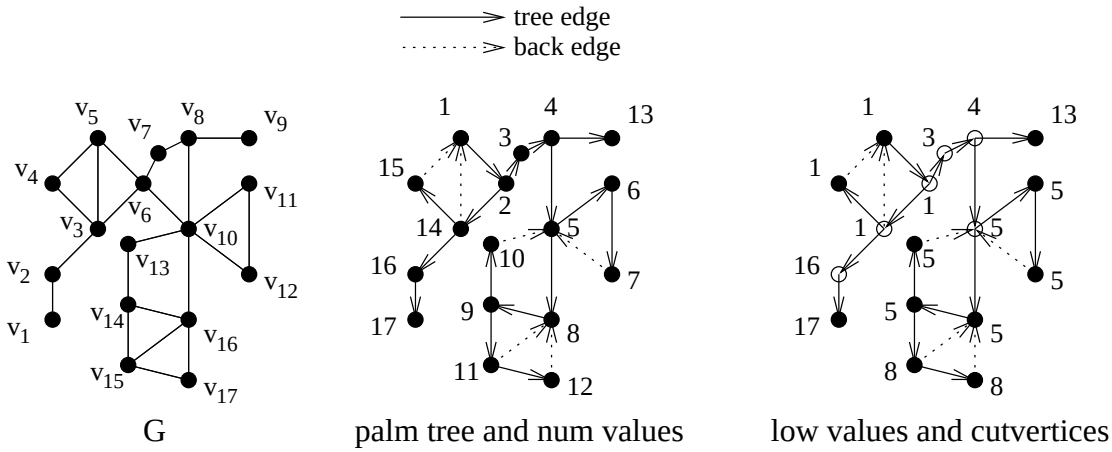


Figure 2–2: An example of a palm tree, numberings num and low, and cutvertices (in white).

The following lemma allows us to easily find all cutvertices in G from this palm tree. From these cutvertices, we can easily find the blocks of G and build its block tree.

Lemma 2.1.5. *Let G be a graph. A non-root vertex v in a palm tree of G is a cutvertex in G if and only if there is a child u of v such that there is no back edge from the tree consisting of u and its descendants to an ancestor of v.*

Proof. Clearly if such a v and child u of v exist then removing v disconnects u from all ancestors of v.

Conversely, if no such child u of v exists then removing v from T leaves all of its ancestors connected and the subtree rooted at each of its children connected. The existence of a back edge from each subtree to an ancestor of v ensures that vertices in all these subtrees are in the same connected component of $G - v$. Since T is spanning, $G - v$ is connected. So v is not a cutvertex. □

This lemma can now be used to find all cutvertices in a connected graph provided we can determine if such a child u exists for each vertex v. This is done by determining for each u, the vertex closest to the root which can be reached (by a back edge) from the subtree rooted at u. Hopcroft and Tarjan [67] formally defines a function low(u) which stores this closest vertex to the root.

They first label the vertices of T (and thus G) in the order they are discovered by DFS using the function num so that ancestors of a vertex have smaller labels (num is sometimes called the *starting time*). Then low can simply be defined as

$$\text{low}[v] = \min(\{\text{num}[w] : xw \text{ is a back edge from some descendant } x \text{ of } v\} \cup \{\text{num}[v]\})$$

and by Lemma 2.1.5, v is a cutvertex if and only if $\text{low}[u] \geq \text{num}[v]$ for some child u of v.

Equivalently,

$$\text{low}[v] = \min(\{\text{low}[u] : u \text{ is a child of } v\} \cup \{\text{num}[w] : vw \text{ is a back edge}\})$$

and we can easily compute low using dynamic programming. This allows us to compute all cutvertices in a connected graph G. Formally, we order the vertices of T in a post-order traversal (or equivalently, we can iterate over the vertices in decreasing values of num) and apply the following algorithm.

Algorithm 2.1.6.

Input: A graph G and a palm tree T of G

Output: low$[v]$ for each $v \in V(G)$

Description.

For v in postorder$(V(T))$,

low$[v] \leftarrow \min(\{\text{low}[u] : u \text{ is a child of } v\} \cup \{\text{num}[w] : vw \text{ is a back edge}\})$

Return low

Again, we can use dynamic programming to obtain the blocks of G by using our list of cutvertices of G. Indeed, any subtree rooted at a cutvertex containing no other cutvertex is a block. Otherwise, if the subtree contains other cutvertices, we need to exclude the subtree rooted at each of these other cutvertices from the block.

More formally, we can describe the algorithms for finding all blocks as follows.

Algorithm 2.1.7.

Input: A graph G, a palm tree T of G, and a list of cutvertices of G

Output: The blocks of G

Description.

Initialize the list of blocks to be empty

For v in postorder($V(T)$),

If v is not a cutvertex,

currentblock(v) $\leftarrow \{v\} \cup \bigcup_{u:u \text{ is a child of } v}$ currentblock(u)

If v is a cutvertex,

newblock $\leftarrow \{v\} \cup \bigcup_{u:u \text{ is a child of } v}$ currentblock(u)

add (the vertices induced by) newblock to the list of blocks

currentblock(v) = $\{v\}$

Return the list of blocks

Note that we can take unions in the above algorithm efficiently since we no longer need the original sets when joining them (e.g., we could use doubly linked lists and join them together).

Now given the cutvertices of G and the blocks of G, we can easily build its block tree as follows.

1. For each cutvertex v, create a node t with $V(G_t) = \{v\}$ (and no edges in G_t).
2. For each block B, create a node t with $G_t = B$.
3. For each block B, add an edge $t_B t_v$ if $v \in B$.

We note that Tarjan (see [67], page 153) gave a single pass algorithm which computes all of the above: num, low, cutvertices and blocks.

2.2 Tree of 3-connected components

The result of the previous section may make us hopeful that similar trees exist for higher connectivities. However, further complications already arise if we wish to define a tree of 3-connected "blocks" of a 2-connected graph.

For example, consider a cycle G. Any two non-adjacent vertices of G form a 2-cut but G contains no (non-trivial) 3-connected subgraphs. So simply defining the "tree" of 3-connected components as an incidence graph would yield a stable set of cuts in this case.

What if instead we try to use the recursive approach as in Definition 2.1.4? In this case, we notice that breaking G on any 2-cut X yields two components U_1, U_2 and adding X back to each gives us two paths C_1, C_2. But these paths are not 2-connected so it is not clear what we should do recursively. To make matters worse, some 2-cuts present in G are no longer completely contained in either C_1 or C_2.

Although, it is possible to obtain a definition as the incidence structure of some cutsets of size 2 and subgraphs of G with edges added to make them 2-connected, it is not immediately clear how to do so. We will thus first give a recursive definition of the tree of 3-connected components of G. We will come back to a definition using "incidences" later on.

We will discuss in more detail why these problems occur and some general ways to fix them in Section 2.3. For the moment, we simply provide definitions which avoid all these problems.

We first follow Tutte's [72] and Hopcroft and Tarjan's [39] treatment of these trees of 3-connected components (but using different terminology) and then discuss variants of these trees.

For Tutte's, Hopcroft and Tarjan's tree and later, the SPQR tree, we allow multigraphs. For all other tree decompositions, we restrict ourselves to simple graphs.

First, we make the following simple modification to our recursive approach by adding an edge between the vertices of the cut whenever we recurse. More precisely, we define the operation of *decomposing on a cut*, used throughout this thesis.

Definition 2.2.1. Let G be a graph and X a cutset in G. By *decomposing on* X, we mean to build $C_1, \ldots, C_k$ as follows.

Build the components $U_1, \ldots, U_k$ of $G - X$. For each $i = 1, \ldots, k$, build X_i, the subset of X with neighbours in U_i and then build C_i the graph obtained from $G[U_i \cup X_i]$ by adding all edges between vertices of X_i (i.e., we add a clique on X_i).

We refer to $C_1, \ldots, C_k$ as the *decomponents* of $G - X$.

We refer to graphs obtained by repeatedly decomposing a graph G on 2-cuts as the *3-connected components* of G. We state this definition more formally.

Definition 2.2.2. Suppose we apply the following procedure to a 2-connected graph G.

1. Find a 2-cut X in G (if it exists) and decompose G on X,
2. recursively apply these steps to each decomponent C_i (until no 2-cuts are found).

We call the resulting graphs *3-connected components* of G.

This ensures that each intermediate graph is 2-connected and each final graph is 3-connected. This leads us to the following definition of a 2-block tree.

Definition 2.2.3. Let G be a 2-connected graph.

A *2-block tree of* G, written $[T, \mathcal{G}]$, is a tree T with a set of graphs $\mathcal{G} = \{G_t\}_{t \in T}$ with the following properties.

- T contains two types of nodes which we refer to as "graph nodes" and "cutting nodes".
- If G contains no 2-cuts then T has a single graph node r and $G_r = G$.
- If G has a 2-cut then there exists a cutting node $t \in T$ such that
  1. $V(G_t) = \{x, y\} = X$, $E(G_t) = \{xy\}$ and X is a 2-cut.
  2. Let $T_1, \ldots, T_k$ be the components (subtrees) of $T - t$. Then $G - X$ has k decomponents $C_1, \ldots, C_k$ such that T_i is a 2-block tree of C_i.
  3. For each i, there is a graph node $t_i \in T_i$ such that $X \subseteq V(G_{t_i})$ and $tt_i \in E(T)$. Furthermore, t has not other edges to T_i.

For $t \in V(T)$, we refer to G_t as the *label* or *index* of t.

There could be many 2-block trees for a graph (for example, every triangulation of a cycle corresponds to a 2-block tree). To guarantee uniqueness of the final graphs (regardless

of the cuts chosen at the intermediate steps), we restrict ourselves to finding 2-cuts X for which there exist three vertex disjoint paths between the vertices of X in G We call these cuts *strong 2-cuts*.

Definition 2.2.4. Let G be a 2-connected graph. A 2-cut X is a *strong 2-cut* if there exist three vertex disjoint paths between the vertices of X in G.

Definition 2.2.5. Suppose we apply the following procedure to a 2-connected graph G.

1. Find a strong 2-cut X (if it exists) and decompose on X,
2. recursively apply these steps to each decomponent C_i (until no strong 2-cuts are found).

Then the resulting graphs are called the *triconnected components* of G.

We can now define the strong 2-block tree, a tree of triconnected components for a 2-connected graph G. Note that triconnected components of a graph may not be 3-connected (as opposed to 3-connected components). For example, the only triconnected component of a cycle C is the cycle as no 2-cut of C is strong. However, cycles and edges are the only triconnected components (of any 2-connected graph) which are not 3-connected.

Definition 2.2.6. Let G be a 2-connected graph. A *strong 2-block tree of* G, written $[T, \mathcal{G}]$, is a tree T with a set of graphs $\mathcal{G} = \{G_t\}_{t \in T}$ with the following properties.

- T contains two types of nodes which we refer to as "graph nodes" and "cutting nodes".
- If G contains no strong 2-cuts then T has a single graph node r and $G_r = G$.
- If G has a strong 2-cut then there exists a cutting node $t \in T$ such that
  1. $V(G_t) = \{x, y\} = X$, $E(G_t) = \{xy\}$ and X is a strong 2-cut.
  2. Let $T_1, \ldots, T_k$ be the components (subtrees) of $T - t$. Then $G - X$ has k decomponents $C_1, \ldots, C_k$ such that T_i is a strong 2-block tree of C_i.
  3. For each i, there is a graph node $t_i \in T_i$ such that $X \subseteq V(G_{t_i})$ and $tt_i \in E(T)$. Furthermore, t has not other edges to T_i.

For $t \in V(T)$, we refer to G_t as the *label* or *index* of t.

It is known that all graph nodes of a strong 2-block tree are either 3-connected graphs, cycles or single edges. We will not prove this here (see [39]).

2.2.1 Algorithms

Hopcroft and Tarjan [39] gave the first linear time algorithm for constructing the strong 2-block tree of a 2-connected multigraph G. Their algorithm is similar to the algorithm described in Section 2.1.1 but is much more involved (in fact, a later correction was necessary [34]).

It uses a two pass DFS. On the first pass, they label the vertices of the input graph G in the order they are discovered and on the second pass, this labelling is used to break ties when choosing a new vertex to visit. This allows them to consider the edges of G in an ordering which does not require backtracking, thus allowing their algorithm to run in linear time. They also need to define and compute num, low and other values on the vertices of G and use properties of the palm tree (created by the second pass DFS).

They first split G into 3-connected components and then merge some of these components to obtains the triconnected components of G. The tree structure on these components is implicitly encoded in the edges added between vertices of the cut (which they refer to as *virtual edges*), thus explaining the need for multigraphs and multi-edges.

2.2.2 Other variants of the tree of triconnected components

Tutte [72] first introduced a tree of triconnected components as the tree of *cleaved components* without algorithmic considerations. Hopcroft and Tarjan [39] refer to the components as *split components* or triconnected components and gave the first linear time algorithm for finding them. Di Battista and Tamassia [24] then formalized the tree implicit in Hopcroft and Tarjan's algorithm into what they call an *SPQR tree*. In particular, this allows Di Battista and Tamassia to design an algorithm to maintain the triconnected components of a graph while it is modified online [23].

Definition 2.2.7. The *SPQR tree* of a 2-connected graph G is a tree T with nodes labelled by triconnected components of G (i.e., subgraphs of G with some added edges). The tree is comprised of four types of nodes, named S, P, Q and R. Q nodes are only used when the graph consists of a single edge (in which case its tree contains a single Q node labelled by G).

In all other cases,

- S nodes are *series* nodes, labelled by a cycle,
- P nodes are *parallel* nodes, labelled by two vertices with at least 3 edges between them,
- R nodes are *rigid* nodes, labelled by 3-connected graphs.

The strong 2-cuts in the graph are implicitly encoded by the intersection of adjacent labels in the SPQR tree T (aside from the added edges which obviously still each correspond to a 2-cut). Thus, from an SPQR tree, we can obtain the incidence structure of triconnected components of G and strong 2-cuts.

We finish off this section by presenting a second non-recursive definition of *strong 2-block trees* (similar to Definition 2.1.2 for block trees). We note that at first it may seem unusual to consider multigraphs in a problem primarily concerned with vertex connectivity. However, we see that multi-edges are necessary for remembering the 2-cuts, for example in the case of Hopcroft and Tarjan's algorithm.

However, we could instead mimic block trees and build an incidence graph of strong 2-cut and triconnected components, thereby encoding the 2-cuts of G in the labels of the tree rather than the added edges. Hence, we can define the following tree.

Definition 2.2.8. The *strong 2-block tree* of a 2-connected graph G is a bipartite graph T with one part indexed by the strong 2-cuts of G and the other part indexed by the triconnected components of G. A node indexed by a strong 2-cut X is adjacent to all nodes indexed by triconnected components completely containing X (and there are no other edges in T).

2.2.3 Application: Planarity Testing

Using algorithms for building block trees and a tree of triconnected components, we can obtain an algorithm which tests if a graph is planar.

Given a graph G, we build its components (using DFS, say), then the blocks of each component and the triconnected components of each block. Finally, we test the planarity of each triconnected component to decide the planarity of G.

This algorithm relies on the following facts about *embeddings* of a planar graph. An *embedding* of a planar graph can be thought of as a drawing of the graph in the plane with no crossing edges. However, just as with graphs, we are not concerned with the "spacial location" of points and curves. Such drawings can be summarized using *combinatorial embeddings* which store the order of the edges around each vertex.

Definition 2.2.9. A *combinatorial embedding* of a planar graph G is a cyclic ordering of the neighbours of every vertex in G. I.e., $\{L_v | v \in V(G)\}$ where L_v is ordering of $N(v)$ up to cyclic permutation.

Theorem 2.2.10 (Whitney's theorem)**.** *[76] Every 3-connected planar graph has a unique combinatorial embedding.*

Theorem 2.2.11. *[50] A graph G is planar if and only if all of its triconnected components are planar.*

We do not prove Whitney's theorem and give an algorithmic proof of one direction of Theorem 2.2.11 in Section 2.2.3.

Algorithm

We describe a simple (non linear time) algorithm due to Bruno et al. [14] which tests if a triconnected graph is planar. We avoid the description of a linear time algorithm due to Hopcroft and Tarjan [40]. The reason we present Bruno et al.'s algorithm is that it has the same flavor as our algorithm. It recurses to smaller graphs by repeatedly deleting or contracting edges and then postprocesses by undoing these operations.

It is easy to test if the input is either a cycle or a single edge (the other possible triconnected components) and they are always planar. So, we can restrict our attention to 3-connected graphs.

The key to their algorithm relies on the following result of Tutte on 3-connected graphs.

Definition 2.2.12. A *wheel* W_n on n vertices is the graph obtained from the cycle C_n by adding a vertex adjacent to all of C_n.

Definition 2.2.13. By *contracting an edge* $e = uv$ in a graph, we mean to remove all edges between u and v and replace the vertices u, v with a single vertex w_e incident to all edges originally incident to either u or v.

We refer to this operation as *edge contraction*. We denote the resulting graph by G/uv (or G/e).

Theorem 2.2.14. *[72] Every 3-connected graph either*

1. *is a wheel,*
2. *contains an edge whose deletion yields a 3-connected graph, or*
3. *contains an edge whose contraction yields a 3-connected graph.*

We do not prove this theorem here. We remark that only vertex connectivity is of interest to us in this particular case so we can keep multiple edges when contracting. This will help later on in embedding the graph. We use the following corollary of the theorem.

Corollary 2.2.15. *For every 3-connected graph G, there is a sequence of 3-connected graphs $G = G_1, \ldots, G_k$ where G_k is a wheel and every graph G_i (except G_1) is obtained from the previous graph G_{i-1} by deleting an edge or contracting an edge.*

With this corollary in hand, describing the algorithm is straightforward. Given an input 3-connected graph G, we find a sequence as in Corollary 2.2.15. To determine if G is planar, start from the wheel at the end of the sequence (which is planar) and iteratively undelete an edge or uncontract an edge. Due to Whitney's theorem and the fact that all graphs

in the sequence are planar, there is only one way to preserve planarity while undeleting or uncontracting an edge.

Formally, we have the following.

Lemma 2.2.16. *Let H be a 3-connected planar graph with $uv \notin E(H)$. Then $H + uv$ is planar if and only if both endpoints of uv lie on a face of H.*

Lemma 2.2.17. *Let G and H be 3-connected planar graphs with $G = H/v_Lv_R$. Let $v \in V(H)$ be the vertex v_Lv_R contracts to. Then G is planar if and only if the edges incident to v_L are in a consecutive clockwise order around v in the (unique) embedding of H.*

We now give a formal description of the algorithm presented in this section.

Algorithm 2.2.18.

Input: A 3-connected graph G.

Output: A planar embedding of G if G is planar and "G is not planar" otherwise

Description.

Build the sequence $G = G_0, \ldots, G_k$ where G_k is a wheel, all G_i are 3-connected and consecutive graphs in the sequence differ by an edge deletion or contraction.

Obtain a planar embedding of G_k

For $i = k, \ldots, 1$,

If $G_i = G_{i-1} - uv$ then

Check if u and v lie in the same face of the embedding of G_i.

If they do, add uv to G_i and update the embedding of G_i to obtain an embedding of G_{i-1}

Otherwise, return "G is not planar".

If $G_i = G_{i-1}/v_Lv_R$ then

Check if the edges incident to v_L are in a consecutive (clockwise) order around v, the contraction of v_Lv_R

If it is, uncontract v and update the embedding of G_i to obtain an embedding of G_{i-1}.

Otherwise, return "G is not planar".

We will make use of this concept of uncontraction seen here in later sections, both in planar and non-planar settings.

Combining embeddings

Now suppose we also want to obtain an embedding of the input graph G when it is planar. The last section already gives us the embedding of the triconnected components of G. We merge them using a tree of these triconnected components to obtain an embedding for each block of G. We then merge the embedding for the blocks using the block tree of G. We now provide the details of this merging operation.

Using combinatorial embeddings, we see that it is easy to merge two embeddings for two decomponents of $G - v$ (for a cutvertex v) as we can simply concatenate the two lists for v in each embedding (and take the union of the remaining lists without changing them). If we use pointers and linked lists, we can merge in $O(1)$ time.

To merge two embeddings L_1, L_2 for two decomponents of $G - \{u, v\}$, again we combine the lists for u and v. We shift $L_{2,v}$ so u is at the beginning, delete u in $L_{1,v}$ and replace it by $L_{2,v}$ (at the same location in the list). We shift $L_{2,u}$ so v is at the beginning, delete v in $L_{1,u}$ and replace it by $L_{2,u}$ (at the same location in the list).

We can also easily delete an edge uv from such an embedding by simply removing u from L_v and v from L_u. This again allows us to merge the embeddings of two decomponents of $G - X$ where X has size 2.

This completes our algorithm for obtaining an embedding of any planar graph G (in polynomial, but not linear time). We now turn to combining decomponents of cutsets of size 3. Although not needed here, it will be useful in later sections.

Combining embeddings on triangles

We can extend the algorithm discussed in the previous section for combining embeddings on vertices and edges to combining planar embeddings on triangles as well. Here we wish

to paste together (identify) two triangles in two different planar embeddings (which can be thought of as the decomponents of $G - X$ for some graph G and $|X| = 3$).

We only consider the case where both graphs we want to paste together are 3-connected.

Instead of combining combinatorial embeddings, we combine combinatorial embeddings of the duals. In other words, we keep track of the cyclically ordered list of edges on each face.

To be able to combine embeddings on a triangle in $O(1)$, we need to keep track of more information. At each edge e, we store pointers to the 2 faces $F(e)$ containing e.

Definition 2.2.19. An *augmented combinatorial embedding* of a planar graph G is a cyclic ordering of the edges of every face in G and a pointer $F(e)$ from each edge to the two faces containing e. I.e., $\{L(f)|f \in F(G)\}$ where $L(f)$ is a closed walk of the boundary of f and $\{F(e)|e \in E(G)\}$.

Note that every face of a 2-connected (and thus 3-connected) planar graph is bounded by a cycle and therefore no edge appears twice in the same face.

We also note that our algorithm does not check whether the newly combined embedding is actually planar. This allows us to later combine multiple embeddings (as well as perform other operations) without having to check planarity of the intermediate embeddings (and only check at the end). We still refer the intermediate outputs as augmented combinatorial embeddings even though they may be for non-planar graphs.

However, if the resulting graph is planar, we do obtain an augmented combinatorial embedding of the resulting graph.

Algorithm 2.2.20.

Input:

- Graphs G_1, G_2,
- a triangular face f_1 with edges u_1v_1, v_1w_1, w_1u_1 in G_1,
- a triangular face f_2 with edges u_2v_2, v_2w_2, w_2u_2 in G_2, and

- an augmented combinatorial embedding L_1, F_1 of G_1 and an augmented combinatorial embedding L_2, F_2 of G_2.

Output: An augmented combinatorial embedding L, F of H obtained from G_1, G_2 by identifying u_1 with u_2 into u, v_1 with v_2 into v and w_1 with w_2 into w.

Description and analysis.

1. Set L to be the disjoint unions of the two embeddings (i.e., $L(f) = L_1(f)$ if f is in G_1 and $L(f) = L_2(f)$ if f is in G_2).
2. Remove $L(f_1)$ and $L(f_2)$.
3. Set $F(e) = (F_1(e) \cup F_2(e)) - \{f_1, f_2\}$ if $e \in \{uv, vw, wu\}$
   Set $F(e) = F_1(e)$ if e has at least one endpoint in $V(G_1) - V(G_2)$
   Set $F(e) = F_2(e)$ if e has at least one endpoint in $V(G_2) - V(G_1)$
4. Return L, F.

□

Note that if f_1 is not given (but u_1, v_1, w_1 is), we can compute it by taking the intersection of $F(u_1v_1)$, $F(v_1w_1)$ and $F(w_1u_1)$.

If the intersection is empty then no face contains u_1, v_1 and w_1. In the case where G_1 and G_2 are 3-connected, this mean the resulting graph G is non-planar. (In fact, in this case, we are able to obtain a certificate for the non-planarity of G.)

To extend our algorithm to decomponents, we need to remove some edges $\{u_1v_1, v_1w_1, w_1u_1\}$. However, we cannot remove edges and update the embedding in $O(1)$. (To delete an edge in $O(n)$, we can also easily remove an edge e from an augmented combinatorial embedding of a 3-connected graph G by replacing both faces f_1, f_2 in $F(e)$ by a single face f with $L(f) = (L(f_1) - e) \cup (L(f_2) - e)$, removing e from F and update $F(e')$ for every $e' \in (L(f_1) \cup L(f_2)) - e$.

For our application in later chapters, we need to repeatedly combine many graphs on triangles. We do not need to delete extra edges at the intermediate steps and just as we do not check intermediate graphs for planarity, we only delete these edges at the very end.

Linear time planarity testing algorithms

Although we presented a non-linear time algorithm, linear time algorithms to test if a graph G is planar and which produce an embedding (usually a combinatorial embedding) of G are well known. In fact, these algorithms also produce a certificate in case G is non-planar (a K_5-minor or $K_{3,3}$-minor; we discuss these later in Section 2.6.5, Theorem 2.6.25).

The first such algorithm was discovered by Hopcroft and Tarjan [40] and use ideas from their algorithm for building the tree of split components (a variant of the strong 2-block tree) [39, 34]. Their algorithm again uses a two pass DFS (and numberings of the vertices). It splits the graph along a cycle and considers the resulting components and their adjacencies to this cycle separately. Di Battista and Tamassia introduced SPQR trees to maintain planar embeddings through a sequence of vertex and edge additions [23].

Simplifications to this algorithm were later suggested by De Fraysseix et al. [20] which avoid the use of SPQR trees. Their "left-right" algorithm is non-recursive and includes an efficient implementation [19] which is freely available.

A different simplification was suggested [48, 63, 12] which orders the vertices of G so that for any k, both the first k vertices and the last k vertices induce a connected graph (the similarity between these three papers was first noted by Haeupler and Tarjan [35]). Such an ordering, sometimes called an *st* numbering, allows successive additions of vertices to an initially empty embedding while preserving the connectivity of the embedded portion of the graph as well as the connectivity of the graph induced by the remaining vertices (where vertices are added in this order).

For further details and a graph theoretical treatment of these planarity testing algorithms, we refer to the reader to Chapter 1 of [66].

2.3 Laminar cutsets and component minimality

The uniqueness of the block tree (defined recursively) can be attributed to two reasons.

1. First, when we decompose on a cutvertex, no new cutvertices are created (present in a decomponent but not the original graph).
2. Second, all cutvertices of the original graph are still cutvertices in some decomponent and thus we will decompose on that cutvertex eventually.

We wish to define and use other trees of cuts obtained by decomposing on different cutsets (for example, 2-cuts and strong 2-cuts from the previous section). To do so, we determine what properties (which cutvertices of a graph do satisfy) we must impose on a type of cut to ensure the following two properties hold.

U1 **(No new cuts)** Let X be any cut (of type of interest). If Y is a cut (of type of interest) in a decomponent of $G - X$ then Y is also a cut (of type of interest) in G.

U2 **(No lost cuts)** Let X be any cut (of type of interest). If $Y \neq X$ is a cut (of type of interest) in G then Y is also a cut (of type of interest) in some decomponent of $G - X$.

We show that if X is a cutset in a decomponent D then it is also a cutset in G. This implies that for many natural types of cuts, (U1) holds automatically.

For the second property to hold, we will need extra conditions on the type of cuts of interest. We introduce the notions of *laminarity* and *component minimality* and show they are sufficient to guarantee the second property.

Definition 2.3.1. We call a cut of size i that splits G into at least j components an (i, j)-*cut*.

Lemma 2.3.2. *If we decompose a graph G on a cut X and $Y \neq X$ is a cut in a decomponent C of $G - X$, then $G - Y$ has at least as many components as $C - Y$. In particular, if Y is an (i, j)-cut of C then it is an (i, j)-cut of G.*

Proof. The second statement follows from the first. A component of $C - Y$ which is disjoint from X is also a component of $G - Y$. At most one component of $C - Y$ contains any vertex of X since X is a clique in C. If such a component exists then $X - Y$ is non-empty and

some component of $G - Y$ contains a vertex of $X - Y$. Therefore, $G - Y$ has at least as many components as $C - Y$. □

The converse (that is property (U2)) does not always hold. For example, decomposing on one part of the bipartition of $K_{3,3}$ yields three K_4's which prevents us from decomposing on the other side of the bipartition of $K_{3,3}$ (as the vertices are now in different decomponents).

Having vertices of Y in different decomponents of $G - X$ is only one of three ways to "lose" the cut Y. We can also lose Y if decomposing on X joins all components of $G-Y$ (due to the clique we added on X). In this case, X is in different components of $G - Y$ (in fact, X intersects all components of $G - Y$) so the vertices of X are in different decomponents of $G - Y$ (i.e., we would lose X in the first way if we decompose on Y). Finally, we can "lose" Y if decomposing on X puts Y in a decomponent of $G - X$ where only one component of $G - Y$ is present (i.e., X cut off all components of $G - Y$ already).

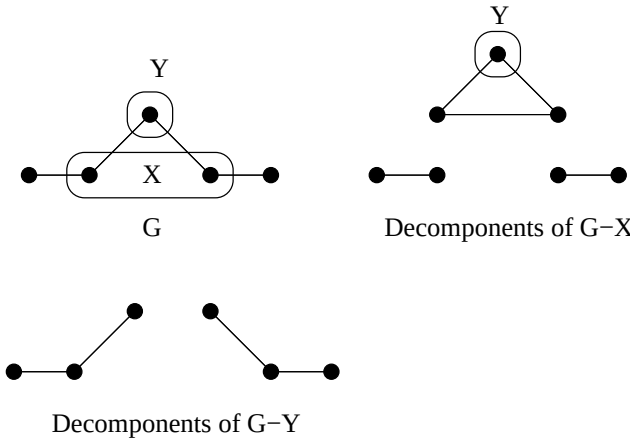


Figure 2–3: Two cutsets X in Y in P_5 and their decomponents. X is in different decomponents of $G - Y$ and Y is no longer a cut in the decomponent of $G - X$ containing Y. This is an example of the first two ways of losing a cut.

To prevent the first two kinds of losses, we can require cuts to be *laminar*.

Definition 2.3.3. Let X and Y be two cuts of G. If X intersects at most one component of $G - Y$ and Y intersects at most one component of $G - X$ then we say that X and Y are *laminar*.

A family of cutsets $\mathcal{F}$ is said to be *laminar* if every pair of cutsets in $\mathcal{F}$ are laminar.

To address the third kind of loss, we could require cuts to be minimal. We instead relax this condition a bit and only require cuts to be *component minimal*.

Definition 2.3.4. We say that a cut X is *component minimal* if there exist components U_1, U_2 of $G - X$ such that for every subset Y of X, $V(U_1)$ and $V(U_2)$ are in the same component of $G - Y$.

U_1, U_2 are the *components certifying* X is component minimal.

The reason we choose the component minimal property rather than the negation of the second kind of problem (i.e., require decomposing on one cut to not decrease the number of components in a different cut) as we did with laminarity is that a cut remains component minimal (and laminar) when we decompose on other cuts (whereas not decreasing the number of components of other cuts may not be preserved).

It is now easy to see that laminar component minimal cuts are not lost.

Lemma 2.3.5. *Let G be a graph and X, Y be two laminar cuts in G with $Y \not\subseteq X$ such that X, Y are component minimal. Then there is a unique decomponent C of $G - X$ that contains all vertices of Y. Furthermore, Y is a cut of C.*

Proof. A decomponent C of $G - X$ containing Y exists by the definition of laminarity. It is unique because Y is not a subset of X. We now show Y is a cut in C.

Let U_1, U_2 be components certifying X is component minimal and label them so that U_1 is not $C - X$. U_1 has edges to all vertices of X and so only one component of $G - Y$ contains vertices of X (and U_1). Thus, there is at least one component of $G - Y$ not containing X and this is also a component of $C - Y$. Since $C - Y$ has at least two components (one containing X and one not containing X), Y is a cut in $C - Y$.

□

We now formalize the definition of "cut type".

Definition 2.3.6. A *cut property* P is a boolean function that takes as input a graph G and a subset X of the vertices of G and outputs whether X is a cut with property P in G.

A cut in G with property P is called a *P-cut*.

1-cuts in connected graphs and strong 2-cuts in 2-connected graphs are minimal. We close this section by showing they are also laminar, thereby proving the uniqueness of block trees and strong 2-block trees.

1-cuts are trivially laminar as it is not possible to separate a set of size 1.

Lemma 2.3.7. *All strong 2-cuts in a 2-connected graph are laminar.*

Proof. Suppose X and Y are strong 2-cuts. Since there exist three vertex disjoint paths between the vertices of X, Y misses at least one of these paths so Y does not separate X. By symmetry, X does not separate Y. □

We now define a more general tree decomposition in terms of a family $\mathcal{F}$ of cuts of the initial graph G.

2.3.1 $\mathcal{F}$-block trees

In this section, we generalize block trees and 2-block trees to arbitrary cuts in a graph G. We define such trees in terms of a cut property P.

Definition 2.3.8. Let G be a graph and P a cut property. A *P-block tree of G*, written $[T, \mathcal{G}]$, is a tree T with a set of graphs $\mathcal{G} = \{G_t\}_{t \in T}$ with the following properties.

- T contains two types of nodes which we refer to as "graph nodes" and "cutting nodes".
- If G contains no P-cut then T has a single graph node r and $G_r = G$.
- If G has a P-cut then there exists a cutting node $t \in T$ such that
  1. $V(G_t) = X$, $E(G_t) = \{xy | x, y \in X\}$ and X is P-cut in G.
  2. Let $T_1, \ldots, T_k$ be the components (subtrees) of $T - t$. Then $G - X$ has k components $C_1, \ldots, C_k$ such that T_i is a P-block tree of C_i.

3. For each i, there is a graph node $t_i \in T_i$ such that $X \subseteq V(G_{t_i})$ and $tt_i \in E(T)$. Furthermore, t has not other edges to T_i.

We can also define a similar tree in terms of a family $\mathcal{F}$ of cutsets (rather than a property). We use this definition more oftenly.

Definition 2.3.9. Let G be a graph and $\mathcal{F}$ a family of cutsets in G. A *$\mathcal{F}$-block tree of G*, written $[T, \mathcal{G}]$, is a tree T with a set of graphs $\mathcal{G} = \{G_t\}_{t \in T}$ with the following properties.

1. T contains two types of nodes which we refer to as "graph nodes" and "cutting nodes".
2. If G contains no cut in $\mathcal{F}$ then T has a single graph node r and $G_r = G$.
3. If G has a cut in $\mathcal{F}$ then there exists a cutting node $t \in T$ such that
   (a) $V(G_t) = X$, $E(G_t) = \{xy | x, y \in X\}$ and X is a cut of $\mathcal{F}$ in G.
   (b) Let $T_1, \ldots, T_k$ be the components (subtrees) of $T - t$. Then $G - X$ has k decomponents $C_1, \ldots, C_k$ such that T_i is an $\mathcal{F}$-block tree of C_i.
   (c) For each i, there is a graph node $t_i \in T_i$ such that $X \subseteq V(G_{t_i})$ and $tt_i \in E(T)$. Furthermore, t has not other edges to T_i.

For both definitions 2.3.8 and 2.3.9, we refer to G_t as the *label* or *index* of t (for each $t \in V(T)$). These two definitions relate in the following way.

Remark 2.3.10. *For a laminar family of component minimal cuts, an $\mathcal{F}$-block tree is a P-block tree where X is a P-cut if and only if X belongs to $\mathcal{F}$ (i.e., $P(G, X)$ is true if $X \in \mathcal{F}$ and false otherwise).*

We now prove the uniqueness of cutting nodes of a family of $\mathcal{F}$-block trees.

Lemma 2.3.11. *Let $\mathcal{F}$ be a family of laminar component minimal cuts in a graph G. Then in any $\mathcal{F}$-block tree of G obtained by decomposing on a cut of $\mathcal{F}$ that is not a superset of any remaining cut of $\mathcal{F}$ at each step, all cuts in $\mathcal{F}$ appear as the index of exactly one cutting node.*

Proof. We proceed by induction on the size of $V(G)$. If G has no cut in $\mathcal{F}$, then its only $\mathcal{F}$-block tree has precisely one graph node, and no cutting nodes and we are done.

Otherwise, for any $\mathcal{F}$-block tree $[T, \mathcal{G}]$ there is a cut $X \in \mathcal{F}$ which corresponds to a cutting node t of the tree and such that deleting it decomposes the tree into subtrees which are $\mathcal{F}$-block trees for the decomponents of $G - X$.

By Lemma 2.3.2 and 2.3.5, all cuts of $\mathcal{F}$ except X appear in some decomponent. By the inductive hypothesis, all cuts of $\mathcal{F}$ completely contained in a decomponent of $G - X$ appear exactly once in the corresponding subtree of T. Therefore, each cut of $\mathcal{F}$ is the label to exactly one cutting node of $[T, \mathcal{G}]$. □

As a corollary, we obtain the same lemma for minimal cuts.

Lemma 2.3.12. *Let $\mathcal{F}$ be a family of laminar minimal cuts in a graph G. Then in any $\mathcal{F}$-block tree of G obtained by decomposing on a cut of $\mathcal{F}$, all cuts in $\mathcal{F}$ appear as the index of exactly one cutting node.*

2.3.2 Uniqueness of $\mathcal{F}$-block tree

Aside from the unique multiset of cutting nodes, we can also get a unique multiset of graph nodes and finally, isomorphism between all $\mathcal{F}$-block trees. However, component minimal and laminarity of cuts is not sufficient (see Figure 2–4) and in most cases, unique multiset of cutting nodes will suffice.

We first need to define equivalence between two $\mathcal{F}$-block trees.

Definition 2.3.13. We say two $\mathcal{F}$-block trees $[T_1, \mathcal{G}_1]$ and $[T_2, \mathcal{G}_2]$ of G are *equivalent* (or the *same* tree) if there is an isomorphism f from T_1 to T_2 such that for each $t \in V(T_1)$, $G_{1,t}$ is isomorphic to $G_{2,f(t)}$ (as graphs labelled by vertices of G).

From the definition of $\mathcal{F}$-block tree, we see there are two ways to get different trees from the same graph G.

1. First, at step 3, there may be a different cuts in $\mathcal{F}$ we can choose to decompose on.
2. Second, at step 3c, once we obtain the $\mathcal{F}$-block tree for decomponents recursively, we can choose which graph nodes to connect t to.

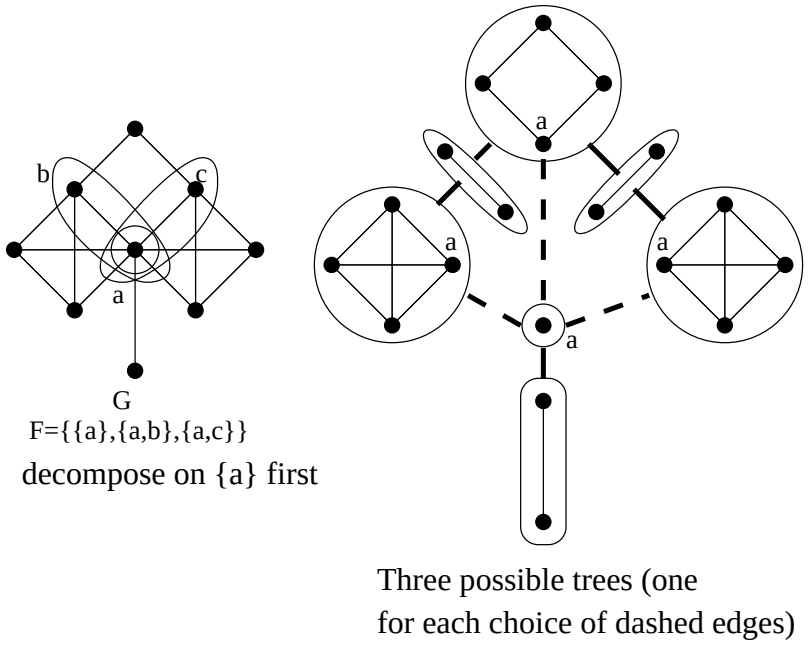


Figure 2–4: Three component minimal cuts $\mathcal{F} = \{\{a\}, \{a, b\}, \{a, c\}\}$ in a graph and three of the possible $\mathcal{F}$-block trees.

There is only one way to complete this second step when $\mathcal{F}$ is a family of minimum cuts.

Lemma 2.3.14. *Let G be a k-connected graph, $\mathcal{F}$ a family of laminar k-cuts in G and $[T, \mathcal{G}]$ an $\mathcal{F}$-block tree of G. Then any clique X of size k in G not indexing a cutting node appears in a unique graph node of T.*

Proof. Suppose X is in two graph nodes. Then X also appears on the path between these two graph nodes. But this path includes at least one cutting node since T alternates between cutting nodes and graph nodes. So X is contained in the index G_t of such a cutting node. Since both X and the index have size k, $X = G_t$, a contradiction. □

In the case that $\mathcal{F}$ is a laminar family of component minimal cuts, we can think of building an $\mathcal{F}$-block tree by first selecting a permutation of the cuts in $\mathcal{F}$ and decomposing on cuts in $\mathcal{F}$ in this order (rather than using the recursive definition given in Definition

2.3.9). We use this remark to show that the $\mathcal{F}$-block tree where $\mathcal{F}$ is a laminar family of minimum cuts is unique.

Theorem 2.3.15. *Let G be a k-connected graph, $\mathcal{F}$ a family of laminar k-cuts in G. Then G has a unique $\mathcal{F}$-block tree.*

Proof. We proceed by induction on $|\mathcal{F}|$. We can assume G has a cut in $\mathcal{F}$. We need only show that every two permutations of the cuts in $\mathcal{F}$ (representing the order in which we decompose on cuts) yield the same tree. Since we can move between any two permutations by repeatedly transposing adjacent elements, it is enough to show that if π_1 is a permutation and π_2 is obtained from π_1 by transposing two adjacent elements then π_1 and π_2 yield the same tree.

If π_1 and π_2 both have first element X, then the corresponding trees both have a cutting node t corresponding to X such that deleting t breaks the tree up into subtrees corresponding to the graphs obtained when we decompose G on X. By the inductive hypothesis, these subtrees are unique, and by Lemma 2.3.14, there is only one way to combine them on a cutting node corresponding to X. So, we see that the two permutations do indeed yield the same tree.

Otherwise, there exist two cuts X, Y such that X is immediately before Y in π_1 while Y is immediately before X in π_2. From the definition of the tree corresponding to a permutation we see that for both permutations, if we delete the nodes corresponding to X and Y then we obtain (i) one subtree corresponding to all the components of $G-X-Y$ which have an edge to both X and Y (this is the subtree containing the unique path between these two nodes), and (ii) a subtree corresponding to each component of $G-X-Y$ which has edges only to the vertices of Y or only to the vertices of X. Furthermore, by the inductive hypothesis, each of these smaller $\mathcal{F}$-block trees is unique, and Lemma 2.3.14 implies there is a unique way to combine them to obtain an $\mathcal{F}$-block tree of G. □

Remark 2.3.16. *The above proof also shows that every cut in $\mathcal{F}$ appears in the $\mathcal{F}$-block tree.*

Recall that 1-cuts of a (connected) graph are trivially laminar and strong 2-cuts in a 2-connected graph are laminar by Lemma 2.3.7. Combined with Theorem 2.3.15, we obtain the uniqueness of block trees and 2-block trees.

Corollary 2.3.17. *The 1-block tree of a connected graph is unique.*

Corollary 2.3.18. *The $\mathcal{F}$-block tree of a 2-connected graph where $\mathcal{F}$ is the family of all strong 2-cuts is unique.*

2.4 Tree decomposition types

Having already discussed $\mathcal{F}$-block trees in the previous section, in this section, we introduce two other tree decomposition types and compare all 3 types. All tree decompositions seen up to this point can be classified as one of these 3 types. Tree decompositions from later sections can also be classified in the same way.

At the end of this section, we will see how to transform one decomposition into another. All tree decomposition types are based on a family of cuts $\mathcal{F}$ in a graph G.

2.4.1 Recursive trees

A second type of tree decomposition occurs naturally in recursive algorithms which we dub *recursive tree decompositions*. This rooted tree is shaped exactly as the recursion tree of the algorithm. Each interior node is labelled by a cut in $\mathcal{F}$ which the algorithm decomposed on and each leaf node is labelled by a final graph the algorithm stopped on.

Definition 2.4.1. Let G be a graph and $\mathcal{F}$ a family of cutsets in G. A *recursive $\mathcal{F}$-tree*, written $[T, \mathcal{G}]$, is a rooted tree T with a set of graphs $\mathcal{G} = \{G_t\}_{t \in T}$ with the following properties.

1. There are two types of nodes in T: graph nodes and cutting nodes. All leaves are graph nodes and all internal nodes are cutting nodes.
2. If G contains no cuts of $\mathcal{F}$ then T is a single graph node r and $G_r = G$.

3. If G contains a cut in $\mathcal{F}$ then the root r of T is a cutting node and G_r is a clique with vertex set X for some cut $X \in \mathcal{F}$.
   (a) Let $T_1, \ldots, T_k$ be the components (subtrees) of $T - t$. Then $G - X$ has k decomponents $C_1, \ldots, C_k$ such that T_i is a recursive $\mathcal{F}$-tree for C_i.
   (b) For each i, there is an edge from r to the root of T_i.

As with $\mathcal{F}$-block trees, we can define an equivalence as follows.

Definition 2.4.2. We say two recursive $\mathcal{F}$-trees $[T_1, \mathcal{G}_1]$ and $[T_2, \mathcal{G}_2]$ of G are *equivalent* (or the *same* tree) if there is an isomorphism f from T_1 to T_2 such that for each $t \in V(T_1)$, $G_{1,t}$ is isomorphic to $G_{2,f(t)}$ (as graphs labelled by vertices of G).

The criteria for obtaining uniqueness for recursive $\mathcal{F}$-trees differs from those for $\mathcal{F}$-block trees. For a recursive $\mathcal{F}$-tree, only the decomposition order affects the final tree we obtain as we always connect the root to the roots of its children. But since the recursive $\mathcal{F}$-tree is rooted, we could also obtain the same tree with different roots. In fact, as with $\mathcal{F}$-block trees, while the multiset of graph nodes or cutting nodes in two recursive $\mathcal{F}$-trees are the same, the actual trees may differ.

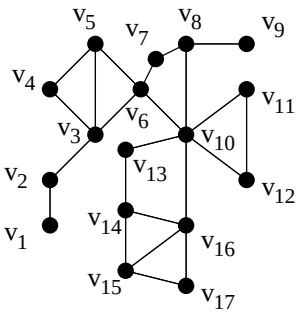
For an example, see Figure 2–5 in which the block tree of a connected graph G and two different recursive $\mathcal{F}$-trees of G where $\mathcal{F}$ is all cutvertices of G are shown.

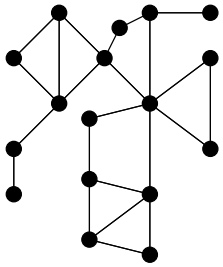
We've seen two examples of recursive $\mathcal{F}$-tree. One such recursive $\mathcal{F}$-tree where $\mathcal{F}$ is the set of cutvertices in Section 2.1. Hopcroft and Tarjan's tree of split components is another example of such trees.

2.4.2 Intersection trees

We now describe our third and final type of tree, the intersection tree. This tree is essentially the $\mathcal{F}$-block tree where only graph nodes are present and the cutting nodes are implicitly encoded in the intersection of adjacent graph nodes.

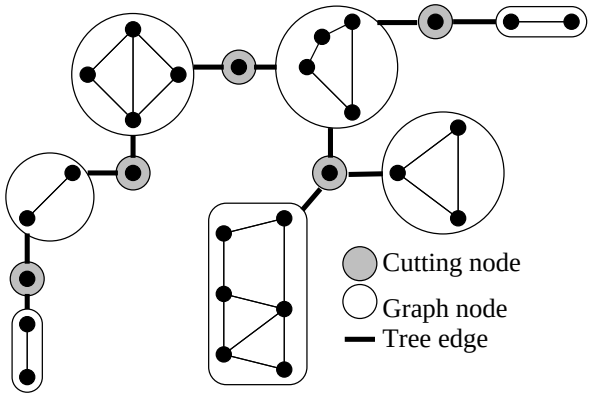
Definition 2.4.3. Let G be a graph and $\mathcal{F}$ a family of cutsets in G. An *$\mathcal{F}$-intersection tree* $[T, \mathcal{G} = \{G_t\}_{t \in T}]$ is defined as follows.



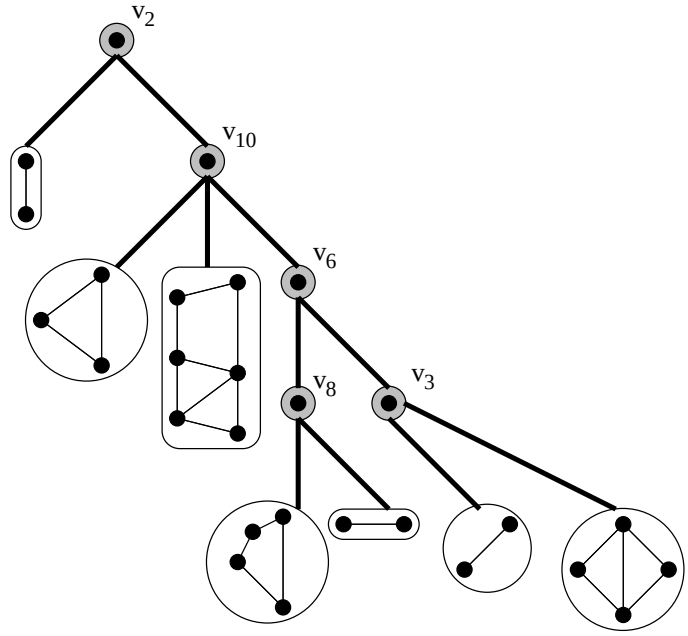


G

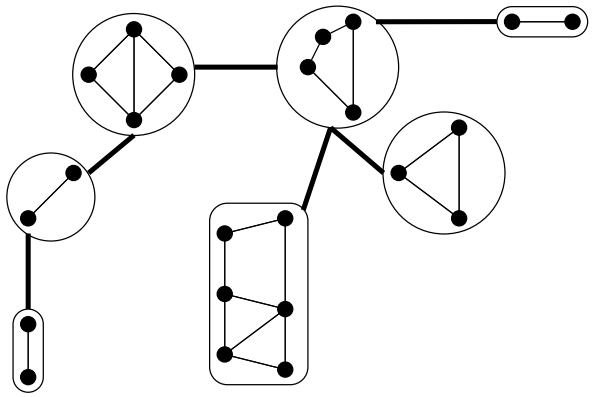
F is all 1−cuts of G



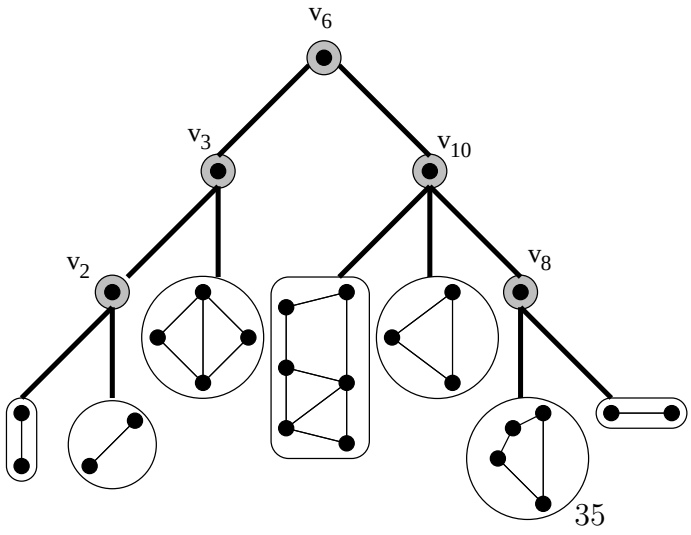
F−block tree of G



Recursive F−tree of G



F−intersection tree of G



A different recursive F−tree of G

Figure 2–5: The three different types of trees where $\mathcal{F}$ is the family of all cutvertices of G.

1. G_t is a graph (not necessarily a subgraph of G) for each $t \in T$ that does not contain a cut in $\mathcal{F}$.
2. For each edge t_1t_2 of T, $X = V(G_{t_1}) \cap V(G_{t_2})$ is a cutset in $\mathcal{F}$ and X is a clique in both G_{t_1} and G_{t_2}.
3. For each $v \in V(G)$, the nodes of T containing v form a subtree.
4. Every vertex $v \in V(G)$ appears in some G_t.
5. Every edge $uv \in E(G)$ appears in some G_t.

If (2) in the above condition for an $\mathcal{F}$-intersection tree is weakened to the following, we refer to the tree as a *weak $\mathcal{F}$-intersection tree*.

2. For each edge t_1t_2 of T, $X = V(G_{t_1}) \cap V(G_{t_2})$ is a cutset in $\mathcal{F}$ (but X is not required to be a clique in both G_{t_1} and G_{t_2}).

For compatibility with the other tree types, we will still refer to nodes of an $\mathcal{F}$-intersection tree as "graph nodes" even though there are no cutting nodes.

See Table 2.4.2 for a table of trees introduced in this section and their types.

2.4.3 Tree type equivalence

We now see that all 3 tree types in this section are not so different. For a fixed $\mathcal{F}$, the following algorithms transform a tree of one type into a tree of the other type.

As we stated in the previous section, $\mathcal{F}$-intersection trees are $\mathcal{F}$-block trees without cutting nodes. The following algorithm makes this connection explicit.

We first need to be able to turn an $\mathcal{F}$-intersection tree into an $\mathcal{F}$-intersection tree where no (graph) node contains another. This is easy as we can just contract adjacent nodes for which one contains the other. To do this in linear time we use dynamic programming.

Algorithm 2.4.4.

Input:

- A graph G,
- a family of cutsets $\mathcal{F}$ in G, and

- a $\mathcal{F}$-intersection tree $[T, \mathcal{G}]$ of G.

Output: An $\mathcal{F}$-block tree $[T', \mathcal{G}']$ of G

Description and analysis. We replace each edge of T by a cutting node adjacent to its endpoints (in later sections, we see that this operation is known as *subdividing an edge*) and label this new cutting node by the intersection of the edge we replaced. (Optional) If a node of T is adjacent to two newly created cutting nodes that are the same, we replace them by a single cutting node. More formally, we proceed as follows.

1. For each $t \in V(T)$, let $G'_t = G_t$ be a graph node.
2. For each $e = t_1t_2 \in E(T)$, let $G'_e = G_{t_1} \cap G_{t_2}$ be a cutting node.
3. Let T' be the vertex-edge incidence graph of T. I.e., $V(T') = V(T) \cup E(T)$ and $E(T') = \{te | t \text{ is an endpoint of } e\}$.
4. For each graph node t of T', bucket sort its neighbouring cutting nodes by label. If any bucket contains more than one node, replace all cutting nodes in that bucket by a single cutting node adjacent to the union of their neighbours in T.
5. Return $[T', \mathcal{G}']$.

□

To turn an $\mathcal{F}$-block tree into an $\mathcal{F}$-intersection tree, we cannot reverse the above transformation as not all cutting nodes have degree 2 in an $\mathcal{F}$-block tree.

Algorithm 2.4.5.

Input:

- A graph G,
- a family of cutsets $\mathcal{F}$ in G, and
- an $\mathcal{F}$-block tree $[T, \mathcal{G}]$ of G.

Output: A recursive $\mathcal{F}$-tree $[T', \mathcal{G}']$ of G.

Description and analysis. We consider the cutting nodes of $[T, \mathcal{G}]$ in any order and decompose on these cutting nodes. More formally, we use the following recursive procedure.

- Let G'_r be any cutting node G_t of T.
- Let $T_1, \ldots, T_k$ be the trees in $T - t$.
- Recurse on each $T_1, \ldots, T_k$ (and corresponding decomponent of $G - V(G'_r)$) to obtain recursive $\mathcal{F}$-trees $[T'_1, \mathcal{G}'_1], \ldots, [T'_k, \mathcal{G}'_k]$.
- Take the disjoint union of all these recursive $\mathcal{F}$-trees together with G'_r and add an edge from r to each of the roots of $T'_1, \ldots, T'_k$.
- Return the resulting recursive $\mathcal{F}$-tree $[T', \mathcal{G}']$

□

Algorithm 2.4.6.

Input:

- A graph G,
- a family of cutsets $\mathcal{F}$ in G, and
- a recursive $\mathcal{F}$-tree $[T, \mathcal{G}]$ of G.

Output: An $\mathcal{F}$-block tree $[T', \mathcal{G}']$ of G.

Description and analysis. We perform a post-order traversal of T to obtain a decomposition order (we ignore the leaf graph nodes of T).

To make this algorithm more efficient, we use dynamic programming.

1. If T is a single node t then build $T' = T = \{t\}$ with $G_t = G'_t$.
2. Otherwise, T is a tree rooted at r with children $r_1, \ldots, r_k$. We proceed as follows.
   (a) Build $G_1, \ldots, G_k$, the decomponents of $G - V(G_r)$
   (b) Recurse on each subtree rooted at r_i to turn the recursive $\mathcal{F}$-tree of G_i into an $\mathcal{F}$-block tree $[T_i, \mathcal{G}_i]$.

(c) By Lemma 2.3.14, since G_r is a clique in each G_i, we can find a graph node G_{i,t_i} containing $V(G_r)$ in each $\mathcal{F}$-block tree $[T_i, \mathcal{G}_i]$.

(d) Take the disjoint unions of all T_i for $i = 1, \ldots, k$, add a root cutting node r labelled by G_r and add an edge from r to t_i in T_i.

Return the resulting tree decomposition.

□

Algorithm 2.4.7.

Input:

- A graph G,
- a family of cutsets $\mathcal{F}$ in G, and
- an $\mathcal{F}$-block tree $[T, \mathcal{G}]$ of G.

Output: An $\mathcal{F}$-intersection tree $[T', \mathcal{G}']$ of G

Description and analysis. We arbitrarily root the tree T at a graph node, add an edge from each graph node to any of its descendants two levels below (i.e., if we think of all edge of T as being directed away from the root and take the square of the tree (as a graph)) and delete all cutting nodes.

More formally,

1. Let $T' = T$.
2. Pick an arbitrary graph node r and root T' at r.
3. For any graph node t that is not r, add an edge (in T') from t to the parent of the parent of t.
4. Delete all cutting nodes in T'.
5. Return T'.

□

Definition 2.4.8. For a given tree, we refer to its *block version*, *recursive version* and *intersection version* to mean the tree obtained by applying some of Algorithms 2.4.4, 2.4.5,2.4.6 and 2.4.7.

2.5 Clique cutset trees

In this section, we examine one example of $\mathcal{F}$-block trees where $\mathcal{F}$ is a set of *minimal clique cutsets*.

Definition 2.5.1. A cutset in a graph that is also a clique is a *clique cutset*.

A *minimal clique cutset* is a clique cutset not completely containing any other (clique) cutset.

Two minimal clique cutsets X and Y of a graph G do not separate each other as $X - Y$ is a clique (in $G - Y$) and $Y - X$ is a clique (in $G - X$). Thus, if $\mathcal{F}$ is a family of minimal clique cutsets of G then $\mathcal{F}$ is laminar and by Lemma 2.3.12, all of $\mathcal{F}$ appear in all $\mathcal{F}$-block trees ($\mathcal{F}$ is component minimal since minimal clique cutsets are minimal cutsets).

We start by showing this $\mathcal{F}$-block tree is unique. We prove the analoguous version of Lemma 2.3.14.

Lemma 2.5.2. *Let G be a graph, $\mathcal{F}$ a family of minimal clique cutsets of G and $[T, \mathcal{G}]$ an $\mathcal{F}$-block tree of G. Then any minimal clique cutset X in G not indexing a cutting node appears in a unique graph node of T.*

Proof. Suppose X is in two graph nodes. Then X also appears on the path between these two graph nodes. But this path includes at least one cutting node since T alternates between cutting nodes and graph nodes. So X is contained in the index G_t of such a cutting node. Since X is minimal and G_t is a minimal clique cutset, $X = G_t$, a contradiction. □

Theorem 2.5.3. *Let G be a graph and $\mathcal{F}$ a family of minimal clique cutsets of G. Then G has a unique $\mathcal{F}$-block tree.*

Proof. We proceed by induction on $|\mathcal{F}|$. We can assume G has a cut in $\mathcal{F}$. We need only show that every two permutations of the cuts in $\mathcal{F}$ (representing the order in which we decompose on cuts) yield the same tree. Since we can move between any two permutations by repeatedly transposing adjacent elements, it is enough to show that if π_1 is a permutation and π_2 is obtained from π_1 by transposing two adjacent elements then π_1 and π_2 yield the same tree.

If π_1 and π_2 both have first element X, then the corresponding trees both have a cutting node t corresponding to X such that deleting t breaks the tree up into subtrees corresponding to the graphs obtained when we decompose G on X. By the inductive hypothesis, these subtrees are unique, and by Lemma 2.5.2, there is only one way to combine them on a cutting node corresponding to X. So, we see that the two permutations do indeed yield the same tree.

Otherwise, there exists two cuts X, Y such that π_1 begins X, Y and π_2 begins Y, X. From the definition of the tree corresponding to a permutation we see that for both permutations, if we delete the nodes corresponding to X and Y then we obtain (i) one subtree which is the $\mathcal{F}$-block tree of the union of X, Y, and all the components of $G - X - Y$ with an edge to both X and Y, and (ii) a subtree corresponding to each component of $G - X - Y$ which has edges only to the vertices of Y or only to the vertices of X. Furthermore, by the inductive hypothesis, each of these smaller $\mathcal{F}$-block trees is unique, and Lemma 2.5.2 implies there is a unique way to combine them to obtain an $\mathcal{F}$-block tree of G. □

Note that for a family $\mathcal{F}$ of minimal clique cutsets, we do not add any edges when decomposing on cuts to create the $\mathcal{F}$-block tree (since all cuts are already cliques).

When $\mathcal{F}$ is all minimal clique cutsets, we refer to the $\mathcal{F}$-block tree as a *clique cutset tree*.

Definition 2.5.4. The $\mathcal{F}$-block tree of G where $\mathcal{F}$ is the set of all minimal clique cutsets of G is the *clique cutset tree of* G.

Structure theorem

If graph nodes of the clique cutset tree of G are also cliques then G actually belongs to the class of *chordal graphs* [29].

Definition 2.5.5. A graph G is *chordal* if it does not contain an induced cycle of length greater than 3.

In fact, the converse also holds. That is, the clique cutset tree of a chordal graph contains only cliques as graph nodes. Gavril [29] was the first to characterize chordal graphs with such a decomposition.

Theorem 2.5.6. *[29] If a graph G is chordal (i.e., G does not contain an induced cycle of length greater than 3) then either*

- *G contains a cutset which is a clique, or*
- *G is a clique.*

It follows immediately that the graph nodes are cliques in any clique cutset tree of G.

The tree decomposition for chordal graphs can be introduced in a second way, using *subtree intersection representation*. We define this decomposition, which is a special case of the Robertson-Seymour tree decomposition of Section 2.6.

Definition 2.5.7. A *subtree intersection representation* of a graph G is a tree T and a family of cliques X_t of G for each $t \in V(T)$ such that

1. G is the union of all cliques X_t, and
2. for any vertex v of G, the nodes of T containing v induce a subtree of T.

This is essentially the intersection version of the clique cutset tree and we can easily transform one tree into another (see Section 2.4.3). However, the (implicit) clique cutsets of a subtree intersection representation are not required to be minimal and the corresponding $\mathcal{F}$-block tree may not be unique. In fact, there may be several subtree intersection representations for the same chordal graph G.

Gavril's theorem can be restated using subtree intersection representations as follows.

Theorem 2.5.8. *[29] A graph is chordal if and only if it has a subtree intersection representation.*

2.5.1 Building a clique cutset tree

In this section, we describe an algorithm which builds a clique cutset tree of a (not necessarily chordal) graph.

We assume G is connected as otherwise, we can first decompose G on the empty (clique) cutset, build a clique cutset tree for each component of G and add an edge from the root (empty) cutting node to any graph node of each component.

We present a simple $O(n(m+n))$ algorithm due to Whitesides [75] for finding one clique cutset. We can then use this algorithm as a subroutine to build a clique cutset tree.

Algorithm 2.5.9. [75] *Input:* A graph G.

Output: Either

1. A clique cutset in G if there is one, or
2. "G has no clique cutset".

This algorithm requires a subroutine for finding either a chordless cycle of length at least four or a clique cutset in a graph that is not a clique which we describe later.

Description.

1. Determine if G is a clique. If so, return "G has no clique cutset".
2. Apply Algorithm 2.5.10 to G which determines if G is chordal and proceed depending on whether the output is a clique cutset or an induced cycle.
3. If **a clique cutset of G is returned**, return that clique cutset.
4. If **an induced cycle C of G is returned**, "grow" a set S as follows.
   (a) Initialize $S = V(C)$.
   (b) Repeat the following until $S = V(G)$.
      i. Find a component U of $G - S$ and compute R, the vertices of S adjacent to some vertex of U.
      ii. If R is a clique, return R as a clique cutset of G.

iii. Otherwise,

A. find an induced path P between two non-adjacent vertices of R whose interior is in U.

B. Add all interior vertices of P to S.

(c) Now, $S = V(G)$ and we return "G has no clique cutset".

Analysis. **Correctness.** If G is a clique then G contains no cutsets and therefore no clique cutset so our output is correct at step 1.

At step 4(b)ii, if we return a clique R then R is indeed a clique cutset as it separates U from the component of $G - R$ containing $S - R$ (which is non-empty as R is a clique but S is not).

It remains to show that if we grow S to be all of $V(G)$ and reach step 4c, G cannot contain a clique cutset X. To do so, we show that for any clique cutset X of G, $S - X$ is contained in a unique component of $G - X$ throughout each iteration of the while loop.

Initially, $G[S]$ is an induced cycle of length at least 4 so any clique intersects one or two consecutive vertices of this cycle so $S - X$ is connected in $G - X$.

Now suppose by induction that $S - X$ is contained in a component U of $G - X$ and we add P with endpoints x, y to S. The argument is the same as the initial case where $S - X$ is used to complete P to a cycle. Since P is induced, X contains at most two consecutive vertices of P. So after removing X, every vertex of $P - X$ has a path in $P - X$ to x or y (since x and y are not adjacent and thus non-consecutive in P). Thus, all remaining vertices of P (i.e., $P - X$) are in the same component of $G - X$ as $S - X$.

By induction, S remains in a single component of $G - X$. Therefore, in all iterations of the while loop, $S - X$ is contained in a single component of $G - X$.

Running time: We can determine if G is a clique in $O(m)$. Algorithm 2.5.10 runs in $O(m + n)$.

Note that the size of S increases at each iteration so there are at most n iterations of the while loop of step 4b. In each iteration, building R and U each take $O(m+n)$. Finding an induced path P (using, say BFS) also takes $O(m+n)$. Finally, updating S takes $O(n)$. Thus, each iteration runs in $O(m+n)$.

Therefore, the whole algorithm runs in $O(n(m+n))$. □

We now describe the subroutine needed to initialize Algorithm 2.5.9. The idea is similar to one iteration of the main while loop of Algorithm 2.5.9.

Algorithm 2.5.10. [75] *Input:* A graph G that is not a clique.

Output:

1. A clique cutset of G if G is chordal, or
2. an induced cycle of length at least 4 in G.

Description and analysis.

1. Find a vertex v of degree less than $|V(G)|-1$. It exists because G is not a clique.
2. Let U be a component of $G-N(v)-v$. Let C be the vertices of $N(v)$ which have a neighbour in U.
3. If C is a clique return it.
4. Otherwise,
   (a) let x and y be two non-adjacent vertices of C and find an induced path between them through U.
   (b) Add v to form an induced cycle of length at least 4 and return it.

Running time: Finding two non-adjacent vertices, building S and finding U all take $O(m+n)$. Finding an induced path P (using, say BFS) also takes $O(m+n)$. Thus, the entire algorithm runs in $O(m+n)$. □

We use Algorithm 2.5.9 as a subroutine in the the following algorithm, we obtain an $O(n^2(m+n))$ algorithm for building a clique cutset tree for any graph.

Algorithm 2.5.11.

Input: A graph G,

Output: A clique cutset tree of G.

Running time: $O(n^2(m+n))$.

Description and analysis. 1. Run Algorithm 2.5.9 to determine if G has a clique cutset.

2. If not, return the tree consisting of a single node labelled by G.
3. Otherwise, we found a clique cutset X in G.
   (a) Build a node t with $V(G_t) = X$, $E(G_t) = \{xy | x, y \in X\}$.
   (b) Recurse on all decomponents $C_1, \ldots, C_k$ of $G - X$. Let $[T_i, \mathcal{G}_i]$ be the $\mathcal{F}$-block tree of C_i obtained recursively and let T be the disjoint union of all these trees.
   (c) For each i, find a graph node $t_i \in T_i$ such that $X \subseteq V(G_{t_i})$.
   (d) For each i, add an edge $tt_i \in E(T)$.
   (e) Return $[T, \mathcal{G}]$.

□

A faster algorithm for building a clique cutset tree was later developed by Tarjan [68] with $O(mn)$ running time. Tarjan actually builds the *binary decomposition tree*, the recursive version of the clique cutset tree we defined (the *atoms* of the binary decomposition tree are the graph nodes of the clique cutset tree).

For chordal graphs, or more generally the class of *clique separable* graphs (see [30]), a clique cutset tree can be built in $O(n^2(m+n))$.

2.5.2 Application: Colouring chordal graphs

We now see how a subtree intersection representation or clique cutset tree can help us colour chordal graphs in polynomial time. This gives us a flavour of algorithms using tree decompositions.

Definition 2.5.12. By *colouring* a graph G, we mean an assignment c of integers (or colours) to vertices of G so that adjacent vertices receive different integers (colour). I.e., $c : V(G) \to \{1, \ldots, k\}$ where $c(u) \neq c(v)$ if $uv \in E(G)$.

A *k-colouring* of G is a colouring of G using at most k different colours.

An *optimal colouring* of a G is a colouring of G which uses the minimum number of colours amongst all colourings of G. The number of colours used in an optimal colouring is denoted by $\chi(G)$.

We solve the following problem on chordal graphs.

Problem 2.5.13. COLOURING

Input: *A graph G.*

Output: *An optimal colouring of G.*

Colouring (also called vertex colouring) in general is NP-complete, even to approximate within $n^{1-\varepsilon}$ for any $\varepsilon > 0$ [78]. We need the following well known properties of colourings.

Remark 2.5.14. *In any colouring c of a graph G, the vertices in a clique $X = \{x_1, \ldots, x_k\}$ of G all receive different colours.*

Thus, we can always permute the colours of c so $c(x_i) = i$.

Lemma 2.5.15. *If X is a clique cutset of G and $C_1, C_2, \ldots, C_\ell$ are the decomponents of $G - X$ then $\chi(G) = \max(\chi(C_1), \ldots, \chi(C_\ell))$.*

We prove this lemma algorithmically.

Algorithm 2.5.16. *Input:*

- A graph G,
- a cut X in G,
- the decomponents $C_1, \ldots, C_\ell$ of $G - X$,
- a k_i-colouring c_i of C_i for $i = 1, \ldots, \ell$.

Output: A $\max(\{k_i\}_{i=1}^{\ell})$-colouring c of G.

Description. This algorithm simply merges all colourings of the decomponents after matching them on X. More formally, we proceed as follows.

1. Label the vertices of X by $x_1, \ldots, x_k$.
2. Permute the colours of c_i so $c_i(x_j) = j$ for each i (using Remark 2.5.14).
3. Set $c(v) = c_i(v)$ if $v \in C_i - X$ and $c(v) = j$ for $v = x_j \in X$.

Analysis. c is a colouring of G since any edge of G is an edge of some decomponent. It uses $\max(\{k_i\}_{i=1}^{\ell})$ colours since it always uses all colours from some c_i. This algorithm proves $\chi(G) \leq \max(\chi(C_1), \ldots, \chi(C_\ell))$.

We now prove the reverse inequality when X is a clique cutset, thus showing the optimality of the colouring c of G. If X is a clique cutset, $C_1, \ldots, C_\ell$ are subgraphs of G so any k-colouring c of G is a k-colouring of C_i by simply restricting c to C_i (i.e., $c_i(v) = c(v)$ if $v \in V(C_i)$). Therefore, $\chi(C_i) \leq \chi(G)$ for each i. □

Now that we know how to merge colourings on clique cutsets, it remains to colour the graph nodes of a clique cutset tree. By Lemma 2.5.6, the graph nodes are cliques and are coloured optimally by simply assigning a different colour to each vertex of the clique.

After we colour all graph nodes, we apply Algorithm 2.5.16 to merge colourings (in the reverse order in which the clique cutset tree was built, for example). This gives us an optimal colouring of G.

We note that there is an easier way to colour chordal graphs in general. All chordal graphs contain a *simplicial vertex*, a vertex whose neighbourhood is a clique [25], so we can

1. delete a simplicial vertex v,
2. recurse on $G - v$,
3. (greedily) assign the lowest numbered colour to v not assigned to a neighbour of v.

Since the neighbourhood of v is a clique, $N(v)$ is a clique cutset in G or G is a clique and in both cases, our colouring is optimal by Lemma 2.5.15.

2.6 Robertson-Seymour tree decomposition

We are now ready to introduce Robertson-Seymour tree decompositions and the theory of graph minors in which they are used. The original goal of their work was to answer Wagner's conjecture.

Conjecture 2.6.1 (Wagner's conjecture). For any infinite sequence of graphs, there exists a graph of the sequence containing another as a minor.

The *Robertson-Seymour tree decomposition* is an $\mathcal{F}$-intersection tree, with no particular restrictions on $\mathcal{F}$.

In the context of graph minors, we are interested in properties of the graph nodes of a Robertson-Seymour tree decomposition. In particular, in answering Wagner's conjecture, they also gave an (approximate) characterization for all graphs excluding a fixed minor H.

We postpone the statement of this structure theorem for the moment as it is fairly involved. We start by introducing the bounded treewidth decomposition and discuss algorithms to build such a decomposition. We then delve into the thoery of graph minors, show how it relates to tree decompositions and state Robertson and Seymour's structure theorem. Finally, we give some applications of the bounded treewidth tree decomposition.

2.6.1 Treewidth

Although the Robertson-Seymour tree decompositionhas no particular restrictions on the set of cuts $\mathcal{F}$, the corresponding structure theorems are obtained when we restrict the graphs indexing graph node. As a first example, we simply restrict the size of each graph nodes. This is formalized using the notion of treewidth.

Definition 2.6.2. Let G be a graph.

The *width* of a Robertson-Seymour tree decomposition $[T, \mathcal{G}]$ is $\max_{t \in T} |V(G_t)| - 1$.

The *treewidth* of G, denoted $\mathrm{tw}(G)$, is the minimum width over all tree decomposition of G. A Robertson-Seymour tree decomposition of width $\mathrm{tw}(G)$ is an *optimal tree decomposition*.

The notation of a tree decomposition and treewidth was first introduced by Halin [37] (under different names) before Robertson and Seymour re-introduced it for the theory of graph minors.

We state the following simple results without proof.

Remark 2.6.3. *The treewidth of a graph is at least one less than the size of its maximum clique.*

The treewidth of a tree is at most 1.

The treewidth of a chordal graph is the size of its maximum clique minus one.

Many NP-hard problems can be solved in linear time on graphs whose treewidth is bounded by a fixed constant k (typical running times are super-exponential in k). We now see how to build an Robertson-Seymour tree decomposition of width at most k (if it exists) and how to use these decompositions to solve NP-hard problems.

Properties of bounded treewidth decompositions

We now state some simple properties of Robertson-Seymour tree decompositions we will need. The first is that the treewidth of a graph G is at most the maximum treewidth of its decomponents.

Lemma 2.6.4. *(see e.g., [11]) Let G be a graph and X a cut in G with decomponents $G_1, G_2, \ldots, G_k$. Then* $\text{tw}(G) \leq \max(\text{tw}(G_1), \text{tw}(G_2), \ldots, \text{tw}(G_k))$.

This is usually stated in terms of the *clique sum* operation (instead of decomponents) where we combine $G_1, \ldots, G_k$ on a cut.

Definition 2.6.5. Let G be a graph and X a i-cut in G with decomponents $G_1, \ldots, G_k$. Then we say G is the *i-clique sum* of $G_1, \ldots, G_k$.

As an easy corollary, when one of the decomponents is a clique.

Corollary 2.6.6. *Let G be a graph and $v \in V(G)$ be such that $N(v)$ is a clique, then* $\text{tw}(G) \leq \max(\text{tw}(G - v), \deg(v) + 1)$.

The second property is that treewidth is non-increasing for subgraphs.

Lemma 2.6.7. *Let G be a graph and H a subgraph of G then* $\mathrm{tw}(H) \leq \mathrm{tw}(G)$.

We can simply restrict a Robertson-Seymour tree decomposition of G to a Robertson-Seymour tree decomposition of H (and the width will not increase).

2.6.2 Building bounded treewidth decompositions

In general, if the treewidth of the input graph is unbounded, determining the treewidth of the graph is NP-complete even for graphs of bounded degree [10], bipartite graphs [45] or complements of bipartite graphs [1]. On the other hand, treewidth can be computed exactly in polynomial time for chordal graphs (indeed the clique cutset tree is a tree decomposition of optimal width), permutation graphs [9], circular-arc graphs [65], circle graphs [45] and distance-hereditary graphs [13].

For graphs of bounded treewidth, Bodlaender [8] gave the first linear time algorithm to build the tree decomposition, improving on Reed's $O(n \log n)$ algorithm [57]. Perkovic and Reed [52] modified Bodlaender's algorithm to also return a (certifying) subgraph of width greater than k and at most $2k$ in case the input graph has treewidth greater than k. Our main result in the next chapter uses some ideas from their algorithm.

2.6.3 Graph minors

In order to state Robertson and Seymour's structure, we need to introduce some notions from the theory of graph mionrs. We recall the definition of edge contraction.

Definition 2.6.8. By *contracting an edge* $e = uv$ in a graph, we mean to remove all edges between u and v and replace the vertices u, v with a single vertex w_e incident to all edges originally incident to either u or v.

Remark 2.6.9. *The graph obtained by contracting a set of edges is always the same, regardless of the order in which we contracted the edges.*

Definition 2.6.10. By *contracting* a subgraph H of a graph G, we mean to contract all edges in $E(H)$.

Definition 2.6.11. A graph H is a minor of a graph G if we can obtain H from G via a sequence of

- vertex deletions,
- edge deletions, and
- edge contractions.

In this case, we say *G has an H-minor* or *G contains an H-minor*.

Remark 2.6.12. *If there is a sequence of vertex deletions, edge deletions, and edge contractions that turns G into H then there is a sequence with all edge contractions occuring at the end.*

Equivalently, H is a minor of G if can be obtained from a subgraph of G via edge contractions. Note that any edge in an intermediate graph of a sequence of contractions corresponds to an edge of the original graph G. The union of all contracted edges induce a forest in G, one tree for each vertex of H.

In fact, instead of a sequence of operations, we can specify a set of vertex disjoint trees of G, one for each vertex of H, such that if uv is an edge in H then there is an edge of G with one endpoint in the tree corresponding to u and the other endpoint in the tree corresponding to v.

This is the definition of a *graph model*.

Definition 2.6.13. An *H-model* in a graph G is a set of vertex disjoint trees $\{T_v | v \in V(H)\}$ and edges $e_{uv} = xy \in E(G)$ with $x \in V(T_u), y \in V(T_v)$ for each $uv \in E(H)$.

We refer T_v as the *vertex image of v* and e_{uv} as the *edge image of uv*.

For an H-model K, we refer to the subgraph of G consisting of the union of $\{T_v | v \in V(H)\}$ and edges $\{e_{uv} | uv \in E(H)\}$ as the *subgraph corresponding to K*. We will refer to *vertices of K* (written $V(K)$) and *edges of K* (written $E(K)$) and other properties of subgraphs as if K were this subgraph.

See Figure 2–6 for an example. We have thus proven the following lemma.

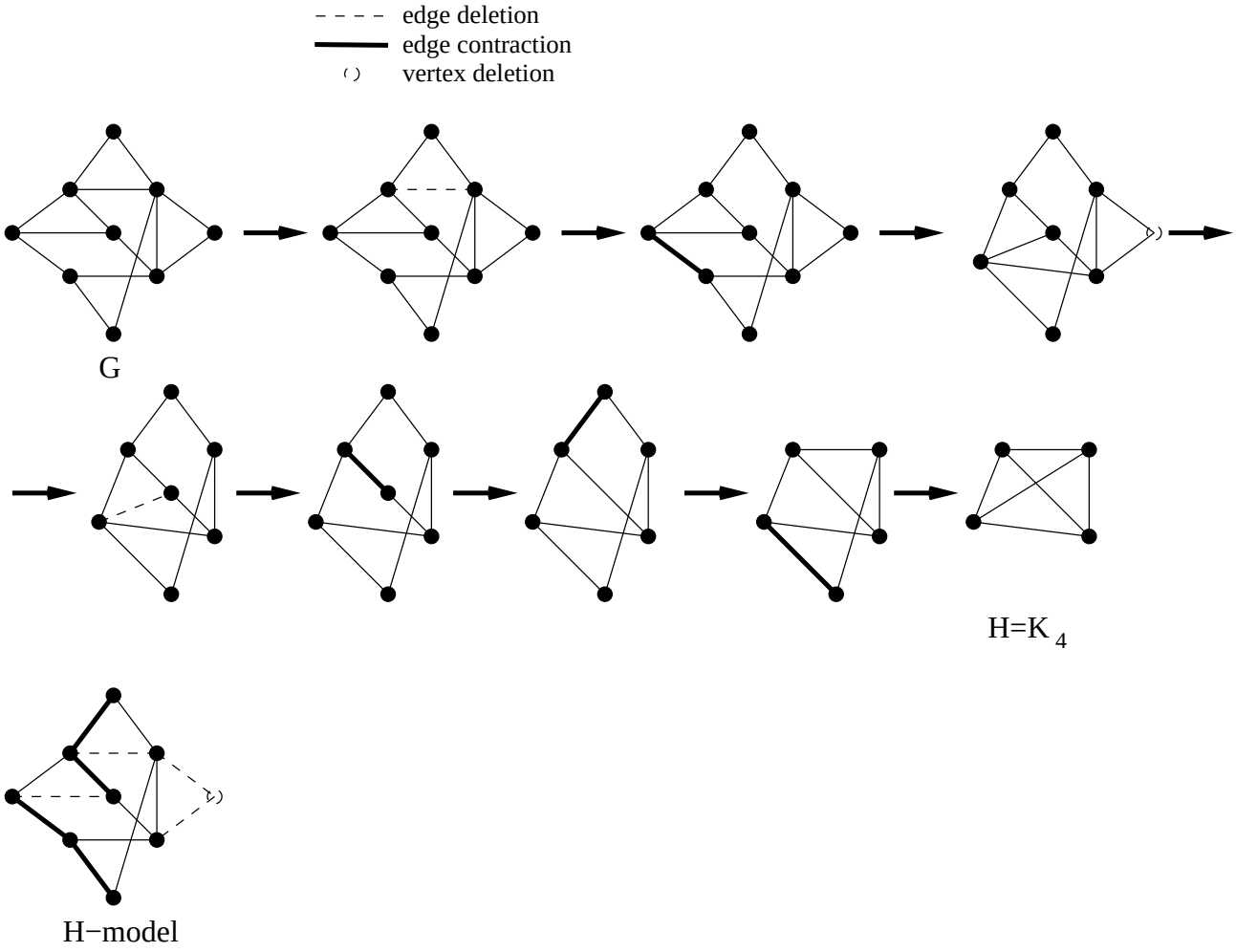


Figure 2–6: A K_4-minor in a graph G and the corresponding model.

Lemma 2.6.14. *A graph G contains H as a minor if and only if G contains an H-model.*

Thus, throughout this thesis, we use minor containment and model containment interchangeably.

We now state some simple properties of minors (and models) that we will use implicitly. It is clear from the definition of a minor that "containing a minor of" is a transitive relation.

Remark 2.6.15. *If G contains H as a minor and H contains F as a minor then G contains F as a minor.*

If G contains an H-model and H contains an F-model then G contains an F-model.

As a special case, if a subgraph H of G contains an F-minor then so does G and G contains an F-model. In, particular, we have:

Remark 2.6.16. *If K_ℓ is a minor of G then all graphs on at most ℓ vertices are minors of G.*

We will need the following algorithm later on.

Algorithm 2.6.17.

Input:

- A graph G,
- a model of H in G, and
- a K_5-model in H.

Output: A K_5-model in G.

Running time: $O(|V(G)| + |E(G)|)$.

Proof. Let $\{Y_v\}_{v \in V(H)}$ be the input H-model in G. Let $\{X_i\}_{i=1}^5$ be the input K_5-model in H. We build sets $Z_i = \bigcup_{v \in V(X_i)} V(Y_v)$ for $i = 1, \ldots, 5$. Return $\{T_i |\ T_i$ is a spanning tree for $G[Z_i], i \in \{1, 2, 3, 4, 5\}\}$ (as a K_5-model of G). Each $G[Z_i]$ is connected since each $Y_v, v \in X_i$ is connected and the edges between these sets (from $\{X_i\}_{i=1}^5$) connect these components to each other. □

This thesis' main focus concerns the algorithmic question of determining whether G contains H as a minor and the structure of graphs that do not contain a fixed graph H as a minor.

Definition 2.6.18. We say that G is *H-minor free* or *G excludes H as a minor* if G does not contain H as a minor.

Complete graph minors are of particular interest to us. We close this section with a very simple example of a structure theorem for graphs excluding a minor H.

Theorem 2.6.19. *A graph G is K_3-minor free if and only if G is a forest.*

This is true due to the fact that being K_3-minor free is equivalent to having no cycles.

Proof. Suppose G contains a cycle C (as a subgraph). Contracting all but three edges of C yields K_3. Therefore, K_3 is a minor of G in this case.

Conversely, suppose G is a forest and thus contains no cycles. Contracting an edge in a tree yields another tree. So all minors of G are also forests. Since K_3 is not a forest, G does not contain K_3 as a minor. □

Equivalently, every graph G without a K_3-minor has an $\mathcal{F}$-intersection tree where each graph node has size at most 2 (for some family $\mathcal{F}$ of cuts in G). The notion of bounding the size of graph nodes will appear again when we introduce the notion of *treewidth*. In particular, this notion will allow us to order all graphs by their "complexity value" (in some sense) and solve problems in time polynomial in the size of the graph but exponential in the complexity values.

2.6.4 Complete graph minors

In this section, we prove some properties of K_ℓ-minor free graphs for any fixed ℓ. We show decomposing on a minimal cutset X where $G-X$ has at least $|X|$ components preserves the property of being K_ℓ-minor free.

Theorem 2.6.20. *If a graph G has a clique cutset X then it has a K_ℓ-minor precisely if there is a component U of $G - X$ such that $G[V(U) \cup X]$ (i.e., the corresponding decomponent) has a K_ℓ-minor.*

Proof. $G[V(U) \cup X]$ is a subgraph of G for any component U of $G - X$ so any K_ℓ-minor of $G[V(U) \cup X]$ is also a K_ℓ-minor of G. We now suppose G contains K_ℓ-model $\{Y_i\}_{i=1}^{\ell}$ and show that $G[V(U) \cup X]$ contains a K_ℓ-minor for some component U of $G - X$.

Since X is a clique, $G[V(U) \cup X]$ contains a $K_{|X|}$-model for any component U of $G - X$ and the theorem is true for $\ell \leq |X|$.

So $\ell > |X|$ and some Y_i does not intersect X. Without loss of generality, it is Y_1. Since $\{Y_i\}_{i=1}^{\ell}$ is a K_ℓ-model, all Y_i intersect X or are contained the component U of $G - X$ containing Y_1 (otherwise, there is no edge between Y_1 and Y_i). We claim $G[V(Y_i) \cap (V(U) \cup X)]$ is connected for each Y_i. If not, two components of $G[V(Y_i) \cap (V(U) \cup X)]$ are connected to each other via a path in G. If P is a shortest such path either we can shorten P using an edge of X (if some of P is outside $G[V(Y_i) \cap (V(U) \cup X)]$) or P is completely contained in $G[V(Y_i) \cap (V(U) \cup X)]$ and in both cases, we have a contradiction. Therefore $\{G[V(Y_i) \cap (V(U) \cup X)]\}_{i=1}^{\ell}$ is a K_ℓ-model in $G[V(U) \cup X]$ as required. □

Theorem 2.6.21. *If X is a minimal cutset of a graph G and the number of components of $G - X$ is at least $|X|$ then the graph obtained from G by adding edges so that X is a clique has a K_ℓ-minor precisely if G has a K_ℓ-minor.*

Proof. G is a subgraph of H, the graph obtained from G by adding a clique onto X so any K_ℓ-minor of G is also a K_ℓ-minor of H. We now suppose H contains K_ℓ-model and show G contains a K_ℓ-minor.

X is a clique cutset in H so by Theorem 2.6.20, $H[U \cup X]$ contains a K_ℓ-model for some component U of $H - X$.

Let $U_2, \ldots, U_{|X|}$ be $|X| - 1$ other components of $H - X = G - X$. We contract all edges in all U_i for $i = 2, \ldots, |X|$. Each component contracts to a vertex of degree $|X|$ since X is a minimal cut. We then contract a maximum matching between the resulting vertices and X (and delete all other components of $G - X$ except U). The resulting graph is $H[V(U) \cup X]$. Thus, G contains $H[V(U) \cup X]$ as a minor and therefore a K_ℓ-model. □

In particular, we can decompose graphs along (k, k)-cuts without affecting K_ℓ-minor freeness.

Corollary 2.6.22. *Let X be a minimal (k, k)-cut in a graph G.*

Then for any $\ell \geq k$, G is K_ℓ-minor free if and only if all decomponents of $G - X$ are K_ℓ-minor free.

We now provide the algorithmic version of the above theorem.

Algorithm 2.6.23.

Input:

- A graph G,
- a minimal (k, k)-cut X in G, and
- a decomponent C of $G - X$

Output: A C-model of G.

Description and analysis.

1. Let $U_1, U_2, \ldots, U_k$ be k components of $G - X$ with $V(C) = V(U_1) \cup V(X)$. Build the graph H obtained from G by contracting each U_i to a single vertex u_i for $i = 2, \ldots, k$ and deleting all other components of $G - X$.
2. Since X is minimal each u_i is adjacent to all of X.

   Let $x_1, \ldots, x_k$ be the vertices of X. Contract $x_i u_i$ for each $i = 2, \ldots, k$.
3. Return H (and the sequence of deletions and contractions performed to obtain H).

□

By the transitivity of the minor relation, we can use the above algorithm to obtain a K_ℓ-model in G given a K_ℓ-model in a decomponent C.

Algorithm 2.6.24.

Input:

- A graph G,
- a minimal (k, k)-cut X in G, and
- a K_ℓ-model in a decomponent C of $G - X$.

Output: A K_ℓ-model in G.

2.6.5 Wagner's theorem

We are now ready to present a more elaborate structure theorem, Wagner's theorem for $K_{3,3}$-minor free graphs and Wagner's theorem for K_5-minor free graphs.

Such structure theorems help us study many families of graphs closed under taking minors (which include graphs embeddable on a fixed surface). As we shall see, this allows such families to be characterized in terms of excluded minors. For example, Kuratowski's theorem characterizing planar graphs can be stated in this way.

Theorem 2.6.25 (Kuratowski's theorem)**.** *[47, 74] A graph G is planar if and only G does not contain $K_{3,3}$ or K_5 as a minor.*

This is also an example of how a topological restriction (being planar) can be translated into a graph minors restriction (excluding $K_{3,3}$ and K_5 as a minor). We will see topology playing an important role again in the theory of graph minors and the Robertson-Seymour structure theorem.

The above theorem is actually Wagner's translation of Kuratowski's theorem in terms of graph minors. Kuratowski's original theorem used subdivisions.

Definition 2.6.26. A graph H is a *subdivision* of a graph G if H is obtained from G by *subdividing* some of the edges, that is, by replacing the edges by paths having at most their endvertices in common.

We say that a graph F has (or contains) a subdivision of G (or G-subdivision) if it contains a subdivision of G as a subgraph.

Theorem 2.6.27. *[47] A graph G is planar if and only G does not contain $K_{3,3}$ or K_5 as a subdivision.*

Subdivisions

If G is a subdivision of H then H is a minor of G as we can simply contract all edges of paths in the subdivision. However, the converse is false (see for example, Figure 2–7).

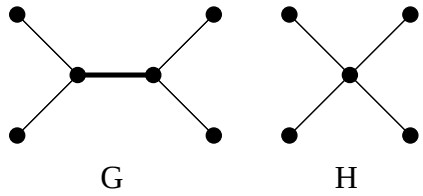


Figure 2–7: H is a minor of G (obtained by contracting the highlighted edge) but G has no H-subdivision as one center of such a subdivision needs to have degree at least 4 and G has maximum degree 3.

However, we can obtain an approximate converse. That is, there exists a finite set of graphs $Z(H)$, all of which have at least $|V(H)|$ vertices and $|E(H)|$ edges such that if H is a minor of G then G contains some graph in $Z(H)$ as a subdivision. The list $Z(H)$ represents all possible trees for vertex images of a model of H.

Theorem 2.6.28. *(see e.g. [54]) For every H, there exists a finite set of graphs $Z(H)$ such that if H is a minor of a graph G then G contains some graph in $Z(H)$ as a subdivision.*

We might think that we can use Kuratowski's theorem to test if G is planar by testing if G contains a K_5-minor and testing if G contains a $K_{3,3}$-minor. However, linear time algorithms for testing planarity were first discovered [40] (see also Section 2.2.3). Linear time algorithms for testing if a graph contains $K_{3,3}$ as a minor [4] and for testing if a graph contains K_5 as a minor were only later discovered. In fact, one of the main contributions of this thesis is a linear time algorithm which determines if an input graph contains a K_5-minor (see Chapter 3).

In contrast to Kuratowski's theorem, Wagner [73, 74] showed we can instead use planarity to characterize $K_{3,3}$-minor free graphs and K_5-minor free graphs.

Theorem 2.6.29 (Wagner's theorem for $K_{3,3}$)**.** *[74] A graph G either*

- *contains a $K_{3,3}$-minor,*
- *is planar,*
- *is the graph K_5,*
- *contains a 0-cut,*

- *contains a 1-cut, or*
- *contains a 2-cut.*

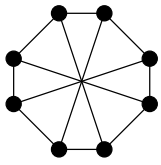


Figure 2–8: The special graph L with 8 vertices and 12 edges. It is also known as V_8 in the literature.

Theorem 2.6.30 (Wagner's theorem for K_5)**.** *[73] A graph G either*

- *contains a K_5-minor,*
- *is planar,*
- *is the graph L,*
- *contains a 0-cut,*
- *contains a 1-cut,*
- *contains a 2-cut, or*
- *contains a 3-cut which splits G into at least 3 components.*

We give an algorithmic proof of this theorem at the end of this chapter (in Section 2.6.9).

We can reformulate Wagner's theorem for K_5-minor free graphs in terms of $\mathcal{F}$-block trees. For simplicity, we refer to 3-cuts in Wagner's theorem as $(3,3)$-*cuts* and more generally define (i,j)-cuts as follows.

Definition 2.6.31. We call a cut of size i that splits G into at least j components an (i,j)-*cut*.

Theorem 2.6.32. *Let G be a connected K_5-minor free graph and $\mathcal{F}$ be the family of all component minimal 1-cuts, 2-cuts and $(3,3)$-cuts of G.*

All graph nodes of an $\mathcal{F}$-block tree of G are either planar or L.

2.6.6 Treewidth and excluding planar graphs

Recall the Robertson-Seymour tree decomposition of a graph G is what we called an $\mathcal{F}$-intersection tree (for general $\mathcal{F}$). For the purpose of excluding minors, we wish to put restrictions on the graph nodes of this tree (other than having no cuts in $\mathcal{F}$). For Wagner's theorem for K_5, the graph nodes are planar or have bounded size.

Robertson and Seymour showed that if H is planar and G is H-minor free then there is a Robertson-Seymour tree decomposition of bounded width. We now state Robertson and Seymour's structure theorem for graphs excluding a fixed planar graph in terms of treewidth.

Theorem 2.6.33. *[60] For any planar graph H, there exists a constant $k(H)$ such that all H-minor free graphs have treewidth at most $k(H)$.*

2.6.7 Structure theorem for H-minor free graphs

The structure theorem for H-minor free graphs for general H is similar to Wagner's theorem for K_5-minor free graphs stated in the last section (Theorem 2.6.30) in that we obtain an $\mathcal{F}$-intersection tree where graph nodes labels are either a fixed graph or embeddable on a fixed surface. This is not quite true as the graph nodes are not quite embeddable (a bounded number of faces can be altered in a very specific way) and we must first delete a bounded number of vertices from each graph node.

We now state this version and fill in the details throughout this section. We follow Lovász's exposition [49].

Definition 2.6.34. We say an $\mathcal{F}$-block tree of a graph G is a $\leq$ 3-block tree if $\mathcal{F}$ is the family of all component minimal 0-cuts, 1-cuts, 2-cuts and 3-cuts of G.

Definition 2.6.35. We say an $\mathcal{F}$-block tree of a graph G is a $\leq k$-block tree if $\mathcal{F}$ is the family of all component minimal of size at most k in G.

Theorem 2.6.36. *[59] Let H be fixed. Then there exists a constant c_H and a surface Σ_H (depending only on H) such that all H-minor free graphs G have an $\leq k$-block tree where graph nodes are "almost embeddable" on Σ_H after we delete at most c_H vertices from them.*

Note that unlike Wagner's theorem, this is only an approximate characterization. That is, all H-minor free graphs have such an $\mathcal{F}$-block tree but graphs with such an $\mathcal{F}$-block tree are not necessarily H-minor free. However, there exists another constant, say c'_H and surface Σ'_H for which the converse holds.

We now present the missing definition of "almost embeddable".

Definition 2.6.37. A graph G is k-almost embeddable on a surface Σ if it is obtained from a graph H embeddable on Σ by

- adding a fringe of width at most k onto (the bounding cycles of) k faces of G, and
- adding at most k vertices with arbitrary adjacencies in G (called *apex vertices*).

Definition 2.6.38. By *adding a fringe of width k to a cycle C*, we mean to

- select p paths $P_1, \ldots, P_p$ of C so that no vertex is in more than k paths,
- for each i, add a vertex v_i adjacent to a subset of $V(P_i)$, and
- for some pairs i, j where $V(P_i)$ intersects $V(P_j)$, add an edge between v_i and v_j.

2.6.8 Recognizing an H-minor free graphs

For arbitrary H, a polynomial time algorithm to construct the decomposition was recently developed by Demaine et al. [22]. In the same paper these authors, building on earlier work of Baker [5], Grohe, and others, develop polynomial time algorithms for a number of optimization problems on this class of graphs. The exponent in the running time of the algorithm depends on $|V(H)|$. It exceeds $\binom{|V(H)|}{2}$.

2.6.9 Algorithmic proof of Wagner's theorem

In this section, we give a flavour of proofs for a structure theorem excluding a fixed graph H as a minor. Specifically, we provide the algorithmic proof of Wagner's theorem for K_5 (Theorem 2.6.30). Our algorithm finds a K_5-model in a non-planar 3-connected graph G with more than 8 vertices and no $(3,3)$-cut.

Our algorithm always succeeds in finding a K_5-model so Wagner's theorem for K_5 follows by checking Wagner's theorem is true for all graphs with fewer than 9 vertices.

Our algorithmic proof assumes Kuratowski's theorem holds. More specifically, it uses the algorithmic version of Kuratowski's theorem [77] which given a non-planar graph returns a $K_{3,3}$ or K_5 subdivision in G.

We are now ready to give the formal specifications.

Algorithm 2.6.39.

Input: A non-planar 3-connected graph G with more than 8 vertices and no $(3,3)$-cut.

Output: A K_5-model of G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis:

Our algorithm uses three subroutines,

(i) the linear time algorithm of [77], which given a non-planar graph finds a K_5 subdivision or $K_{3,3}$ subdivision in it in linear time,

(ii) a linear time algorithm which given a $K_{3,3}$ subdivision within a 3-connected graph G with no $(3,3)$-cuts finds either a subdivision of L or a K_5-model, and

(iii) a linear-time algorithm which given a subdivision of L in a 3-connected graph which is not L finds a K_5-model in the graph.

Algorithm 2.6.39 first runs the first of these subroutines. If a K_5 subdivision is returned, we can easily find and return a K_5-model within the subdivision. Otherwise, the second subroutine is applied. If it returns a K_5-model, so does Algorithm 2.6.39. Otherwise, the third subroutine is applied to the subdivision of L returned by the second subroutine, and Algorithm 2.6.39 returns its output.

To complete our description of Algorithm 2.6.39 it remains to describe the last two subroutines. We note that the second subroutine exploits the fact that if we subdivide two disjoint edges of a $K_{3,3}$ and add an edge between the two new vertices, we obtain L.

Algorithm 2.6.40.

Input:

- A non-planar 3-connected graph G with more than 8 vertices and no $(3,3)$-cut.
- A subdivision F of $K_{3,3}$ in G whose centers are given as two sets of vertices $A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3\}$ and whose edge set has been decomposed into a set of paths $\{Q_{i,j} | 1 \leq i, j \leq 3\}$ such that
  (I) $Q_{i,j}$ has endpoints a_i and b_j and
  (II) these paths are disjoint except at common endpoints.

Output: Either

- a subdivision of L in G, or
- a K_5-model in G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis:

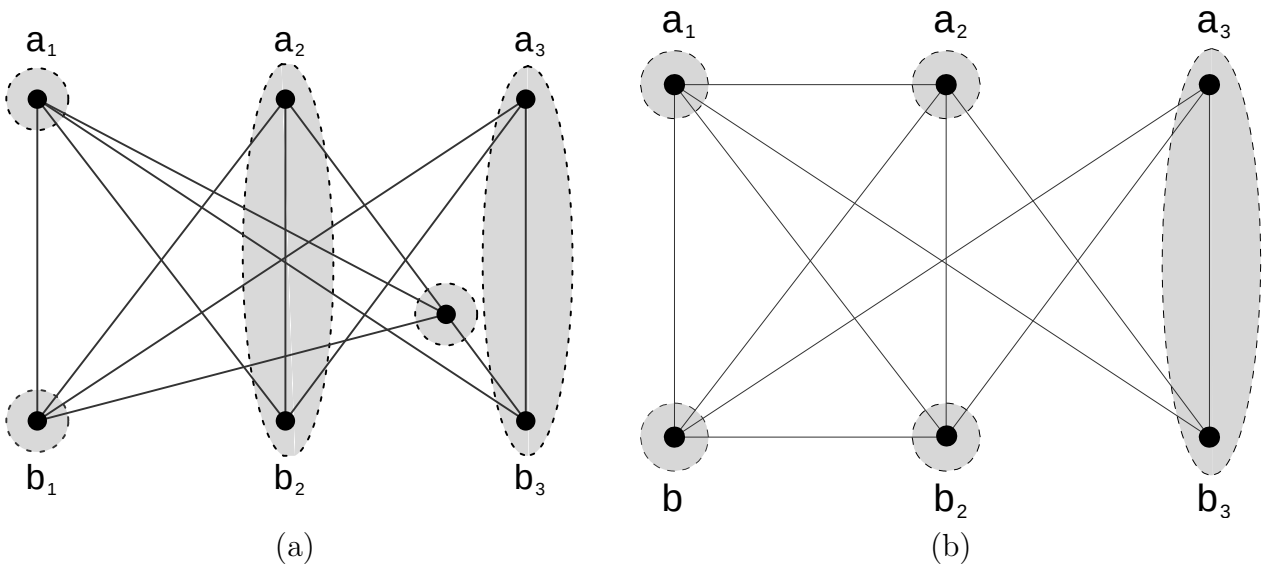


Figure 2–9: K_5-models obtained from a $K_{3,3}$-model.

1. Determine the at most 2 components of $G - A$, and find a component U of $G - A$ containing two elements of B.
2. Find a path P between these two elements of B in U. Find a subpath P_B of P linking two components of $F - A$ which is internally disjoint from F (this can be done by

traversing P from one endpoint until we encounter a vertex of F which is in a different component of $F - A$, and then backtracking along P from this point until we encounter a vertex of F).

3. In the same way, find a subpath P_A of $F - B$ which links two different components of $F - A$, and is internally disjoint from F.
4. If both endpoints of P_B are in B and both endpoints of P_A are in A then
   (a) if P_B and P_A are disjoint we can obtain a K_5-model, where the $Q_{i,j}$ between the vertex of A not on P_A and the vertex of B not on P_B is contracted to a vertex, as depicted in Fig. 2–9(b), while
   (b) if P_B and P_A intersect then we can obtain a K_5 model where we contract the connected graph $(P_B \cup P_A) - A - B$ to a vertex, and use the K_4 subdivision within F centered at $(P_B \cap B) \cup (P_A \cap A)$.
5. Otherwise, relabel A and B so that P_B has an endpoint which is not in B.
6. If the endpoints of P_B are on the interior of disjoint $Q_{i,j}$ then $F \cup P_B$ is a subdivision of L. Return it.
7. Otherwise, we can reindex A, B and $Q_{i,j}$ so that P_B runs from a vertex on the interior of $Q_{1,1}$ to a vertex on $Q_{1,2} - a_1$.
8. $G - B$ has at most two components while $(F \cup P_B) - B$ has three, so as in (2) we can find a path P_1 internally disjoint from $F \cup P_B$ whose endpoints are in different components of $(F \cup P_B) - B$.
9. If the endpoints of P_1 are on the interior of disjoint $Q_{i,j}$ then $F \cup P_1$ is a subdivision of L. Return it.
10. If P_1 has an endpoint on $Q_{1,1}$ (and hence no endpoint on $P_B \cup Q_{1,2}$), we check if this endpoint is a_1. If not, $F \cup P_1 \cup P_B$ contains Fig. 2–9(a) as a minor (by contracting the path between the endpoints of P_1 and P_B on $Q_{1,1}$, the path between the endpoint

of P_B and b_2 on $Q_{1,2}$ and the path from the other endpoint of P_1 to A) and hence a K_5-model, which we return.

If so, $F \cup P_1 \cup P_B$ contains Fig. 2–9(a) as a minor (by contracting the path between the endpoints of P_B and b_1 on $Q_{1,1}$, the path between the endpoint of P_B and b_2 on $Q_{1,2}$ and the path from the other endpoint of P_1 to A in F) and hence a K_5-model, which we return.

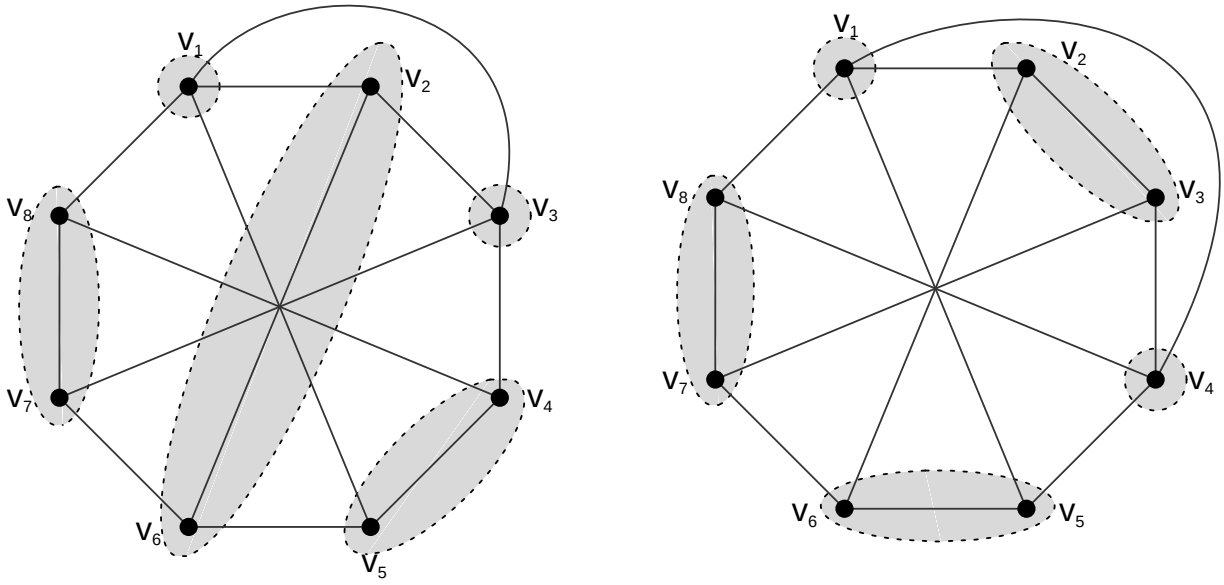
11. If P_1 has an endpoint on the subpath of $Q_{1,2}$ from b_2 to the endpoint of P_B (and hence no endpoint on $Q_{1,1} \cup P_B$), $F \cup P_1 \cup P_B$ contains Fig. 2–9(a) as a minor (by reversing the roles of $Q_{1,1}$ and $Q_{1,2}$ in the above argument (but in this case, the endpoint cannot be a_1)) and hence a K_5-model, which we return.
12. If P_1 has an endpoint on P_B (and hence no endpoint on $Q_{1,1} \cup Q_{1,2}$), $F \cup P_1 \cup P_B$ contains Fig. 2–9(a) as a minor (by contracting the subpath of P_B between the endpoints of P_1 on P_B and the endpoint of P_B on $Q_{1,1}$, the path between the endpoint of P_B and b_2 on $Q_{1,2}$ and the path from the other endpoint of P_1 to A) and hence a K_5-model, which we return.
13.
14. Thus, we can contract the subpath of $Q_{1,1}$ from the endpoint of P_B to a_1, the subpath of $Q_{1,2}$ from the endpoint of P_B to a_2 and the subpaths from each endpoint of P_1 to A in F to obtain Fig. 2–9(b) as a minor and hence a K_5-model, which we return.

We turn now to our second subroutine, it exploits the following observations:

Observation 2.6.41. *Adding an edge e between two non-adjacent vertices of L yields a graph which contains a K_5-model.*

Proof. Let L be labelled as in Fig. 2–10(a). By symmetry, we are free to choose the first endpoint of e to be v_1. Now we need only consider $e = v_1v_3$ or $e = v_1v_4$, as $e = v_1v_7$ is symmetric to the first case and $e = v_1v_6$ is symmetric to the second. In the first case there is

a K_5-model with vertex images $v_1, v_3, v_2v_6, v_4v_5, v_7v_8$ (see Fig. 2–10(a)). In the second there is a K_5-model with vertex images $v_1, v_4, v_2v_3, v_5v_6, v_7v_8$ (see Fig. 2–10(b)). □



(a) (b)

Figure 2–10: K_5-model obtained from L with an added edge.

Observation 2.6.42. *Subdividing an edge uv of L and adding an edge between the new vertex x and any vertex y of $L - u - v$ yields a graph with a K_5-model.*

Proof. One of u or v is not adjacent to y since L contains no triangles. Contract the edge between that vertex and x. This yields L with an added edge (between y and one of u or v). By Observation 2.6.41, this graph contains a K_5-model. □

Observation 2.6.43. *Subdividing two edges of L and adding an edge between the two subdivision vertices yields a graph with a K_5-model.*

Proof. For any vertex v that is an endpoint of the first subdivided edge uv, but not the second, we can contract the path from v to the vertex subdividing uv. We then apply Observation 2.6.42. □

Algorithm 2.6.44.

Input:

- A non-planar 3-connected graph G with more than 8 vertices and no $(3,3)$-cut.
- A subdivision F of L in G.

Output:

- A K_5-model in G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis:

1. If F is not L,
   (a) find a path Q of F corresponding to a subdivided edge uv of L whose interior contains a vertex.
   (b) Use DFS to find a path P in $G - u - v$ from the interior of Q to $F - Q$. (P exists since G is 3-connected).
   (c) By either Observation 2.6.42 or 2.6.43, $F \cup P$ contains a K_5-model which we find and return.
2. If F is L,
   (a) find a vertex $v \in V(G) - V(F)$ (such a vertex exists since G contains more than 8 vertices).
   (b) Find 2 paths P_1, P_2 from v to F that intersect only at v. Let $Q = P_1 \cup P_2$ be the path between two vertices f_1, f_2 of F (internally disjoint from F).
   (c) If $f_1 f_2$ is not an edge of F (and thus L), by Observation 2.6.41, $F \cup Q$ contains a K_5-model which we find and return.
   (d) If $f_1 f_2$ is an edge of F,
      i. replace this edge by Q in F to obtain a new model of L in G that contains a subdivided edge (with non-empty interior).

ii. Apply the algorithm in case 1 (i.e., steps 1a), 1b) and 1c)).

The last two algorithm presented both run in linear time since they only need to find a constant number of paths (using DFS) and a constant number of checked (which can all be done in constant time once the paths are built).

2.6.10 Bounded treewidth and linear time algorithms

Many NP-hard problems can be solved in linear time on graphs whose treewidth is bounded by a fixed constant k (typical running times are super-exponential in k). See for example [61, 3, 6, 7]. Such algorithms require a Robertson-Seymour tree decomposition of width k as input. Bodlaender [8] showed if G has bounded treewidth then we can find an optimal tree decomposition of G in linear time.

In fact, Courcelle [16, 15, 17] showed that all problems that can be formulated in *monadic second order logic* can be solved in linear time. His method relies on evaluating a large number of formulas (this number depends only on k) at each node of the Robertson-Seymour tree decomposition (for the graph corresponding to the tree rooted at that node) and uses dynamic programming.

For example, we can formulate the problem of determining if X is an independent set in monadic second order logic as follows

$$\text{Indep}(X) = \forall x \in X, \forall y \in X, \neg \,\text{Adj}(x, y)$$

Arnborg et al. [2] extended Courcelle's result to *extended monadic second order logic (EMS)*, which allows them to solve counting and optimization problems in polynomial time. This extension essentially allows weights on the vertices and edges of G and evaluations (for instance, of sum of these weights over a set). For a restricted class of formulas, known as linear extended monadic second order logic with uniform cost measure, they show that problems encoded using such formulas can be solved in linear time on graphs of bounded treewidth.

For example, we could have set weight 1 to every vertex in G and ask to maximize $|X|$ in the above formula in order to find the maximum independent set in G.

Their result is obtained using a linear time algorithm which transforms a labelled graph of bounded treewidth to a labelled binary tree and an algorithm to transform a monadic second order logic property for labelled graphs to a monadic second order logic property for labelled binary trees.

Courcelle et al. [18] then extended these results to graphs of bounded clique-width (another graph parameter). It should be noted that the hidden constant in the running time in these algorithms (as well as many other algorithms presented in this thesis) are very high compared to direct dynamic programming algorithms (see [31] for work on rendering such algorithm more practical by formulating problems in "datalog"). See [32] and [46] for surveys on more results on such algorithmic meta-theorems.

Monadic second order logic and extended monadic second order logic

We now provide the formal definition for monadic second order logic and extended monadic second order logic. We use notation and definitions from [2] as we need their results in other chapters of this thesis.

Definition 2.6.45. *Monadic second order logic* is a language which contains the propositional logic connectives $\wedge, \neg, \vee, \rightarrow, \leftrightarrow$ (with the usual meanings: and, not, or, implies, if, and only if, respectively) , individual variables $x, y, z, x_1, x_2, \ldots$, existential ($\exists$) and universal ($\forall$) quantifiers and predicates. Moreover, it contains set variables $X, Y, Z, U, X_1, X_2, \ldots$, the membership symbol $\in$, and it allows existential and universal quantification over set variables.

A (labelled) graph consists of an incidence relation (Adj), unary predicates to designate vertices (V) from edges (E) and unary predicates corresponding to vertex and edge labels.

Extended monadic second order logic further allows *evaluations* (vertex and/or edge weight functions denoted by f_j for $j = 1, \ldots, m$) over subsets definable in monadic second

order logic, in the form $\Phi(X_1, \ldots, X_\ell) \wedge \psi(|X_1|_1, \ldots, |X_\ell|_m)$, where $|X_i|_j$ is to be interpreted as $\sum_{x \in X_i} f_j(x)$, Φ is a monadic second order logic formula and ψ is an *evaluation relation*.

An *evaluation relation* is a propositional formula whose are sign conditions on polynomials built from $|X_i|_j$.

We can also replace the relation ψ by an objective function and ask for its extremal value:

$$\max_{\Phi(X_1,\ldots,X_\ell)} F(|X_1|_1, \ldots, |X_\ell|_m)$$

.

For *linear extended monadic second order logic*, we require the evaluations to be linear in the terms $|X_i|_j$ and that the evaluation relation is missing (i.e., ψ is identically true).

A graph property that can be expressed in monadic second order logic is called a *MS-property*. A problem which can be expressed in extended monadic second order logic is called a *EMS problem*. A problem which can be expressed in linear extended monadic second order logic is called a *linear EMS problem*.

We will use set equality and set operations as shorthand for their corresponding monadic second order logic formulas. We will sometimes use $v \in V(G)$ and $e \in E(G)$ to mean the corresponding formulas using the graph's predicates.

We are now ready to formally state the theorems introduced in the previous section.

Theorem 2.6.46. *[16, 15, 17] For each MS-property P, and for each class K of graphs of treewidth universally bounded by k, the problem $P(G)$ for $G \in K$ can be decided in linear time, if G is given together with a tree decomposition of width at most k.*

Theorem 2.6.47. *[2] For each class K of graphs of treewidth universally bounded by k, every EMS linear extremum problem P can be solved in linear time with the uniform cost measure (where an arithmetic operation or comparison is charged constant cost), for $G \in K$, if G is given with a tree decomposition of width at most k.*

For all our applications, weights we give to vertices and edges are either 0 or 1 and objective functions are linear with coefficients 0 or 1 (which allows us to assume the uniform cost measure).

Dynamic programming examples

We give an example in this section and use a tree decomposition of bounded width to find an optimal colouring. In the next section, we revisit this example using monadic second order logic.

Remark 2.6.48. *A Robertson-Seymour tree decomposition $[T, \mathcal{G}]$ of G of width w actually gives us a chordal supergraph H of G. We obtain H by adding edges so each (graph) node G_t is a clique. Then $[T, \mathcal{G}]$ is the intersection version of a clique cutset tree of H.*

We can then use Algorithm 2.5.16 to obtain an optimal colouring of H. This is a $tw(G) + 1$ colouring of G, but may not be optimal.

We now describe a dynamic programming algorithm for colouring G. Our algorithm takes an input k and Robertson-Seymour tree decomposition of width $\text{tw}(G)$ and determines if G has a k-colouring. This algorithm runs in time linear in $|V(G)|, |E(G)|$ and exponential in k. To obtain an optimal colouring, we may try all values from $1, \ldots, tw(G) + 1$.

Algorithm 2.6.49.

Input:

- A graph G,
- a Robertson-Seymour tree decomposition of G of width $tw(G)$, and
- an integer k.

Output: "Yes" if G has a k colouring. "No" otherwise.

Running time: $O(|V(G)| + |E(G)|)$.

Basically, we wish to build the following lookup using dynamic programming. We arbitrarily root the tree at a node r. For each node t and each (proper) k-colouring c of its

label, our table contains a boolean value which indicates whether this colouring extends to (the graph induced by the union of nodes in) the subtree rooted at t.

There are at most n nodes and at most $(tw(G)+1)^k$ colourings of any particular node index G_t. So our final table contains $n(tw(G)+1)^k$ entries. We fill in the entries for the leaves first (by iterating over all colourings in each leaf and checking that it is proper) and proceed in a post-order traversal.

To fill an entry for an internal node, we check if this colouring can extend to each of its children (by looking at the corresponding entries in the table for its children). We can in fact build a Robertson-Seymour tree decomposition where all internal nodes have degree 3 and thus 2 children. This allows us to look up at most $(tw(G)+1)^{2k}$ entries.

Definition 2.6.50. A *canonical Robertson-Seymour tree decomposition* is a Robertson-Seymour tree decomposition where each internal node has degree three.

Remark 2.6.51. *We can obtain a canonical Robertson-Seymour tree decomposition (of the same width) from any Robertson-Seymour tree decomposition in linear time.*

We now describe Algorithm 2.6.49.

Description and analysis.

1. $k \geq tw(G)+1$, return "Yes"
2. Arbitrarily root T are r and compute a postorder of $V(T)$ with respect to this root.
3. For t in postorder$(V(T))$,
   (a) Build a list L_t of all proper colourings of (G_t) by iterating over all $k^{|V(G_t)|}$ colourings of G_t and checking it is proper.
   (b) For c in L_t,
      i. For each child s of t, for each colouring c' of G_s which agrees with c on $G_t \cap G_s$, check if Table$[s][c']$ is true.
      ii. If for every child s, there is such a colouring c', set Table$[t][c]$ to true (this includes the case where t has no children).
      iii. Otherwise, set Table$[t][c]$ to false.

4. Check if Table(G_r, c) is true for some proper colouring c in L_r. If so, return "Yes". If not, return "No".

□

Monadic second order logic example

We can easily formulate the k-colouring in monadic second order logic by first defining a predicate to test if a partition is a colouring and then a predicate to test if a set of k sets is a partition.

$$\begin{aligned} k-\text{Col}(X_1, \ldots, X_k) &= \text{Indep}(X_1) \wedge \ldots \wedge \text{Indep}(X_k) \\ k-\text{Part}(X, X_1, \ldots, X_k) &= (X = X_1 \cup \ldots \cup X_k) \\ &\quad \wedge (X_1 \cap X_2 = \emptyset) \wedge (X_1 \cap X_3 = \emptyset) \wedge \ldots \wedge (X_{k-1} \cap X_k = \emptyset) \end{aligned}$$

The monadic second order logic sentence for k-colouring is then

$$\exists X_1, \ldots X_k k-\text{Col}(X_1, \ldots, X_k) \wedge k-\text{Part}(V(G), X_1, \ldots, X_k)$$

We now state some more useful predicates in monadic second order logic which we will need later. Adjacent determines if there is an edge between two sets of vertices X and Y. Conn(X) determines if a set of X induces a connected subgraph (specified as a predicate).

$$\begin{aligned} \text{Adjacent}(X, Y) &= \exists x \exists y \, (x \in X \wedge y \in Y \wedge \text{Adj}(x, y)) \\ \text{Conn}(Z) &= \forall X \forall Y 2-\text{Part}(Z, X, Y) \rightarrow \text{Adjacent}(X, Y) \end{aligned}$$

A final formula we need, $H-$model, tests if a graph induced by X contains an H-model. We assume the vertices of H are labelled $1, \ldots, |V(H)|$.

$$\begin{aligned} \text{H}-\text{model}(X) &= \exists X_0, X_1, \ldots, X_{|V(H)|} \, \text{Conn}(X_1) \wedge \ldots \wedge \text{Conn}(X_{|V(H)|}) \\ &\quad \wedge (|V(H)| + 1)-\text{Part}(X, X_0, X_1, \ldots, X_{|V(H)|}) \\ &\quad \wedge \wedge_{i,j,ij \in E(H)} \text{Adjacent}(X_i, X_j) \end{aligned}$$

Table 2–1: Table of different trees introduced in this section

Name	$\mathcal{F}$	Requirement on G	Type	Unique cutting nodes	Unique tree	Note
block tree	all 1-cuts	connected	block	yes	yes	
	all 1-cuts	connected	recursive	yes	no	
	all 1-cuts	connected	intersection	yes	no	
2-block tree	all 2-cuts	2-connected	block	yes	no	
tree of split components	all strong 2-cuts	2-connected	recursive	yes	no	allows multi graphs as in dices
tree of split components before merging	all 2-cuts	2-connected	recursive	no	no	allows multi graphs a indices an non-unique only due t triangulation of cycles
SPQR tree	all strong 2-cuts	2-connected	intersection	yes	yes	allows multi graphs a indices an (graph) node are labelle by S,P,Q,I depending o their index
clique cutset tree	all minimal clique cutsets	chordal	block	yes	yes	all grap nodes are als cliques
clique cutset tree	all clique cutsets		block	no	no	
binary decomposition tree	all clique cutsets		recursive	no	no	
Robertson-Seymour tree decomposition			weak intersection	no	no	
Robertson-Seymour tree decomposition of bounded width w			weak intersection	no	no	all grap nodes indice have at mos w vertices
$(3,3)$-block tree	all $(3,3)$-cuts	3-connected	block	yes	yes	

CHAPTER 3
Recognizing K_5-minor free graphs

In this chapter, we present a linear time algorithm which determines if an input graph G contains a K_5-minor and outputs a K_5-model if it does. If G is K_5-minor free, it outputs a special tree decomposition as a certificate.

We discuss how this tree decomposition can be used to design algorithms in Chapter 5.

Our tree decomposition is derived from Wagner's theorem, which we now recall.

Theorem 3.0.52 (Wagner's theorem for K_5)**.** *A graph G either*

- *contains a K_5-minor,*
- *is planar,*
- *is the graph L,*
- *contains a 0-cut,*
- *contains a 1-cut,*
- *contains a 2-cut, or*
- *contains a $(3,3)$-cut.*

Also recall that a $(3,3)$-cut is a 3-cut which decomposes G into at least 3 components.

Definition 3.0.53. We call a cut of size i that splits G into at least j components an *(i,j)-cut.*

We first describe a simple polynomial time, but not linear time algorithm that does not use tree decompositions.

By Wagner's theorem, it is easy to determine if a graph G without 0-cuts, 1-cuts, 2-cuts or $(3,3)$-cuts is K_5-minor free. We only need to check if G is L (using brute force) and if G is planar (using one of the linear time planarity testing algorithms [40, 20, 48, 63, 12, 35] or

the algorithm described in Section 2.2.3) as G has a K_5-minor if and only if it is not L and non-planar.

By Corollary 2.6.22, if we find a smallest cut X of one of the three types (0-cut, 1-cut, 2-cut or $(3,3)$-cut) then we can decompose on X and recursively determine if each decomponent of $G - X$ has a K_5-minor in order to determine if G has a K_5-minor (G has a K_5-minor if and only if a decomponent does).

Finding a 0-cut, 1-cut, 2-cut or $(3,3)$-cut in G (if it exists) is also easy as we can simply try and delete every subset of $V(G)$ of size at most 3. Thus, we can use the above recursive algorithm to determine if an input graph contains a K_5-minor.

Using tree decomposition, we see that the algorithm is simply

1. building block tree,
2. building a 2-block tree for each block,
3. building a recursive tree for each 2-connected component using $(3,3)$-cuts of those components, and
4. testing if all resulting graph nodes are either planar or isomorphic to L.

As discussed in Sections 2.1 and 2.2, the block trees and 2-block trees of its graph nodes can be built in linear time. We name the last type of trees $(3,3)$-block trees.

Definition 3.0.54. A $(3,3)$-*block tree* of a graph G is an $\mathcal{F}$-block tree where $\mathcal{F}$ is the family of $(3,3)$-cuts of G.

To return a K_5-minor of G if a K_5-minor is found in a graph node, we simply run Algorithm 2.6.23 and 2.6.17.

Since testing for planarity and isomorphism to L can also be done in linear time, except for step 3, our algorithm runs in linear time. So we only need to focus on the case where G is 3-connected for the remainder of this section. More formally, our main result is an algorithm with the following specifications.

Algorithm 3.0.55 (K_5-minor recognition)**.**

Input: A 3-connected graph G

Output: Either

- a K_5-model of G if G has a K_5-minor, or
- a $(3,3)$-block tree of G otherwise.

3.1 $(3,3)$-block trees for K_5-minor free graphs

In this section, we give an overview of the main result of this chapter, Algorithm 3.0.55, a linear time algorithm which finds a $(3,3)$-block tree for K_5-minor free graphs. If the input contains a K_5-minor, it instead returns a certifying K_5-model. As discussed in the previous section, we only need to focus on input graphs that are 3-connected.

In each iteration, our recursive algorithm either finds a K_5-model or recursively applies itself on a set of smaller 3-connected graphs. Having solved the problem on these smaller graphs, it uses the solution to these new problems to construct the solution to the old problem quickly. Our reductions will reduce the problem so much that the entire algorithm runs in linear time provided the reduction and construction subroutines take linear time.

Some of our reductions are straightforward. For example suppose the graph has a large stable set S of vertices of degree 3, each of which has the same neighbourhood (in G) as at least 4 vertices in $V(G) - S$. In this case, deleting the vertices of S does not significantly change the structure of the $(3,3)$-block tree. Indeed having constructed a tree for the smaller graph $G - S$, to obtain a tree for G we note that the neighbourhood of each deleted vertex will be a $(3,3)$-cut in $G-S$, and for each such vertex v, we need to add a pendant graph node which is a clique on $\{v\} \cup N(v)$ adjacent to the unique cutting node of the tree decomposition of $G - S$ containing $N(v)$. Note that the reduced graph is 3-connected. On occasion we reduce via the deletion of a large set S of vertices of this type.

Another reduction involves finding a laminar set $\mathcal{F}$ of 3-cuts of G such that the $\mathcal{F}$-block tree has at least εn graph nodes. We then decompose (on cuts in $\mathcal{F}$) into the set of graph nodes of this tree. We note that these cuts need not be $(3,3)$-cuts but they will be such that

(a) given a K_5-model in one of the smaller graphs we can quickly find one in G and

(b) we can paste $(3,3)$-block trees of the smaller graphs together into a $(3,3)$-block tree for G in linear time.

Our last reduction involves contracting a matching M in G with a linear number of vertices to obtain a new 3-connected graph H. Since minor containment is transitive (Remark 2.6.15), if H contains a model of K_5, so does G and we can uncontract to find this model quickly. Obtaining our output for G given a $(3,3)$-block tree for H is much more complicated and forms the major part of this chapter. To do so, we need to insist that M has some special properties. To wit, M must be induced and only use vertices whose degrees in G are small.

This allows us to massage the $(3,3)$-block tree of H to obtain a $(3,3)$-block tree of G. Our first step is to identify where the $(3,3)$-cuts of G lie with respect to the $(3,3)$-block tree of H. It turns out that each of them either corresponds to a $(3,3)$-cut of H or corresponds to a triangle of H or to certain P_3s of H which have very special properties. This allows us to refine our decomposition tree so as to obtain an $\mathcal{F}$-block tree for a set $\mathcal{F}$ containing all $(3,3)$-cuts of G. We then process this tree, modifying it so as to obtain a $(3,3)$-block tree for G unless a K_5-model of G is found in the process.

Finally, we note that we can stop our recursion whenever we find a K_5-model by simply returning it. The following three subroutines will be useful in doing so.

Algorithm 3.1.1. [58]

Input: A graph F with $|E(F)| \geq 64|V(F)|$.

Output: A K_5-model of F.

Running time: $O(|V(F)|)$ (if we can obtain the first $64|V(F)|$ edges without reading the remaining edges)

We also use the algorithmic version of Wagner's theorem discussed in Section 2.6.9, whose specification we now recall.

Algorithm 2.6.39.

Input: A non-planar 3-connected graph G with more than 8 vertices and no $(3,3)$-cut.

Output: A K_5-model of G.

Running time: $O(|V(G)| + |E(G)|)$..

We refer to the operation in Algorithm 2.6.24 as *uncontracting a K_5-model* whose specification we now recall.

Algorithm 2.6.24.

Input:

- A graph G,
- a model H in G, and
- a K_5-model in H.

Output: A K_5-model in G.

Running time: $O(|V(G)| + |E(G)|)$.

This completes our informal description of our algorithm.

3.2 Formal description

Having described the algorithm informally we are now ready to present a formal specification. Throughout the chapter, $d = 2000000$ and $\varepsilon = d^{-6}$. Our algorithm uses a number of subroutines with these specifications. We label them to indicate where they can be found in this chapter.

The first subroutine is used to determine which reduction to make.

Algorithm 3.4.1.

Input: A 3-connected graph G of minimum degree at least 3 with at most $64|V(G)|$ edges.

Output: One of the following.

(O1) A stable set T of at least $5\varepsilon|V(G)|$ vertices of degree 3 such that for each $v \in T$, there exists 4 vertices in $V(G) - T$ that have degree 3 (in G) and have exactly the same neighbours (in G).

(O2) A matching M of size at least $d^4\varepsilon|V(G)|$ in G, all of whose vertices have degree at most d in G.

(O3) A K_5-model of G.

Running time: $O(|V(G)|)$.

The remaining subroutine will help handle each reduction.

Algorithm 3.4.2.

Input:

- A 3-connected graph G, and
- a matching M of size at least $d^4\varepsilon|V(G)|$ in G, all of whose vertices have degree at most d in G,

Output:

- an induced matching N with at least $\varepsilon|V(G)|$ edges, all of whose vertices have degree at most $2\varepsilon^{-1}$ in G, such that contracting the edges of N in G yields a 3-connected graph, or
- a family $\mathcal{F}$ of at least $\varepsilon|V(G)|$ laminar 3-cuts of G and a matching M'' such that for each X in $\mathcal{F}$, $G[X]$ contains an edge of M''.

Running time: $O(|V(G)| + |E(G)|)$..

Algorithm 3.5.1.

Input:

- A 3-connected graph G,
- a stable set T of degree 3 vertices of G such that for each vertex v in T, there exists 4 vertices of $V(G) - T$ of degree 3 (in G) with the same neigbhours as v, and
- a $(3, 3)$-block tree of $H = G - T$.

Output: A $(3,3)$-block tree of G.

Running time: $O(|V(G)| + |E(G)|)$..

Algorithm 3.5.5.

Input:

- A 3-connected graph G,
- an induced matching M, all of whose vertices have degree at most d in G, in G such that $H = G/M$ is 3-connected, and
- a $(3,3)$-block tree of H.

Output:

- A K_5-model of G, or
- a $(3,3)$-block tree of G.

Running time: $O(|V(G)|)$.

Algorithm 3.4.6.

Input:

- A 3-connected graph G, and
- a family $\mathcal{F}$ of at least $\varepsilon|V(G)|$ laminar 3-cuts of G and a matching M'' such that for each X in $\mathcal{F}$, $G[X]$ contains an edge of M''.

Output: An $\mathcal{F}$-block tree of G.

Running time: $O(|V(G)| + |E(G)|)$..

Algorithm 3.5.4.

Input:

- A 3-connected graph G,
- a set $\mathcal{F}$ of laminar 3-cuts of G each of which contains an edge, which decompose G into at least $\varepsilon|V(G)|$ pieces,
- an $\mathcal{F}$-block tree $[T, \mathcal{G}]$ of G, and
- a $(3,3)$-block tree of each graph in $\mathcal{G}$.

Output: A $(3,3)$-block tree of G.

Running time: $O(|V(G)| + |E(G)|)$..

Having completely specified our subroutines, we now describe the main algorithm of this chapter.

Algorithm 3.0.55.

Input: A 3-connected graph G.

Output:

- A K_5-model of G, or
- a $(3,3)$-block tree of G.

Running time: $O(|V(G)|)$.

Description: If G has fewer than 20 vertices we solve the problem by brute force, otherwise we proceed as follows.

1. Count the number of vertices in G. Count the number of edges in G. Stop counting if we reach $64|V(G)|$ edges and apply Algorithm 3.1.1 to find a model of K_5 in G and return it.
2. Run Algorithm 3.4.1.
   (a) If it returns **output (O1) (a set T of degree 3 vertices with many common neighbours)**,
      i. recurse on $H = G - T$.
      ii. If a K_5-model is returned, return that model.
      iii. If a $(3,3)$-block tree is returned, apply Algorithm 3.5.1 and obtain a $(3,3)$-block tree of G.
   (b) If **output (O3)** a K_5-model is returned, we return it.
   (c) Finally, **if output (O2), a large matching M,** is returned, apply Algorithm 3.4.2 (to G and M).
      i. If a K_5-model of G is returned, we return it.

ii. If a set of cuts is returned, apply Algorithm 3.4.6 which returns an $\mathcal{F}$-block tree of G. Recurse on all graphs nodes of this $\mathcal{F}$-block tree.

If any solution obtained recursively is a K_5-model, return the corresponding K_5-model in G.

If all solutions obtained recursively are $(3,3)$-block trees, apply Algorithm 3.5.4 and obtain a $(3,3)$-block tree of G.

iii. If a matching M is returned, contract the edges of M to obtain a graph H. Recurse on H.

If a K_5-model of H is returned, return its uncontraction.

If a $(3,3)$-block tree of H is returned, apply Algorithm 3.5.5. If a K_5-model is returned, return its uncontraction. Otherwise, we obtained a $(3,3)$-block tree of G.

3. In all cases, we obtain a $(3,3)$-block tree of G (if we have not already returned a K_5-model).
4. Check if all graph nodes of this $(3,3)$-block tree are L or planar [40].
5. If so, return this $(3,3)$-block tree.
6. If not, apply Algorithm 2.6.39 to a non-planar non-L graph node to obtain a K_5-model in that graph node. Return the corresponding K_5-model in G.

Analysis: Let $n = |V(G)|$. Since our subroutines run in linear time, each iteration of this algorithm runs in $O(cn)$ time for some large c (once we preprocess to check if the graph has $O(n)$ edges). We claim that this implies that the whole algorithm runs in $O(C(n-4))$ time on graphs of size at least 5 where $C = c\varepsilon^{-1}$. If we recurse by contracting a matching or deleting vertices of degree 3, the new subproblem has size at most $n(1-\varepsilon)$ and the bound follows easily by induction. Suppose then that we decompose G on a family $\mathcal{F}$ of 3-cuts into subproblems. Any subproblem of size 4 is already solved as the graph is a K_4 so we need not spend time on them. Hence every considered subproblem has size at least 4, so if there

are k subproblems of total size N then the total time spent is $CN - C4k + cn$. Now the subproblems are disjoint except on the 3-cuts so $n-3 = N-3k$ and $k > \varepsilon n$ so $Ck > cn$ and we are again done by induction. We have that the time spent is inductively at most Cn.

To complete our description of the algorithm, it remains to fill in the details of the subroutines whose specification have been set out in this section. We describe all pre-recursion subroutines in Section 3.4 and all post-recursion subroutines in Section 3.5. Before doing so we analyze the small cutsets in a graph and how they can interact with the small cutsets in a minor obtained from it by contracting a matching in Section 3.3.

3.3 The Structure of 2-cuts and 3-cuts

We need to examine the structure of 2-cuts and 3-cuts more closely. In this section, we first introduce the $(3,3)$-block tree, an $\mathcal{F}$-block tree where $\mathcal{F}$ is the family of all $(3,3)$-cuts, and show that if a 3-connected graph G is not $K_{3,3}$ then its $(3,3)$-block tree is unique. Then we examine the ways in which the 2-cuts of a 2-connected graph can interact. Third, we determine the correspondence between cutsets of a 3-connected graph G and a 3-connected graph H obtained from it by contracting an induced matching M.

3.3.1 The $(3,3)$-block tree is unique

We note that a $K_{3,3}$ has two $(3,3)$-cuts, its two stable sets of size three. Decomposing on either of these cuts yields three K_4s, which cannot be decomposed any further. Thus, a $K_{3,3}$ has two $(3,3)$-block trees, each of which has only one cutting node even though the graph has two $(3,3)$-cuts. These two $(3,3)$-block trees would be isomorphic if it were not for the requirement that the graph labelling a node is itself labelled by vertices of G. In this section, we show that every 3-connected graph G which is not a $K_{3,3}$ has a unique $(3,3)$-block tree which has a cutting node corresponding to each of its $(3,3)$-cuts.

To do so, we show that the $(3,3)$-cuts of any $G \neq K_{3,3}$ are laminar and then only need to apply Theorem 2.3.15.

Lemma 3.3.1. *Let F be a 3-connected graph that is not $K_{3,3}$. Let X be a $(3,3)$-cut in F. There does not exist a $(3,3)$-cut separating X.*

Proof. Suppose to the contrary that Y is a $(3,3)$-cut separating X. Since F is 3-connected, every component of $F-X$ contains a neighbour of every vertex of X and thus every component of $F-X$ intersect Y. Since there are at least three such components, we see that the vertices of Y are in different components of $F-X$ and there are exactly three components of $F-X$. Thus X also separates Y and the same argument shows that the vertices of X are in different components of $F-Y$. Thus, X and Y are disjoint and each component of $F-X-Y$ has edges to at most one vertex of X and at most one vertex of Y. Since F is 3-connected, there are no such components and F is a $K_{3,3}$. □

Sometimes, we will also want to decompose on 3-cuts containing an edge. We now show that if G is 3-connected and $G \neq K_{3,3}$, adding a laminar family of such cuts to a family of $(3,3)$-cuts of G yields a laminar family of cuts of G.

Lemma 3.3.2. *Let X be a $(3,3)$-cut of a 3-connected graph F . There does not exist a 3-cut Y separating X such that $F[Y]$ contains an edge.*

Proof. Suppose to the contrary that Y is a 3-cut separating X such that $F[Y]$ has an edge. Since $F[Y]$ has at most two components, Y fails to intersect some component K of $F-X$. Thus $X-Y$ is contained in one component of $F-Y$ (which also contains K), a contradiction. □

Lemma 3.3.3. *Let F be a 3-connected graph . Let X be a 3-cut in F which contains an edge. There does not exist a $(3,3)$-cut separating X.*

Proof. Suppose to the contrary that Y is a $(3,3)$-cut separating X. Then Y must contain a vertex of each component of $G-X$. Thus, X separates Y, contradicting Lemma 3.3.2. □

Lemma 3.3.4. *If we decompose a 3-connected graph $G \neq K_{3,3}$ on a 3-cut X which is either a $(3,3)$-cut or contains both endpoints of some edge then, for any other $(3,3)$-cut Y of G, there is exactly one decomponent of $G - X$ which completely contains Y.*

By Theorem 2.3.15, the $(3,3)$-block tree of a 3-connected graph that is not $K_{3,3}$ is unique.

Corollary 3.3.5. *Every 3-connected graph $G \neq K_{3,3}$ has a unique $(3,3)$-block tree.*

In fact, we've shown the following more general statement.

Corollary 3.3.6. *Every 3-connected graph $G \neq K_{3,3}$ has a unique $\mathcal{F}$-block tree where $\mathcal{F}$ is the union of all $(3,3)$-cuts of G and a family of laminar 3-cuts, each of which contains an edge.*

3.3.2 The Structure of 2-cuts

For our algorithm, we need to use more properties of the 2-block tree than those discussed in Section 2.2.

Recall that a 2-connected graph may have different 2-block trees. For example, each triangulation of the 6-cycle gives a different 2-block tree. We can obtain all 2-block trees of a graph G from its SPQR tree SPQR(G) by triangulating each cycle (series node) in the SPQR tree. In other words, the 2-block tree only changes depending on the triangulation chosen.

There is only one triangulation for a cycle of length 3 so we focus on series nodes containing cycles of length at least four in this section.

We refer to a cycle indexing a series node of SPQR(G) of length least four as an *series node cycle.*

We note also that the number of leaves in the 2-block tree changes depending on this choice of triangulation(s). We will sometimes want to build a special 2-block tree which typically has more leaves than the other 2-block trees for the same graph.

Definition 3.3.7. A 2-block tree is *special* if it is obtained from SPQR(G) by triangulating each series node cycle C of length at least sixcorresponding to graph nodes within it as follows. Let $C = c_1, \ldots, c_k$. Split C along the $\lfloor k/2 \rfloor$ cuts $c_1c_3, c_3c_5, c_5c_7, \ldots$ (ending with either $c_{k-2}c_k$ if k is odd or $c_{k-1}c_1$ if k is even). Triangulate the remaining (smaller) "interior" cycle in any way. Triangulate series node cycles of length four in any way.

We can build a special 2-block tree for G in linear time by first building SPQR(G) and then triangulating the series node cycles as described above. We now deduce some useful properties of this tree.

The first property holds for all 2-block trees. Recall that a virtual edge in an SPQR tree is a 2-cut that is decomposed on (they corresponding to cutting nodes in a 2-block tree).

Lemma 3.3.8. *The number of cutting nodes in a 2-block tree of a 2-connected graph H is at least one quarter of the number of vertices in the 2-cuts of H.*

Proof. Let C_1 be the set of virtual edges of SPQR(G), and C_2 be the set of series node cycles of length at least four. The number of cutting nodes of a 2-block tree for H is $|C_1| + \sum_{K \in C_2}(|K| - 3)$. Every vertex of H in a 2-cut is either in some element of C_1 or on some element of C_2. Thus, the number of such vertices is at most $2|C_1| + \sum_{K \in C_2} |K|$. Since each series node cycle of C_2 has size at least four, the result follows. □

The second property holds because we have attempted to maximize the number of leaves of the 2-block tree.

Lemma 3.3.9. *Let H be a 2-connected graph and T a SPQR tree of H. If $S \subseteq V(H)$ then either*

1. *T has at least $|S|/15$ leaves,*
2. *the total number of vertices in series node cycles of T minus three times the number of series nodes of T is at least $|S|/15$, or*

3. *there is a subset $S' \subseteq S$ of vertices of size at least $|S|/15$ no two of which are in the same series node cycle of T and no two of which are in a 2-cut.*

Proof. Greedily pick a maximal subset S' of S with property (3). If $|S'| \geq |S|/15$ then we return S' (as output 3). We cannot add $s \in S \setminus S'$ to S' for one of two possible reasons. Either s is in the same series node with at least 4 vertices as some vertex of S', or s is in a virtual edge with some of S'.

If at least half of $S - S'$ are in series nodes with at least 4 vertices then T contains at least $|S - S'|/4 \geq 14|S|/(4 \cdot 15)$ vertices in excess of 3 in all of its serial nodes. Thus, we can output (2).

If at least half of $S - S'$ are in virtual edges (and not serial nodes with at least 4 vertices) then either T has $|S|/15$ leaves or T has $|S|/15$ disjoint paths of length at least 7, each of which contains only nodes of degree 2 in T (since T has at most $|S|/15$ internal nodes of degree at least 3 but at least $7|S|/15$ nodes) and which begins and ends with a cutting node.

In the first case, we output (1). In the second case, for each path, we pick a vertex in one of the middle cutting node that does not appear in the end cutting node of the path. Thus, we picked at least $|S|/15$ vertices. No two of these vertices form a 2-cut as there exists a path in H between the vertices of the end cutting node of each path that does not use the vertex chosen in that path (and the at least $|S|/15$ paths are disjoint). □

Lemma 3.3.10. *Let H be a 2-connected graph and S a subset of vertices of H. If there is a special 2-block tree for H with at most $|S|/15$ leaves then there is a subset S' of S of size at least $|S|/15$ no two vertices of which form a 2-cut of H.*

Furthermore, given a special 2-block tree with at most $|S|/15$ leaves, we can find such an S' in linear time.

Proof. Suppose there is a special 2-block tree with at most $\frac{|S|}{15}$ leaves. We remove from S all vertices which are contained in series node cycles of length $k > 5$, or are in a 2-cut both of whose vertices are contained in a rigid node with $d > 2$ 2-cuts.

The number of leaves in a tree T with at least 2 vertices is

$$\sum_{v \in V(T)} \max(d(v) - 2, 0) + 2.$$

For a special 2-block tree, the triangles of the triangulation of series node cycles of length $k > 5$ contribute at least $\lfloor \frac{k}{2} \rfloor - 2 \geq \frac{k}{7}$ to this sum. Each 3-connected graph node (R node) of SPQR(H) which contains the vertices of $d > 2$ 2-cuts contributes $d - 2 > \frac{2d}{7}$ to this sum (and contains at most $2d$ vertices in 2-cuts). So the number of vertices in these two types of nodes is at most $\max(7, 7) = 7$ times the number of leaves which we know is at most $\frac{|S|}{15}$.

Thus, in our first pass, we remove at most $\frac{7|S|}{15}$ vertices. We also remove the vertices of S which are in no 2-cut. Clearly, we can assume we remove at most $\frac{|S|}{15}$ more vertices as otherwise we can return this set. We let T be the remaining set of at least $\frac{7|S|}{15}$ vertices of S. It is easy to find T in linear time given our special 2-block tree.

We note that any rigid node of SPQR(H) which contains a vertex x in a 2-cut contains another vertex y such that x, y is a 2-cut in the SPQR tree and hence in a cutting node of our special 2-block tree. Thus, no graph node of SPQR(H) contains more than 5 vertices of T. We join two vertices of T if the subtrees of nodes of SPQR(H) containing them intersect. This yields a chordal graph F of maximum clique size at most 5. This graph has a 5-colouring and hence has a stable set of size at least $\frac{|T|}{5} \geq \frac{7|S|}{15 \cdot 5} \geq \frac{|S|}{15}$.

This is the desired S'. Indeed, no two vertices in S' are in a 2-cut since otherwise, they appear in a series node cycle or a strong 2-cut (and thus appear in at least two graph nodes).

We use Algorithm 2.5.16 to find such a colouring and stable set S'.

□

The third result focuses on graphs whose 2-cuts have a special property.

Lemma 3.3.11. *Let H be a 2-connected graph for which we can 2-colour the set S of vertices in 2-cuts so that every 2-cut has a vertex of each colour. Then given a 2-block tree for H we can find within it, in linear time, either*

(i) $\frac{|S|}{50}$ *leaves, or*

(ii) $\frac{|S|}{50}$ *disjoint paths, each with 11 nodes, all of which begin and end with a cutting node and contain only vertices which have degree 2 in the tree.*

Proof. We can assume that H is not just a 4-cycle. We note that any two non-consecutive vertices in a series node cycle of SPQR(H) form a 2-cut. Thus, our 2-colouring implies that the SPQR tree contains only series node cycles of length (at most) 4. Furthermore, since H is not just a 4-cycle, any such 4-cycle contains a 2-cut of the SPQR tree (i.e. a strong 2-cut).

When we triangulate such a 4-cycle to make the 2-block tree, we add one more 2-cut using two of these four vertices. Any vertex in S which is not in one of these 4-cycles is in a cutting node of the SPQR tree. It follows that the total number of cutting nodes of the 2-block tree is at least $|S|/2$. We delete all of the vertices of the 2-block tree which are graph nodes which contain the endpoints of at least 3 2-cuts along with all cutting nodes which are adjacent to at least 3 graph nodes. If we delete more than $|S|/50$ vertices then the 2-block tree has more than $|S|/50$ leaves and we are done. So, we are left with a set of disjoint paths of the 2-block tree, every vertex of which has degree 2 in the tree. If the sum of the degrees of the deleted vertices is more than $3|S|/50$ then again the block tree has $|S|/50$ leaves and we are done. Thus, there are at most $3|S|/50$ paths, and they must contain in total at least $25|S|/50 - |S|/50 = 24|S|/50$ cutting nodes. We traverse each of the paths, repeatedly splitting off a path with 11 nodes, which begin and end with a cutting node, until there are only 5 cutting nodes left on the path. We build a set of at least

$$\frac{\frac{24|S|}{50} - 5\frac{3|S|}{50}}{6} \geq \frac{|S|}{50}$$

paths, which we return. □

3.3.3 Cutset Structure Relative to a Contracted Matching

In this section, we study the correspondence between the cuts in a 3-connected graph G and those in the graph H obtained by contracting an induced matching M in G.

To ease notation, we use f to refer to the function from $V(G)$ to $V(H)$ corresponding to the contraction of the edges of M. If v is in no edge of M, $f(v) = v$. If v is in an edge of M, $f(v)$ is the vertex to which it is contracted. So $V(H) = f(V(G))$ and $V(G) = f^{-1}(V(H))$.

Note that if Y is a cut in H then $f^{-1}(Y)$ is a cut in G. Therefore since M is a matching and G is 3-connected, H is 2-connected.

In the other direction, if X is a 3-cut of G but $f(X)$ is not a cut in H, then for some component U of $G - X$ we must have that $f(U) \subseteq f(X)$. If U has more than one vertex then there is an edge uv of U. This is not an edge of M as otherwise $f(U) \not\subseteq f(X)$. Since M is induced, we can therefore assume without loss of generality that $f(u) = u$. But, again, we contradict $f(U) \subseteq f(X)$. So U is a vertex u. Since $f(U) \subseteq f(X)$, we know that xu is an edge of the matching M for some vertex x of X. Since M is induced it follows that $f(v) = v$ for every other neighbour v of u and hence $N(u) \subseteq X$. Since u has at least three neighbours, it follows that u has exactly three neighbours x, y, z in G and $X = \{x, y, z\}$. Thus the only cutsets of size 3 in G whose image in H is not a cutset are $N(u)$ for vertices u of degree 3 which are in the matching and for which $G - u - N(u)$ is connected.

We are actually more interested in which $(3, 3)$-cuts X of G do not correspond to $(2, 3)$-cuts or $(3, 3)$-cuts of H. If this is to happen then for some component U of $G - X$, $f(U) \subseteq f(X)$.

Mimicking the arguments above, we see that this occurs precisely if $U = \{u\}$ and $X = N(u)$ for some degree 3 vertex u in the matching and for which $G - u - N(u)$ has exactly two components. Note that in this case $f(X)$ is a $(3, 2)$-cut of H which induces a P_3 (i.e., an induced path with 3 vertices) or a triangle.

In summary then, a $(3,3)$-cut X of G is of one of the following types:

1. $f(X)$ is a $(2,3)$-cut of H,
2. $f(X)$ is a $(3,3)$-cut of H,
3. $f(X)$ is $(3,2)$-cut of H which induces a connected subgraph of H, and $X = N(v)$ for some vertex v of degree 3 in G which is in M.

3.4 Reductions

In this section, we present and prove the correctness of the subroutines we apply to reduce our problem to a set of smaller problems.

In Section 3.4.1 we provide the details of Algorithm 3.4.1 which returns either a set T of vertices of degree 3 or a matching M. In Section 3.4.2 we describe Algorithm 3.4.2 which we apply if Algorithm 3.4.1 returns a matching M. It returns either a matching N or a family $\mathcal{F}$ of 3-cuts. Finally in Section 3.4.3 we describe how to recurse given T, N, or $\mathcal{F}$. Recursing on $\mathcal{F}$ requires an application of Algorithm 3.4.6 which builds the $\mathcal{F}$-block tree.

3.4.1 Finding a Matching or a Set of Special Degree 3 Vertices

In this section we give the details of Algorithm 3.4.1, which we now restate.

Algorithm 3.4.1.

Input: A 3-connected graph G of minimum degree at least 3 with at most $64|V(G)|$ edges.
Output: One of the following.

(O1) A set T of at least $5\varepsilon|V(G)|$ vertices of degree 3, each of which has the same neighbourhood as at least 4 vertices of $V - T$.

(O1) A matching M of size at least $d^4\varepsilon|V|$ in G, all of whose vertices have degree at most d in G.

1. A K_5-model of G.

Running time: $O(|V(G)|)$.

Description and analysis.

1. Check if G contains at least $64|V(G)|$ edges. If so, apply Algorithm 3.1.1 to G and return the K_5-model it outputs.
2. Build S, the set of vertices of degree at most d in G.
3. We find and delete a maximal matching M from $G[S]$ (see [43]).
4. Let $n = |V(G)|$.

   If M has more than $d^4\varepsilon n$ edges, we return M.
5. Otherwise, we proceed as follows.
   (a) The vertices of M along with the vertices of degree exceeding d form a hitting set B for the edges of G. Since G has at most $64|V(G)|$ edges and M has at most $2d^4\varepsilon n < \frac{2n}{d}$ vertices, it follows that B has at most $|B \cap S| + |B - S| \leq \frac{2n}{d} + \frac{128n}{d} = \frac{130n}{d}$ vertices. Build B.

   (b) $V - B$ is a stable set and every vertex of $V - B$ has at least three neighbours in B. For each of these vertices in turn, we try and choose a pair of its neighbours which has not yet been chosen for a vertex already considered. We can check this condition easily because the vertices of $V - B$ have bounded degree.

   (c) If we choose $64\frac{130n}{d}$ such pairs then they correspond to the subdivided edges of a subdivision of a high density graph and we apply Algorithm 3.1.1 to this high density minor and return the output K_5-model.

   (d) Otherwise, letting S^* be the set of vertices for which we have chosen a pair of neighbours, and F the graph with vertex set B and edge set the pairs we have chosen, we see that F has fewer than $64\frac{130n}{d}$ edges and $\frac{130n}{d}$ vertices.

   (e) We note that the neighbourhood of every vertex of $V - B - S^*$ forms a clique in F. Thus, if any such vertex has degree exceeding three then we have found a K_5-model, which we return.

   (f) So, the neighbourhood of every vertex of $V - B - S^*$ in F is a triangle. We determine if F is 64-degenerate.

(g) If F is not 64-degenerate,

i. we find a subgraph of F of minimum degree at least 64 and apply Algorithm 3.1.1 to obtain a K_5-model, which we return.

(h) If F is 64-degenerate,

i. we construct an order in which each vertex has at most 64 neighbours appearing earlier in the order.

ii. Note that every triangle of F consists of a vertex v and two adjacent neighbours of v earlier in the order. Thus, F has at most $\binom{64}{2}|B|$ triangles.

Since we failed to choose a pair for all vertices of $S' = V - B - S^*$, we can pick a subset $T \subseteq S'$ of at least $|S'| - 4\binom{64}{2}|B|$ vertices whose neighbourhood is shared with at least four other vertices of $S' - T$.

iii. We return T as output (1).

We claim T contains at least $5\varepsilon n$ vertices. Indeed, $T = V - B - S^* - (S' - T)$ so

$$\begin{aligned} |T| &\geq n - |B| - |S^*| - 4\binom{64}{2}|B| \\ &\geq n - \left(4\binom{64}{2} + 1\right)\frac{130n}{d} - 64\frac{130n}{d} \\ &= n - \frac{1056770n}{d} > \frac{n}{3} \geq 5\varepsilon n \end{aligned}$$

□

3.4.2 Massaging the Matching

In this section we describe and analyze Algorithm 3.4.2 which we apply if Algorithm 3.4.1 returns a matching M. In order to do so we use results about the 2-block trees of 2-connected graphs discussed in Section 2.2.

Algorithm 3.4.2.

Input:

- A 3-connected graph G, and
- a matching M of size at least $d^4\varepsilon|V(G)|$ in G, all of whose vertices have degree at most d in G,

Output:

- an induced matching N with at least $\varepsilon|V(G)|$ edges, all of whose vertices have degree at most $2\varepsilon^{-1}$ in G, such that contracting the edges of N in G yields a 3-connected graph, or
- a family $\mathcal{F}$ of at least $\varepsilon|V(G)|$ laminar 3-cuts of G and a matching M'' such that for each X in $\mathcal{F}$, $G[X]$ contains an edge of M''.

Running time: $O(|V(G)| + |E(G)|)$..

Outline

We outline here our approach for Algorithm 3.4.2 and provide details later in the section. We first find a reasonably large submatching M' of M such that if a vertex has neighbours on two edges of M' then it has degree at least 7. We contract M' to obtain a graph H. We note that H is 2-connected, since G is 3-connected. We build its 2-block tree $[T, \mathcal{H}]$. We apply Lemma 3.3.10 to H and S where S is the set of vertices edges of M' contracted to. How we proceed next depends on the structure of the outcome of this lemma.

- If the 2-block tree of H has many leaves, we look at the cutting nodes separating these leaves from the rest of H.

  If more than half of these cutting nodes correspond to three vertices of G, they form a set of laminar 3-cuts (in G), each of which contains an edge (of M'). We return this family of cuts.

  If more than half of these cutting nodes correspond to four vertices of G, we pick an appropriate edge from each leaf graph node adjacent to one of these cutting nodes.

We choose these edges so that they form a matching N in G and no two edges are in a 4-cut of G. We then use N to obtain either a submatching of N whose contraction leaves the graph 3-connected or find a family of laminar 3-cuts each of which contains an edge of N. We return this submatching or family of cuts.

- If there is a subset $S' \subseteq S$ of vertices of size at least $|S|/15$, no two of which are in a 2-cut of H, we construct a submatching N of M' consisting of the preimage of S'. Again, we obtain a submatching of N or a family of cuts as in the previous case. We return this submatching or family of cuts.

We now provide the necessary details.

Details

Lemma 3.4.3. *Given*

- *a graph G, and*
- *a matching M of size at least $d^4\varepsilon|V(G)|$ in G, all of whose vertices have degree at most d in G,*

we can construct in linear time an induced submatching M' of M of size at least $30d^2\varepsilon|V(G)|$ which is induced and such that no vertex of G of degree at most 6 has a neighbour in two distinct matching edges.

Proof. Since the maximum degree of the vertices in the matching M is at most d, we can greedily find a submatching M' of size at least $\frac{|M|}{12d} > 30d^2\varepsilon|V(G)|$ which is induced and such that no vertex of degree at most 6 has a neighbour in two distinct matching edges. □

Lemma 3.4.4. *Given*

- *a graph G,*
- *an induced matching M' of size at least $30d^2\varepsilon|V(G)|$ in G, all of whose vertices have degree at most d in G, and such that no vertex of degree at most 6 has a neighbour in two distinct matching edges, and*

- *a 2-block tree* $[T, \mathcal{H}]$ *of* $H = G/M'$

we can construct in linear time either

- *an induced matching* N *of* G*, all of whose vertices have degree at most* $2\varepsilon^{-1}$ *in* G*, with at least* $5\varepsilon|V(G)|$ *edges of which no two edges are in a 4-cut, or*
- *a family* $\mathcal{F}$ *of at least* $\varepsilon|V(G)|$ *laminar 3-cuts of* G *and a matching* M'' *such that for each* X *in* $\mathcal{F}$*,* $G[X]$ *contains an edge of* M''*.*

Proof. Let $n = |V(G)|$. Let $S = f(M)$ be the set of vertices obtained by contracting M. By Lemma 3.3.10 applied to H and S, either

1. T has at least $|S|/15$ leaves, or
2. there is a subset $S' \subseteq S$ of vertices of size at least $|S|/15$ no two of which are in a 2-cut of H.

In the second case, we return $f^{-1}(S')$ as N. Thus, we may assume T has at least $|S|/15 \geq (30d^2\varepsilon n)/15 > 40\varepsilon n$ leaves.

Since the 2-block tree for H has at least $40\varepsilon n$ leaves, it has at least $20\varepsilon n$ leaves L such that L has at most ε^{-1} vertices (since the graphs corresponding to these leaves are disjoint, except for the cut Y which separates L from the rest of H).

We distinguish two cases.

Case 1: If at least half of the leaves L of size at most ε^{-1}, $f^{-1}(Y)$ has size 3 where Y is the cut separating L from the rest of H then for each such L and Y, $f^{-1}(Y)$ is a 3-cut containing an edge separating L from a component of $G - f^{-1}(Y)$ containing the rest of G. We return this family $f^{-1}(Y)$ for each such Y and the submatching M'' of M' consisting the edge of M' in $f^{-1}(Y)$ for all such Y. This family of cuts contains at least $|S|/30 > \varepsilon n$ cuts and are laminar since the cuts in cutting nodes of T are laminar

Case 2: If at least half of the leaves L of size at most ε^{-1}, $f^{-1}(Y)$ has size 4 where Y is the cut separating L from the rest of H then we proceed as follows to choose an edge for each such L and Y. Since G is 3-connected and no vertex of degree at

most 6 in G has neighbours in two different matching edges we see that L has at least 7 vertices. We try to find two disjoint paths of L joining the two edges of $f^{-1}(Y)$. If we fail, we find a 1-cut separating them. This 1-cut forms a 3-cut with one of the edges in $f^{-1}(Y)$ separating some of $L - f^{-1}(Y)$ from the rest of G. But this 3-cut corresponds to a 2-cut in H, which is impossible. So we find two disjoint paths and we can also find an edge of $L - f^{-1}(Y)$ for which the deletion of its endpoints leaves a path between the two edges of $f^{-1}(Y)$ (as we can use an edge on one path unless they both have 3 vertices, in which case our degree condition guarantees that each midpoint is joined to some other vertex in L). We choose this edge (for this L and Y).

We return all chosen edges as N. Note that N is an induced matching of size at least $10\varepsilon n > 5\varepsilon n$ and no two edges of N form a 2-cut in G.

□

Lemma 3.4.5. *Given*

- *a graph G, and*
- *a matching N of G, all of whose vertices have degree at most $2\varepsilon^{-1}$ in G, with at least $5\varepsilon|V(G)|$ edges of which no two edges are in a 4-cut,*

we can construct in linear time either

- *a submatching N' of N with at least εn edges such that G/N' is 3-connected, or*
- *a family $\mathcal{F}$ of at least $\varepsilon|V(G)|$ laminar 3-cuts of G and a matching M'' such that for each X in $\mathcal{F}$, $G[X]$ contains an edge of M''.*

Proof. Let $n = |V(G)|$.

We build the 2-block tree $[T, \mathcal{H}]$ of $H = G/N$. Since G is 3-connected, every 2-cut and thus every cutting node of T contains a contracted edge of N (which is a vertex in H).

If T has at least εn cutting nodes then we uncontract each of the corresponding cuts to obtain a family $\mathcal{F}$ of 3-cuts, each of which contains an edge (of N). Since the cutting nodes of T are laminar, so are the 3-cuts in $\mathcal{F}$. We return this family of cuts.

If T has less than εn cutting nodes then by Lemma 3.3.8, the number of vertices of H that are in 2-cuts is at most $4\varepsilon n$. We remove from N any contracted edge of N in a 2-cut of H to build N'. G/N' is now 3-connected (since we eliminated all 2-cuts.). We return N'. □

We are now ready to describe Algorithm 3.4.2.

Description and analysis. (of Algorithm 3.4.2) Given G and M, we apply Lemma 3.4.3 (with the same input) to obtain a submatching M' of M. We build the 2-block tree $[T, \mathcal{H}]$ of $H = G/M'$.

We run Algorithm 3.4.4 on G, M' and $[T, \mathcal{H}]$. If we obtain a family $\mathcal{F}$ of cuts and matching M'', we return it. If we obtain a matching N, we apply Lemma 3.4.5 to G and N and return its output. □

3.4.3 Decomposing

In this section, we obtain the set of smaller problems to recurse on given one of the following structures in G (found in one of the previous sections).

1. A set of at least $5\varepsilon|V(G)|$ vertices of degree 3 which have the same neighbourhood as at least 4 other such vertices.
2. An induced matching N with at least εn edges such that contracting the edges of N in G yields a 3-connected graph.
3. A set of laminar 3-cuts of G each of which contains an edge, which decompose G into at least εn pieces.

If we obtain 1., we simply remove these degree 3 vertices and recurse on the resulting graph.

If we obtain 2., we contract this matching N and recurse on the resulting graph.

In the remainder of this section, we assume that we have 3. We provide the details of Algorithm 3.4.6 which we apply when Algorithm 3.4.2 finds a set of laminar 3-cuts, each of which contains an edge.

Algorithm 3.4.6.

Input:

- A 3-connected graph G, and
- a family $\mathcal{F}$ of at least $\varepsilon|V(G)|$ laminar 3-cuts of G and a matching M'' such that for each X in $\mathcal{F}$, $G[X]$ contains an edge of M''.

Output: An $\mathcal{F}$-block tree of G.

Running time: $O(|V(G)| + |E(G)|)$..

Theorem 3.4.7. *There is an algorithm with the specifications of Subroutine 3.4.6.*

Proof. Note that each cut of $\mathcal{F}$ is a 2-cut in G/M''. Furthermore, this family of corresponding 2-cuts is laminar since $\mathcal{F}$ is laminar.

Since the 3-cuts in $\mathcal{F}$ are laminar, the $\mathcal{F}$-block tree is well defined and unique. We add a triangle onto each cut in $\mathcal{F}$ to obtain G' which has the same $\mathcal{F}$-block tree as G by laminarity of the cuts in $\mathcal{F}$. This ensures that every cut of $\mathcal{F}$ maps to a 2-cut containing an edge (which is a strong 2-cut and appears in all 2-block trees). We then contract M'' to obtain H. We build the 2-block tree for H and then coarsen it by removing any cuts except the ones we are interested in (we can coarsen the 2-block tree by simply deleting all cutting nodes corresponding to a cut of $\mathcal{F}$, replacing each resulting component T_i of the tree by a single graph node indexed by the graph induced by the union of vertices in indices of nodes of T_i). This gives us an $\mathcal{F}'$-block tree $[T, \mathcal{H}]$ of the contracted graph where $\mathcal{F}'$ is obtained from $\mathcal{F}$ by contracting the edges in all cuts.

We then build an $\mathcal{F}$-block tree of G by uncontracting the edges in all nodes of the $\mathcal{F}'$-block tree of H. I.e., if $f : G \to H$ is the map for contracting M'' then we set $G'_t =$

$G'[f^{-1}(V(H_t))]$. We uncontract all nodes simultaneously. This yields an $\mathcal{F}$-block tree of G' which is an $\mathcal{F}'$-block tree of G.

□

After we apply Algorithm 3.4.6 to output 3. of Algorithm 3.4.2 and obtain an $\mathcal{F}$-block tree, we recurse on each graph node of this $\mathcal{F}$-block tree.

In fact, we apply a brute force algorithm on each graph node of size at most $2\varepsilon^{-1}$ to avoid recursing on them. Since $\mathcal{F}$ contains εn cutting nodes, it contains at least εn graph nodes so at least half the graph nodes have size at most $2\varepsilon^{-1}$.

Since the union of all graph nodes in the $\mathcal{F}$-block tree is $O(n)$ so the total running time for recursing on all graph nodes of size at most $2\varepsilon^{-1}$ is bounded by cn.

3.5 New Solutions from Old

In this section, we present the subroutines we use to construct a solution to an instance of our problem using the solution to the subproblems we reduced to.

3.5.1 Adding degree 3 vertices

In this section, we describe how to obtain a solution for G given a solution for a graph H obtained from G by removing a special set of degree 3 vertices. In other words, we provide the details of Algorithm 3.5.1, which we now restate.

Algorithm 3.5.1.

Input:

- A 3-connected graph G,
- a stable set T of degree 3 vertices of G such that for each vertex v in T, there exists 4 vertices of $V(G) - T$ of degree 3 (in G) with the same neigbhours as v, and
- a $(3,3)$-block tree of $H = G - T$.

Output: A $(3,3)$-block tree of G.

Running time: $O(|V(G)| + |E(G)|)$..

The algorithmic steps are simply as follows. We assume the list T is given as a list of triplets encoding the neighbourhood of each vertex v in T and a pointer to from the triplet to v.

Description and analysis.

- Arbitrary order on the vertices of G and sort the triplets (t_1, t_2, t_3) in T so that $t_1 < t_2 < t_3$.
- For each cutting node of the $(3,3)$-block tree of H, build a triplet (i_1, i_2, i_3) of its vertices with $i_1 < i_2 < i_3$. Call this new list T'.
- Merge the two lists of triplets, but mark the elements of T' before doing so.
- Radix sort the merged list.
- Now the triplets that have the same value for all coordinates occupy a contiguous region in the sorted list. Note that each region of equal triplets contains exactly one marked triplet (corresponding to a cutting node).

  Pass through the list once and put the marked triplets at the beginning of the region of equal triplets.

  For each unmarked triplet, build a graph node of the corresponding vertex with its neighbourhood. Add an edge between this graph node and the cutting node corresponding to the triplet of the same value (if we read through the list, this is the last marked triplet).

Return the $(3,3)$-block tree with these added nodes. □

3.5.2 Uncontracting a Matching: Easy Case

In this section, we describe a linear time algorithm which will construct the $(3,3)$-block tree for G given a $(3,3)$-block tree for a 3-connected graph H obtained from G by contracting the edges of an induced matching all of whose vertices have degree between 4 and d. Here we consider a simpler case of the problem, the more difficult case being described in Section 3.5.4. In this case, we suppose that every $(3,3)$-cut of G can be obtained from taking a

subset of 3 vertices of $f^{-1}(Y)$ for some $(3,3)$-cut Y of H. Recall that f is the map from G to H contracting the edges of the induced matching.

Suppose Y is a $(3,3)$-cut in H and we want to determine if a subset S of three vertices of $f^{-1}(Y)$ is a $(3,3)$-cut in G. If the vertices in $Y - S$ connect the components of $G - S$ into fewer than three components then S is not a $(3,3)$-cut in G. Otherwise, S is a $(3,3)$-cut in G.

This concept is formalized using the notion of a component graph.

Definition 3.5.2. Let G be a graph and Z a subset of the vertices of G. Let $C_G(Z)$ be the (simple) graph obtained from G by contracting all the edges with no endpoint in Z. $C_G(Z)$ is called the *component graph* of Z in G.

A vertex in $C_G(Z)$ that is not in Z is called a *component vertex*.

$S \subseteq Z$ is a $(3,3)$-cut in G if and only if S is a $(3,3)$-cut in $C_G(Z)$. However, $C_G(Z)$ is typically much smaller than G.

We outline the main algorithm of this section.

For each cutting node t of the $(3,3)$-block tree of H,

1. Build $C_H(H_t)$.
2. Build $C_G(f^{-1}(H_t))$.
3. Determine which sets of three vertices of G contained in $f^{-1}(H_T)$ are $(3,3)$-cuts of G by examining this graph.
4. Construct the $(3,3)$-block tree for G by combining the information that we have obtained.

Building the component graph for H_t is easy. Since the components of $H - H_t$ correspond to components $T - t$ of the decomposition tree T and edges from H_t to $H - H_t$ have already been assigned to the nodes of the tree. We further note that the vertices in the cut obtained by contracting an edge of the matching have bounded degree so that the total size of this component graph is constant. Furthermore, if we are given the component graph of H_t in H,

we can easily obtain the component graph of $f^{-1}(H_t)$ in G. To do so, we simply uncontract the vertices contained in $f(M) \cap H_t$ in $C_H(H_t)$. We then look at the neighbourhood of edges of the vertices in $f^{-1}(H_t)$ to decide the endpoints of the corresponding edges in $C_G(f^{-1}(H_t))$. This can be done with a list of edges vw with $v \in H_t$ to $w \in H - H_t$ and a pointers from each edge to the component of $H - H_t$ containing v. (When we uncontract v, the component containing w does not change.)

To build the $(3,3)$-block tree of G, we root the $(3,3)$-block tree of H. We then use dynamic programming, examining this tree from the leaves to the root, to build the $(3,3)$-block tree of G. We first describe a quadratic time algorithm to do this and then improve it to a linear time algorithm.

We traverse the $(3,3)$-block tree of H in a post-order traversal. When we examine a cutting node, uncontract everything (of the matching) in this cutting node or below it that is not already uncontracted. For every component consisting of a union of child graph nodes cut off from the parent graph node by a $(3,3)$-cut contained in the 3-cut of this cutting node, make a graph node in the new $(3,3)$-block tree. Merge the remaining graph nodes into the parent graph node.

This algorithm may take quadratic time as we may merge the same set of vertices multiple times. To make this a linear time algorithm, we replace these child graph nodes with an auxiliary reminder vertex encoding what merges we should do. In particular, when deciding if a graph node is cut off by a cutting node from a parent, we need not look at any graph nodes that will be merged into it. When we complete our traversal of the $(3,3)$-block tree, we consider the tree in post-order repeatedly expanding a graph node by replacing the auxiliary vertices corresponding to children of a particular cutting node by the corresponding graph nodes.

We note that the algorithm described in this section can be used (without modification) to build the $\mathcal{F}$-block tree of G where $\mathcal{F}$ is the family of sets X of G such that $f(X)$ is a

$(3,3)$-cut in H. We refer to this tree as the *partial* $(3,3)$*-block tree* and use it in the general case in Section 3.5.4.

3.5.3 Combining $(3,3)$-Block Trees

In this section, we describe how to obtain a solution to G given the solution to the graph nodes of an $\mathcal{F}$-block tree of G where $\mathcal{F}$ is a set of laminar 3-cuts of G, each of which contains an edge.

If we obtained a K_5-model as a solution from any of our subproblems, we can easily obtain a K_5-model for G by using the following lemma. It shows that all graph nodes of the $\mathcal{F}$-block tree are minors of G.

Theorem 3.5.3. *Let G be a 3-connected graph and $\mathcal{F}$ a set of laminar 3-cuts in G, each of which contains an edge. Then the index of graph nodes of the $\mathcal{F}$-block tree of G are all minors of G.*

The proof is similar to Corollary 2.6.22.

Proof. We prove that decomposing on any one cut $X \in \mathcal{F}$ yields decomponents which are minors of G. The theorem then follows by repeatedly decomposing on cuts of $\mathcal{F}$ to obtain the index of graph nodes of the $\mathcal{F}$-block tree.

Suppose we decompose G on $X = \{x_1, x_2, x_3\} \in \mathcal{F}$ with $x_1x_3 \in E(G)$ and obtain decomponents $C_1, \ldots, C_k$. C_1 is a minor of G as we can contract all of U_2 (the component corresponding to C_2) into a single vertex u_2 and the contract u_2x_2 to form a clique on X. By symmetry, each C_i is a minor of G. □

If on the other hand we obtain a $(3,3)$-block tree for each graph node, we apply Algorithm 3.5.4, for which we now provide details.

Algorithm 3.5.4.

Input:

- A 3-connected graph G,
- a set $\mathcal{F}$ of laminar 3-cuts of G each of which contains an edge which decompose G into at least εn pieces,
- an $\mathcal{F}$-block tree $[T, \mathcal{G}]$ of G, and
- a $(3,3)$-block tree of G_t for each of the graph nodes G_t.

Output: A $(3,3)$-block tree of G.

Running time: $O(|V(G)| + |E(G)|)$..

Description and analysis. Since $\mathcal{F}$ is laminar, by Lemma 2.3.11, each cutset $X \in \mathcal{F}$ appears in a cutting node t. By Lemma 2.3.14, for each s adjacent to t in T, there is a unique graph node $(G_s)_t$ of the $(3,3)$-block tree of G_s containing all of X.

We first build an $\mathcal{F}'$-block tree $[T', \mathcal{G}']$ by

- taking the disjoint union of all cutting nodes of T and all $(3,3)$-block trees for graph nodes of T (where all nodes x keep their indexing graph G_x), and
- joining every cutting node G_t to $(G_s)_t$ for each s adjacent to t in T.

(Thus, $\mathcal{F}'$ consists of the union of $\mathcal{F}$ and all $(3,3)$-cuts in graph nodes of T.)

Note that as we constructed it, T' is still bipartite and a tree (with cutting nodes in one bipartition and graph nodes in another). Now any cutting node t' with at least 3 neighbours corresponds to a $(3,3)$-cut of G.

On the other hand, cutting nodes with exactly two neighbours in T' correspond to $(3,2)$-cuts X of $\mathcal{F}$. In particular, since we added a triangle for each cut, their neighbouring graph nodes contain a triangle and is thus not L (recall this is the 8 vertex graph of figure 2–8). So their neighbouring graph nodes s_1 and s_2 are planar and X is a triangular face in G_{s_1} and G_{s_2} (otherwise, X would be separating in some graph node and that graph node would contain vertices from different components of $G - X$, which is impossible by definition). Thus, we may obtain a new planar graph $G_{s_1 s_2}$ by putting all of G_{s_1} inside this triangular

face X and all of G_{s_2} on the outside. We then contract s_1t and ts_2 in T' to obtain a new graph node s_1s_2 with the corresponding graph $G_{s_1s_2}$ that we just built.

We apply the above contraction operation for all cutting nodes of degree 2 in T'. This yields the $(3,3)$-block tree of G. □

3.5.4 Uncontracting a Matching: Hard Case

Overview

In this section we present and prove the correctness of Algorithm 3.5.5 which given a $(3,3)$-block tree of a graph H obtained by contracting a special matching M of G, returns either a $(3,3)$-block tree of G or a K_5-model in G.

Algorithm 3.5.5.

Input:

- A 3-connected graph G,
- an induced matching M in G, all of whose vertices have degree at most d in G, such that $H = G/M$ is 3-connected, and
- a $(3,3)$-block tree of H

Output:

- A K_5-model of G, or
- a $(3,3)$-block tree of G.

Running time: $O(|V(G)|)$.

First, we note that we may assume all graph nodes of the $(3,3)$-block tree of H are planar. Otherwise, since we add a triangle on every $(3,3)$-cut we decompose on and L is triangle-free, the $(3,3)$-block tree of H consists of a single graph node which is L and we use a brute force algorithm.

We need to find three types of $(3,3)$-cuts in G discussed in Section 3.3.3. They are $(3,3)$-cuts X where

1. $f(X)$ is a $(2,3)$-cut of H,

2. $f(X)$ is a $(3,3)$-cut of H, or
3. $f(X)$ is $(3,2)$-cut of H which induces a connected subgraph of H, and $X = N(v)$ for some vertex v of degree 3 in G which is in M.

There is no $(3,3)$-cut X such that $f(X)$ is a 2-cut in H since we chose M so that H is 3-connected. In Section 3.5.2, we already discussed how to find the $(3,3)$-cuts for which $f(X)$ is a $(3,3)$-cut in H. Thus, we can build the $\mathcal{F}$-block tree for these cuts $\mathcal{F}$ (i.e., the partial $(3,3)$-block tree). We then construct a $(3,3)$-block tree for each graph node of the partial $(3,3)$-block tree (or find a K_5-model) and apply Algorithm 3.5.4 to combine them into the $(3,3)$-block tree of G. It remains to construct the $(3,3)$-block tree where only the third kind of cuts appears.

So, the key is to handle the $(3,3)$-cuts of G for which $f(X)$ is a $(3,2)$-cut in H. Before formally presenting the algorithm, we discuss how we handle such cuts.

Our first objective is to turn $(3,2)$-cuts of H into separating triangles in some graph node. After this transformation, we describe how to find all separating triangles. We then update the $(3,3)$-block tree of H using these triangles to obtain a $(3,3)$-block tree of G.

Turning $(3,2)$-cuts into Triangles

We now attempt to add edges to the graph nodes of our decomposition, maintaining the planarity of its nodes, so that every $(3,3)$-cut of G corresponds either to a cutting node of the tree or to a separating triangle in one of the graph nodes. If we fail, then we will find and return a K_5-model.

As discussed in the previous section, cuts of the third type which are of interest to us only arise from contracting an edge uv containing a degree 3 vertex v. Furthermore, such a contraction yield either a triangle or a P_3. Triangles are already separating triangles in H (if they arise from a $(3,3)$-cuts in G) so we only need to worry about turning P_3s into triangles. We refer to these P_3s which arise from contracting an edge containing a degree 3 vertex of G as *contracted P_3s*.

We claim that we can add edges between the endpoints of all contracted P_3's to turn them into triangles, whilst maintaining the planarity of the graphs corresponding to the graph nodes of our tree decomposition.

We now show that any such contracted P_3's (i.e., a P_3 obtained by contracting an edge containing a degree 3 vertex) lie in a unique graph node.

Lemma 3.5.6. *The endpoints of any contracted P_3 lie in the same graph node G_t and no other graph node.*

Proof. To begin, we note that since such contracted P_3's are connected 3-cuts of H, they are not separated by the $(3,3)$-cuts of H. Thus, each of them is contained in the subgraph G_t corresponding to some graph node t. Furthermore, t is unique (since the intersection of the graphs corresponding to any two distinct graph nodes is a clique of size at most 3). □

We can thus add an edge between the endpoints of a contracted P_3 by adding the edge in the graph node containing that P_3. We still need to show that adding such an edge preserves the planarity of the graph node.

Lemma 3.5.7. *We can add an edge between the endpoints of a contracted P_3 in a graph node G_t while preserving the planarity of G_t.*

Proof. Each P_3 is a 3-cut of G_t as well since they do not separate the $(3,3)$-cuts so they cannot be removed as we perform a decomposition. This implies that the vertices of the P_3 do not all lie together on some face of the embedding of the graph node G_t (the embedding is unique due to 3-connectivity) containing it. Otherwise, by adding a vertex in this face adjacent to all of the vertices of the P_3 and contracting two components of the 3-cut to a single vertex, we would obtain a planar drawing of a graph with a $K_{3,3}$-minor. It also implies that there is a face containing the endpoints of the P_3. Indeed an (induced) P_3 is a cutset of the 3-connected planar graph G_t precisely if there is a face containing its endpoints but not its midpoint.

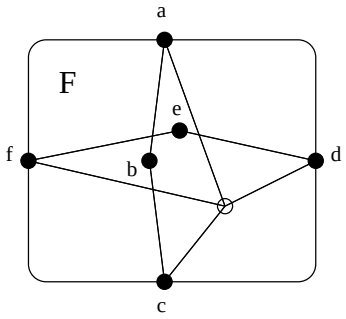


Figure 3–1: A K_5-minor obtained from two P_3's with both endpoints on F and distinct midpoints not on F. The added vertex is white.

We see then that for any specific P_3 corresponding to a $(3,3)$-cut of G, we can add an edge between its endpoints to the graph node which contains it whilst maintaining the planarity of this graph node. □

In fact, we can add all such edges while preserving planarity.

Lemma 3.5.8. *We can add all edges between the endpoints of contracted P_3s in any graph node G_t while maintaining the planarity of G_t.*

Proof. It is enough to show that there does not exist a face F of the embedding of G_t and two P_3's abc and def with no endpoints in common such that a, d, c, f appear in the given cyclic order around F. If the midpoints b and e are distinct, the boundary of F, along with a vertex added inside F adjacent to all four of a, c, d, and f, which yields a planar drawing of a graph with a K_5-minor (see Fig. 3–1).

So, we need only consider two P_3's with the same midpoint. In this case, the midpoint is the image of a matching edge which joins two vertices of degree 3 each of whose neighbourhoods is a $(3,3)$-cut. In this case, it is easy to verify that the corresponding $(3,3)$-cuts of G cross which is impossible. Thus we can indeed add all the edges to make these P_3's into triangles. □

We now show how to add all such edges in linear time. We need only generate a list of edges to add, since for any graph node t and corresponding subgraph G_t with the extra edges added, finding a planar embedding can be done in linear time and since G_t is 3-connected it remains so when we add these edges and this embedding will be unique.

Theorem 3.5.9. *We can add all edges between endpoints of contracted P_3's which are $(3,2)$-cuts in linear time.*

We prove this theorem assuming we have the following subroutine.

Lemma 3.5.10. *There is a linear time algorithm which given an embedding of a 3-connected graph G_t and a list of $O(|V(G)|)$ P_3's in that graph outputs the sublist of the input P_3's with endpoints in a face not containing its midpoint.*

Proof. (of Theorem 3.5.9)

We only need to determine which vertices v of degree 3 in G contained in the matching M are contracted to the midpoint of a P_3 that is a $(3,2)$-cut in H. As remarked in the previous proof, such a P_3 is a $(3,2)$-cut precisely if there is a face containing its endpoints but not its midpoint in the planar embedding of the unique graph node containing this P_3.

Thus, our problem reduces to determining for each G_t which of at most $|V(G_t)|$ P_3's of G_t satisfy this property. We can directly apply Lemma 3.5.10 to obtain this list and thus, the list of edges we need to add.

□

We now prove Lemma 3.5.10 which itself requires Lemma 3.5.11.

Proof. We now show how to determine, in linear time, for which of a list of $O(|V(G_t)|)$ P_3's in a planar 3-connected graph G_t, there is a face containing the endpoints of the P_3 but not the midpoint. In a 3-connected planar graph, every two nonadjacent vertices lie together in at most one face. We will separately check for which P_3's some face contains their endpoints and for which P_3's there is a face containing all three vertices of the P_3.

Because G_t is 3-connected, the latter occurs precisely if the P_3 is part of the boundary of this face in the unique embedding of G_t. But we can check this by generating an embedding and enumerating all the P_3's on its face boundaries (note we sort these lists lexicographically before comparing them, to do so we use bucket sort, which is linear in the size of the graph). So the second problem can be solved in linear time; it remains to solve the first.

We are actually asking which of the endpoint pairs have a common neighbour in the face-vertex incidence graph. We can generate the face-vertex incidence graph J in linear time, and it has linear size. Furthermore, it is planar and so has degeneracy at most 5 (that is, we can repeatedly remove vertices of degree at most 5 until the graph is empty). We now simply apply Lemma 3.5.11. □

We now describe the final subroutine needed.

Lemma 3.5.11. *For any graph J of degeneracy bounded by some constant C, and list of $O(|V(J)|)$ pairs of vertices, we can determine all common neighbours of all pairs in the list in $O(|V(J)|)$ time.*

Proof. We consider an ordering $<$ of $V(J)$ for which each vertex has at most C neighbours appearing before it in the order. For every w, there are at most C^2 pairs of vertices where both vertices appear before w in the order and have w as a common neighbour. So we can determine, for each pair on the list, all their common neighbours appearing after both of them in linear time. It remains to determine for every pair u, v with $u < v$, the set of common neighbours which are less than v. There are at most C neighbours of v appearing before v, which makes this task easy. Formally, as we strip vertices of degree at most C out of J to generate our order, we can orient the edges of J so that each edge xy with $x < y$ is directed from x to y. Then, we can traverse the edge list of J once to obtain for each v the list $N^-(v)$ of vertices $w < v$ such that wv is an edge of J. Now we can traverse our set of vertex pairs, replacing a pair (u, v) with $u < v$ with the at most C pairs uw for each w

in $N^-(v)$. Next we can traverse the edge list of J again to see which of these new pairs are edges of J. Finally u and v have w as a common neighbour which appears before v if and only if we found that uw is an edge and that $w \in N^-(v)$.

Again, before comparing the lists we sort them using bucketsort which takes linear time. This completes the description of the linear time algorithm. □

Updating the Decomposition Tree

Having added an edge between the endpoints of every contracted P_3, we are ready to find all newly formed triangles which are separating. In fact, we start by finding all separating triangles in every graph node.

Lemma 3.5.12. *We can find all separating triangles in any graph node G_t in linear time.*

Proof. We exploit the fact that each G_t has a unique planar embedding. Thus, each non-separating triangle is a face. So, since we have a planar embedding of the graph, to generate all the separating triangles we need only generate all the triangles. But, for an edge uv, we have that uvw is a triangle precisely if w is a common neighbour of u and v. We use Lemma 3.5.11 with $C = 5$ to generate the set of such w for every edge uv, thereby obtaining the set of triangles of G. □

Our next step will be to modify the decomposition tree of H to take all $(3,3)$-cuts found in the previous lemma into account.

Lemma 3.5.13. *We can add all $(3,3)$-cuts found in Lemma 3.5.12 to the $(3,3)$-block tree of H (as cutting nodes) in linear time.*

Proof. Basically when we uncontract the midpoint ux of the P_3 and do the cut on $N(u)$, we turn the P_3 into a triangle (which we have already done), and then replace the vertex of H corresponding to the matching edge by x. This latter step does not change H at all because after turning the P_3 into a triangle this vertex has the same neighbourhood in G and H as the old contracted vertex did. What we do need to do is to split apart the graph into

pieces using all the cuts corresponding to these separating triangles in our extended planar embedding which have now become $(3,3)$-cuts. We note that some of these correspond to P_3s we have turned into triangles and others correspond to triangles in the embedding of G_t.

Thus, we take the list of all separating triangles from Lemma 3.5.12 and apply Lemma 3.5.14 to add all these separating triangles to our decomposition. □

Lemma 3.5.14. *Given a list of separating triangles, we can update the* $(3,3)$*-block tree in linear time.*

Proof. We check which of these separating triangles corresponds to a $(3,3)$-cut X of G which is $N(v)$ for some vertex v of degree 3 which is in the matching. This simply involves looking at the preimage of each separating triangle (under the contraction of the matching) and can easily be done in linear time. We obtain a set of separating triangles on which we will split G_t, adding a cutting node and a graph node with vertex set $\{v\} \cup N(v)$ incident to it for each such triangle.

We now have to partition the graph further, because each separating triangle which becomes a 3-cut will cause us to split our planar embedding into two smaller embeddings, with our separating triangle as a face in each of these embeddings. A linear time algorithm to do so has been developed by Havet, Quercini, and Reed [38]. It uses a breadth first search of the face vertex graph of a planar embedding, exploiting the fact that the vertices of separating triangles all lie on the same level of this tree or on levels i and $i+2$ for some i. They treat each such pair of consecutive levels separately to solve the problem (this can be done in two ways, either using the fact that the graph under consideration has bounded tree width or via a more combinatorial approach). □

Building the separating triangle block tree

In this section, we describe a linear time algorithm which takes a 3-connected planar graph G and a list $\mathcal{T}$ of separating triangles (in the unique embedding of G) and builds an $\mathcal{T}$-block tree of G.

We first describe an algorithm which runs in polynomial but not linear time which we later speed up.

We can repeatedly decompose on a triangle in $\mathcal{T}$ to recursively build the $\mathcal{T}$-block tree. Instead, at each step, we could choose the triangle of $\mathcal{T}$ to decompose on more carefully. We choose a triangle that when decomposed on will yield a decomponent which contains no more separating triangles of $\mathcal{T}$ (i.e., that decomponent is a leaf graph node of the $\mathcal{T}$-block tree).

Note that since G is 3-connected, there are always two decomponents of $G - T$ for any $T \in \mathcal{T}$ so this guarantees we are always working with a single graph (as opposed to a list of decomponents in general).

One way to find is to

1. remove all dual edges in triangles of $\mathcal{T}$ in the dual G^* of G,
2. find all resulting dual components $\mathcal{U}^* = U_1^*, \ldots, U_k^*$ of $G^* - \cup_{T \in \mathcal{T}} E(T^*)$,
3. construct the (bipartite) triangle-component incidence graph B (with vertex set $\mathcal{U}^* \cup \mathcal{T}$ and edges $\{(U_i^*, T) | \text{the dual of an edge of } T \text{ has an endpoint in } U_i^*\}$), and
4. find a component U_i^* incident to only one triangle T in B.

Indeed, if we decompose on T, U_i (the dual of U_i^*) is a decomponent containing no other separating triangle of $\mathcal{T}$ (by construction of B).

Such a "leaf" component U_i^* always exists and we can always pick T to be a smallest triangle (in terms of area in a (non-combinatorial) embedding of G) and the "inside" of T (which contains at least one vertex) is a component incident to only T^*.

Having described a polynomial time algorithm, we now present the details of our linear time algorithm. We speed up our algorithm keeping track of the dual components $U_1^*, \ldots, U_k^*$ and the triangle-component incidence graph throughout without rebuilding them at each step. To do so, we only keep track of a list of vertices and dual vertices for each dual component U_i^* (rather than the whole dual graph).

We also find "leaf" components faster by keeping a queue of "leaf" components and simply update this queue for free while performing other updates.

Algorithm 3.5.15.

Input:

- A planar 3-connected graph G, and
- a list $\mathcal{T}$ of separating triangles in G.

Output: A $\mathcal{T}$-block tree of G.

Description and analysis.

1. We initialize a list of graph nodes (that is initially empty). In fact, this list will only contain vertices of graph nodes.
2. Build H^* by removing all dual edges in triangles of $\mathcal{T}$ in the dual G^* of G (i.e., $H^* = G^* - \cup_{T \in \mathcal{T}} E(T^*)$).
3. Find all resulting dual components $\mathcal{U}^* = U_1^*, \ldots, U_k^*$ of H^*.
4. Construct the (bipartite) triangle-component incidence graph B (with vertex set $\{V(U_i^*) | U_i^* \in \mathcal{U}_i^*\} \cup \mathcal{T}$ and edges $\{(V(U_i^*), T) |$ the dual of an edge of T has an endpoint in $U_i^*\}$).
5. Find all components U_i^* incident to only one triangle T in B and put them in a queue Q. As we remarked, there is at least one such component and Q is now non-empty.
6. While Q is non-empty
   (a) Remove the top component U_i^* from Q. U_i^* is adjacent to only one triangle T.
   (b) Add $V(U_i^*)$ to our list of graph node vertices to output.
   (c) Remove U_i^* from B.

(d) Merge all components incident to T in B (i.e., delete these vertices for these components, replace them by a single vertex adjacent to the union of their neighbourhood and replace the index for these components by the union of the index of all removed components).

(e) Check if the newly merged component has degree 1. If so, add it to Q.

Since every triangle of B has degree 6, we can delete the vertex corresponding to a triangle in constant time. We can store the vertices of each component as a doubly linked list so we may merge them in constant time. In each iteration of step 6, we merge at most 3 such lists in constant time.

We can easily check if a newly merged vertex has degree 1 by looking at the first four edges (or less if there are fewer) incident to it. Thus step 6e), can also be completed in constant time.

However, we cannot perform step 6d) (i.e., update the bipartite graph with the adjacencies of the newly merged vertex) in constant time. The at most 3 vertices (representing components) that we merge may not have bounded degree so updating B may be costly. If we think of vertices representing components of B as elements of a set, we are simply performing a union operation at each step (and a find operation for other substeps of step 6). In fact, B is planar and we use the results of [33] which allows us to perform all union-find operations of step 6 in linear time.

B is planar since it is the minor of a graph consisting of one vertex for each component of $\mathcal{U}^*$ and three vertices for each triangle $T \in \mathcal{T}$, one vertex in the middle of each edge of T, all connected to each other (using the same "lines" as the original embedding of T (we kept three vertices for T so we could use the original embedding)). We can then contract the 3 vertices for each triangle to obtain B.

Therefore, our entire algorithm runs in linear time as required.

□

Algorithms

We now summarize the algorithmic steps described in the previous subsections.

Algorithm 3.5.16 (Main algorithm)**.**

Input:

- a 3-connected graph G,
- a matching M in G such that $H = G/M$ is 3-connected,
- a $(3,3)$-block tree of H, and
- a partial $(3,3)$-block tree of G,

Output:

- a K_5-model of G, or
- a $(3,3)$-block tree of G

Algorithm:

- Add all edges between contracted P_3's inside graph nodes of the $(3,3)$-block tree of H which are $(3,2)$-cuts (Algorithm 3.5.17).
- Find all separating triangles in all graph nodes of the input $(3,3)$-block tree of H (using Algorithm 3.5.20 on each graph node).
- Obtain a $(3,3)$-block tree of G (Algorithm 3.5.18).

Algorithm 3.5.17.

Input:

- a 3-connected graph G,
- a matching M in G such that $H = G/M$ is 3-connected,
- a $(3,3)$-block tree of H, and
- a partial $(3,3)$-block tree of G,

Output: The input $(3,3)$-block tree of H with an edge between all contracted P_3's which are $(3,2)$-cuts.

Algorithm:

- For each graph node F,
  - Obtain a planar embedding of F.
  - Build a list L_1 of all contracted P_3's (for each vertex v in M of degree 3 in G, check if $N(v)$ induces a P_3).
  - Build a list L_2 of P_3's in F with endpoints in a face not containing its midpoint (using Algorithm 3.5.19).
  - Build a list L_3 of elements of L_1 that are also in L_2 (i.e., $L_1 - L_2$).
  - For each P_3 in L_3, add an edge between its endpoints.
  - Obtain a planar embedding which includes all added edges.

Algorithm 3.5.18.

Input:

- a 3-connected graph G,
- a matching M in G such that $H = G/M$ is 3-connected,
- a $(3,3)$-block tree of H,
- a partial $(3,3)$-block tree of G, and
- a list L_1 of all separating triangles in H.

Output: A $(3,3)$-block tree of G.

Algorithm:

- For each graph node G_t
  - For each P_3 y, ux, z that is now a separating triangle in G_t where $N(u) = \{x, y, z\}$ in G
    - Add a new cutting node containing $\{x, y, z\}$.
    - Add a new graph node containing u, x, y, z adjacent to the cutting node corresponding to the P_3 y, ux, z.
  - Replace G_t by a set of graph nodes using algorithm 3.5.15
- Return the modified $(3,3)$-block tree as the $(3,3)$-block tree of G.

Algorithm 3.5.19.

Input: A planar 3-connected graph G and a list L_0 of P_3's in G.

Output: The list L_2 of all P_3's in L_0 with endpoints in a face not containing its midpoint.

Algorithm:

- For each face in the planar embedding of G, enumerate all consecutive triplets of vertices.

  This yields a list L_1 of P_3's with all vertices contained in some face.
- Build the face-vertex incidence graph F and obtain a planar embedding of it.
- Repeatedly remove a minimum degree vertex in F until no vertices are left.

  Define $<$ on $V(F)$ where $u < v$ if u is removed later than v.
- For each edge $e = (u, v) \in E(F)$ with $u < v$ orient e from u to v
- We now build a list L of all pairs of vertices in F with a common neighbour using Algorithm 3.5.21.
- Output all elements of $L_0 - L_1$ with endpoints (as a pair) in L.

Algorithm 3.5.20.

Input: A 3-connected planar graph G_t.

Output: A list of all separating triangles in G_t.

Algorithm:

- Use Algorithm 3.5.21 to obtain a list L_2 of common neighbours of G_t.
- Build a list L_1 of all triangles in G_t using the list of common neighbours L_2 (by checking if $uv \in E$ for each $((u, v), w)$ in L_2).
- Remove from L_1 all triangles which are faces in G_t.
- Return L_1.

Algorithm 3.5.21.

Input: A planar graph F.

Output: A list L_2 of triplets $((u, v), w)$ of all pairs of vertices u, v with a common neighbour

w.

Algorithm:

- Let L_2 be initially empty.
- For each pair u, v, for each $w \in N^-(u) \cap N^-(v)$, add $((u, v), w)$ to L_2.
- For each vertex w, add $((u, v), w)$ to L_2 for every pair of vertices (u, v) in $N^-(w)$.
- Return L_2.

CHAPTER 4
2-Disjoint Rooted Paths

In this section, we present a linear time algorithm for the two disjoint rooted paths problem (2-DRP) where we are given an input graph G and four vertices $s_1, t_1, s_2, t_2 \in V(G)$ and we wish to determine whether there exist two vertex disjoint paths in G, one from s_1 to t_1 and one from s_2 to t_2.

Our algorithm is similar to the algorithm described in the previous section as it also makes one of several reductions before recursively applying itself on set of much smaller problems. Our algorithm also produces a tree decomposition as a certificate when the desired paths do not exist. However, we solve an equivalent problem in an auxiliary graph and build a tree decomposition of this auxiliary graph if the desired paths do not exist in the original problem.

We first establish this well known equivalence in the next section before survey past results in Section 4.2 and delving into the details in remaining sections.

4.1 2-DRP and Attached K_5-Minors

In this section, we establish the equivalence between 2-DRP and the problem of finding "attached" minors. Our main algorithm will solve this equivalent problem.

Problem 4.1.1. *[2-DRP]*

Input: *A graph G and four vertices s_1, t_1, s_2, t_2.*

Output: *"Yes" if there exist two vertex disjoint paths, one from s_1 to t_1 and the other from s_2 to t_2, and "no" otherwise.*

Definition 4.1.2. We say an instance of 2-DRP is *feasible* or *realizable* if the desired $s_1 - t_1$ and $s_2 - t_2$ paths exist in G.

Necessary condition

First, we derive a necessary condition for the paths to exist. Suppose G is planar and s_1, t_1, s_2, t_2 appear in this order on a face of G (see Figure 4–1). Then then vertex disjoint $s_1 - t_1$ and $s_2 - t_2$ paths do not exist as any two curves, one from s_1 to t_1 and the other from $s_2 - t_2$, must intersect so any two paths must intersect.

More formally, we can add a vertex x^* to G adjacent to s_1, t_1, s_2, t_2 and edges $s_1t_1, t_1s_2, s_2t_2, t_2s_1$ (see Figure 4–1) and obtain a planar graph G^* (since G is planar and s_1, t_1, s_2, t_2 appear in this order on a face of G). Now if the desired paths exist in G, we obtain a K_5-minor in G^*, a contradiction to Kuratowski's theorem.

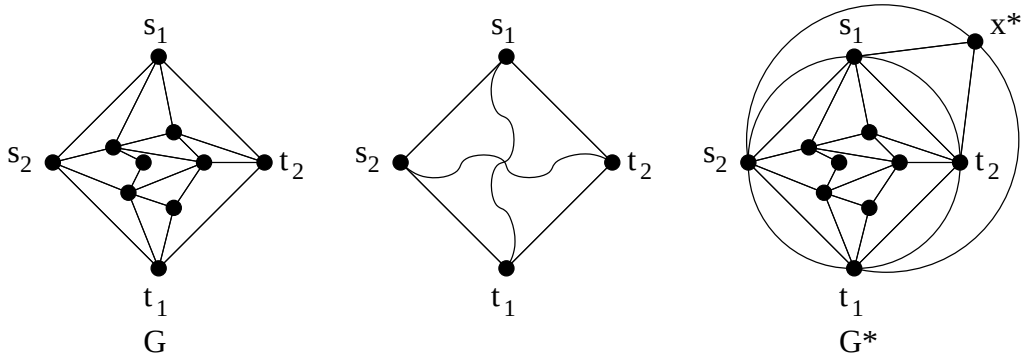


Figure 4–1: A planar graph G with vertices s_1, t_1, s_2, t_2 on a face, two curves inside this face intersecting and the auxiliary graph G^* obtained by adding edges $s_1t_1, t_1s_2, s_2t_2, t_2s_1$ and a vertex x^* adjacent to s_1, t_1, s_2, t_2.

This auxiliary graph G^* is key to our equivalent problem.

Definition 4.1.3. Let (G, s_1, t_1, s_2, t_2) be an instance of 2-DRP. We call G^*, the graph obtained from G by adding a vertex x^* to G adjacent to s_1, t_1, s_2, t_2 and edges $s_1t_1, t_1s_2, s_2t_2, t_2s_1$ the *auxiliary graph* obtained from (G, s_1, t_1, s_2, t_2).

In fact, we have just shown that if G^* (which we can build without first checking the planarity of G) has no K_5-minor, the desired paths do not exist. The converse is almost true.

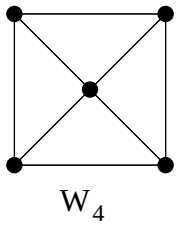


Figure 4–2: The 4-wheel, W_4

Connectivity reductions and equivalence

For this converse, we must first eliminate small cuts as we now see K_5-models separated by small cuts from s_1, t_1, s_2, t_2 are of no use in finding the desired paths.

Suppose now that G contains a minimal cut X of size at most 3 and a component U of $G - X$ does not contain any of $\{s_1, t_1, s_2, t_2\}$. Then we claim the desired paths exist in G if and only if they exist in the graph G' obtained by deleting U and adding a clique onto X (this is not quite a decomposition on X as we keep all components with a vertex of $\{s_1, t_1, s_2, t_2\}$ in the same decomponent).

Indeed, if the paths exist in G then any path that uses a vertex of U can be shortened by using an edge we added between vertices of X so the paths exist in G'. On the other hand, if the paths exist in G', since $|X| \leq 3$, at most one (added) edge between vertices of X is used. We can replace this edge by a path between them using only vertices of U (such a path always exists in $G[U \cup X]$ as X is a minimal cut).

Thus, we've shown the desired paths exist in G if and only if they exist in G'. The following three remarks on the above discussion gives us the exact characterization.

The first is that if no cut of size at most 3 with a component containing none of $\{s_1, t_1, s_2, t_2\}$ exists and G^* contains a K_5-minor then the desired paths exist [42] (see [53] for this version stated in terms of minors).

The second remark states that the component U discussed above is disjoint from $\{s_1, t_1, s_2, t_2\}$ so X is also a cut in the auxiliary graph G^*. In fact, we can restate our operation of removing

U and adding a clique on X as building G^*, decomposing on X in G^* and removing the extra vertices and edges in G^*.

Thus, if $\{x^*, s_1, t_1, s_2, t_2\}$ cannot be separated by a cut of size at most 3 in G^*, we can decompose on all cuts of size at most 3 in G^* (in increasing order of size) and focus on the final decomponent containing $\{x^*, s_1, t_1, s_2, t_2\}$. If this final decomponent is K_5-minor free, then the desired paths do not exist in our original instance of 2-DRP.

The third remark tells us that if a K_5-minor is present in this final decomponent then the desired paths do exist [53].

Thus, a K_5-minor exists in this final decomponent in G^* if and only if the desired paths exist in G.

We now formalize the reduction discussed in this section and our three remarks.

Lemma 4.1.4. *[53] (see also [42]) Let (G, s_1, t_1, s_2, t_2) be an instance of 2-DRP and G^* be the auxiliary graph obtained from (G, s_1, t_1, s_2, t_2). If G^* is 4-connected and contains a K_5-minor then there exist vertex disjoint $s_1 - t_1$ and $s_2 - t_2$ paths*

Remark 4.1.5. *Let (G, s_1, t_1, s_2, t_2) be an instance of 2-DRP and G^* be the auxiliary graph obtained from (G, s_1, t_1, s_2, t_2).*

Let H^* be the decomponent containing $A = \{x^*, s_1, t_1, s_2, t_2\}$ in the graph obtained from G^* by decomposing on cuts of size at most 3 that do not separate $\{x^*, s_1, t_1, s_2, t_2\}$ in G^* (in increasing order of size) until no such cut exist.

H^* contains a K_5-minor if and only if (G, s_1, t_1, s_2, t_2) is feasible.

We can restate the definition of a K_5-minor in the final decomponent as follows. Using this definition, we give an equivalent problem to 2-DRP.

Definition 4.1.6. Let A be a set of vertices in G and K a clique model of G. A *detacher* (of K from A) is a cut X such that some component U of $G - X$ completely contains a vertex image of K but none of A.

We say that a K_5-model is *attached to* A if there does not exist a detacher of K of size at most 3.

In terms of attached minors, we've shown the following.

Theorem 4.1.7. *[53] (see [71, 62, 64] for the original statement not involving K_5-minors) Let (G, s_1, t_1, s_2, t_2) be an instance of 2-DRP and let G^* be the auxiliary graph obtained from (G, s_1, t_1, s_2, t_2).*

Then (G, s_1, t_1, s_2, t_2) is feasible if and only if there is a K_5-model attached to G^*.

Problem 4.1.8 (Attached K_5-model (AK5))**.**

Input: *A graph G^* and a subgraph A of G^* that is a 4-wheel.*

Output: *"Yes" if G^* contains a K_5-model attached to A. "No" otherwise.*

Corollary 4.1.9. *If we can solve attached K_5-model in linear time then we can solve 2-DRP in linear time.*

Note that although we give a linear time algorithm for solving 2-DRP in this chapter, our algorithm does not return the desired paths if they exist.

The most difficult part for AK5 is again the case where G^* is 3-connected. The remainder of this chapter focuses on solving the following restricted version of AK5.

Problem 4.1.10 (Restricted attached K_5-model (RAK5M))**.**

Input: *A 3-connected graph G^* and an induced 4-wheel A in G^* with no 3-cut separating A.*

Output: *"Yes" if G^* contains a K_5-model attached to A. "No" otherwise.*

We close this section by proving that a linear time algorithm for 2-DRP given a linear time algorithm for RAK5M.

Corollary 4.1.11. *If we can solve RAK5M in linear time then we can solve 2-DRP in linear time.*

We give an algorithmic proof, using the linear time algorithm for attached K_5-minor as a subroutine.

Algorithm 4.1.12.

Input: A graph G and four vertices s_1, t_1, s_2, t_2.

Output: "Yes" if there exist two vertex disjoint paths, one from s_1 to t_1 and the other from s_2 to t_2, and "no" otherwise.

Description.

1. If $s_1t_1 \in E(G)$, determine if s_2 and t_2 are in the same connected component of $G - s_1 - t_1$ and if so, return "Yes". Return "No" otherwise.
2. If $s_2t_2 \in E(G)$, determine if s_1 and t_1 are in the same connected component of $G-s_2-t_2$ and if so, return "Yes". Return "No" otherwise.
3. Otherwise ($s_1t_1 \notin E(G)$ and $s_2t_2 \notin E(G)$), proceed as follows.
   (a) Set $A = \{s_1, t_1, s_2, t_2\}$.
   (b) Build the auxiliary graph G^*.
   (c) Build the component of G_0^* of G^* containing A.
   (d) Build the block tree of G_0^* and find the graph node G_1^* containing A.
   (e) Build the 2-block tree of G_1^* and find the graph node G_2^* containing A.
   (f) If there is a 3-cut separating A in G_2^*, return "No".
   (g) Determine if G_2^* has a K_5-model attached to A using the algorithm for RAK5M.
   (h) If so, return "Yes". Return "No" otherwise.

Analysis. Our earlier discussion proves the correctness of this algorithm.

Indeed, (G, s_1, t_1, s_2, t_2) is feasible if and only if $(G_1, s_1, t_1, s_2, t_2)$ is feasible where G_1 is the graph obtained from G_1^* by removing x^* and any edges we added to G. $(G_1, s_1, t_1, s_2, t_2)$ is feasible if and only if $(G_2, s_1, t_1, s_2, t_2)$ is feasible where G_2 is the graph obtained from G_2^* by removing x^* and any edges we added to G.

By Theorem 4.1.7, $(G_2, s_1, t_1, s_2, t_2)$ is feasible if and only if G_2^* has a K_5-model attached to A.

If G_2^* has a 3-cut separating A, since G_2^* is 3-connected, this cut is either $\{s_1, t_1, x^*\}$ or $\{s_2, t_2, x^*\}$ (since these are the only cuts of size at most 3 in A). In the first case, any

$s_2 - t_2$ path in G_2 uses one of $\{s_1, t_1\}$ so the desired paths do not exist in G_2. Similarly, in the second case, any $s_1 - t_1$ path in G_2 uses one of $\{s_2, t_2\}$ so the desired paths do not exist in G_2. □

4.2 Previous work

Jung [42] first showed the equivalence between 2-DRP and the planarity in G^* when G is 4-connected. The extension of Jung's theorem to all graphs was shown independently by Thomassen, Seymour, and Shiloach [71, 62, 64]. They prove the following theorem.

Theorem 4.2.1. *[62, 71, 64] Suppose a graph G with four vertices labelled s_1, s_2, t_1, t_2 does not contain vertex disjoint s_1-t_1 and s_2-t_2 paths. Then either*

- *G can be embedded in the plane such that s_1, s_2, t_1, t_2 appear in this order in some face, or*
- *G has a cutset of size at most 3.*

The corollary of this theorem in terms of K_5-models (Theorem 4.1.7) was first stated explicitly by Reed [53].

Theorem 4.2.2. *[53] If there is a detacher to a K_5-model then there is a detacher of size at most 3.*

Corollary 4.2.3. *Let G be a graph with four vertices labelled s_1, s_2, t_1, t_2. Vertex disjoint s_1-t_1 and s_2-t_2 paths exist in G if and only if there is a K_5-model in G^* with no detachers with respect to $A = \{s_1, t_1, s_2, t_2, x^*\}$ of size at most 3.*

As we stated in the previous section, the main difficult part arises when G is 3-connected and we are looking for 3-cuts. Using their theorem, Thomassen, Seymour, and Shiloach independently gave the first polynomial time algorithm for 2-DRP. Tholey [69] improved this to a $O(m\alpha(m, n))$ algorithm by using a more sophisticated approach for testing for 3-cuts. Later, Tholey [70] further improved his algorithm to run in $O(n\alpha(m, n))$.

4.3 Closest Detachers and Laminarity

As for K_5-minor containment of the previous section, our algorithm provides a certificate in the form of a tree decomposition when an attached K_5-model does not exist.

We'd like to build the $\mathcal{F}$-block tree where $\mathcal{F}$ is all detachers of size 3 (to some K_5-model from the special set $A = \{s_1, t_1, s_2, t_2, x^*$ of vertices). However, there could be exponentially many models and therefore detachers. Furthermore, they may not be laminar. In this section, we describe the laminar set $\mathcal{D}$ of detachers we restrict ourselves to so the $\mathcal{D}$-block tree can be built in linear time.

We note that the graph node G_A containing A in this hypothetical $\mathcal{D}$-block tree needs not be 4-connected (as suggested in Section 4.1). It only needs to contain no detachers.

To do so, we need the notion of a closest detacher.

Definition 4.3.1. A *closest detacher* X of clique model K from a set of vertices A is a detacher of K from A of minimum size which (subject to this) minimizes the size of the union of components of $G - X$ containing a vertex image of K.

The closest detacher is the detacher which is effectively closest to K (and furthest from A). At first, it may seem that the opposite (the "furthest detacher" closest to A, furthest from K) would be more useful as decomposing on it reduces the size of the decomponent containing A the most. However, it turns out the closest detacher has many properties our algorithm will need. We now examine these properties.

We now show the closest detacher to a fixed K_5-model is unique.

Lemma 4.3.2. *[53] Let A be an induced 4-wheel in a graph G.*

Let X be a closest detacher of a K_5-model K and U_X the union of all components of $G - X$ intersecting K. Then for any other minimum size detacher Y of K, the union U_Y of components of $G - Y$ intersecting K contains all of U_X.

For completeness, we repeat (parts of) the proof of Lemma 7.10 in [53] here.

Proof. Let U^* be the union of components of $G - X - Y$ intersecting K. Let Y_1 be the vertices in $X \cup Y$ adjacent to a vertex of U^*. Let Y_2 be the vertices in $X \cup Y$ contained in A or adjacent to a vertex of A.

Note that both Y_1 and Y_2 are detachers of K and hence have size at least $|X|$ (as X is a minimum size detacher). Any vertex in both Y_1, Y_2 must also be in both X and Y (otherwise, there is a path in $G - X$ or $G - Y$ from K to A). That is, $|Y_1 \cap Y_2| \leq |X \cap Y|$.

By construction of Y_1, Y_2, $Y_1 \cup Y_2 \subseteq X \cup Y$ and thus $|Y_1 \cup Y_2| \leq |X \cup Y|$. So

$$|Y_1| + |Y_2| = |Y_1 \cup Y_2| + |Y_1 \cap Y_2| \leq |X \cup Y| + |X \cap Y| = |X| + |Y| = 2|X|$$

So, both Y_1 and Y_2 have size $|X|$. Since U^* is contained in U, $U^* = U$ (or Y_1 contradicts our choice of X). However, U^* is also contained in U_Y, hence U_X is contained in U_Y as required. □

Corollary 4.3.3. *The closest detacher is unique.*

Lemma 4.3.4. *If X is a closest detacher for a K_5-model K (from a set A of vertices) and Y is a cut separating X from K with $|Y| \leq |X|$ then $Y = X$.*

Proof. Suppose not and let G, K, A, X, Y be a counter-example. So Y separates X from K but $Y \neq X$. Then we claim Y is a detacher of K from A. Indeed, suppose there is a path from a component of $G - Y$ completely containing a vertex image of K to A in $G - Y$. Then this path does not contain any of X since Y separates X from K (in particular, X does not intersect this vertex image of K) and thus it is a path of $G - X$ from a component completely containing a vertex image of K to A, a contradiction to X being a detacher.

Therefore, Y is a closest detacher. By Corollary 4.3.3, $Y = X$, a contradiction. □

4.4 Pruned K_5-tree

We are now ready to define the set $\mathcal{D}$ of detachers for the $\mathcal{D}$-block tree our algorithm builds when no attached K_5-minor exists.

We simply pick $\mathcal{D}$ to be the set of all *top detachers*, closest detachers that shrink the decomponent containing A the most.

Definition 4.4.1. A closest detacher X is a *top detacher* if there does not exist another closest detacher Y (with respect to a different K_5-model) such that X does not intersect the decomponent of $G - Y$ containing A.

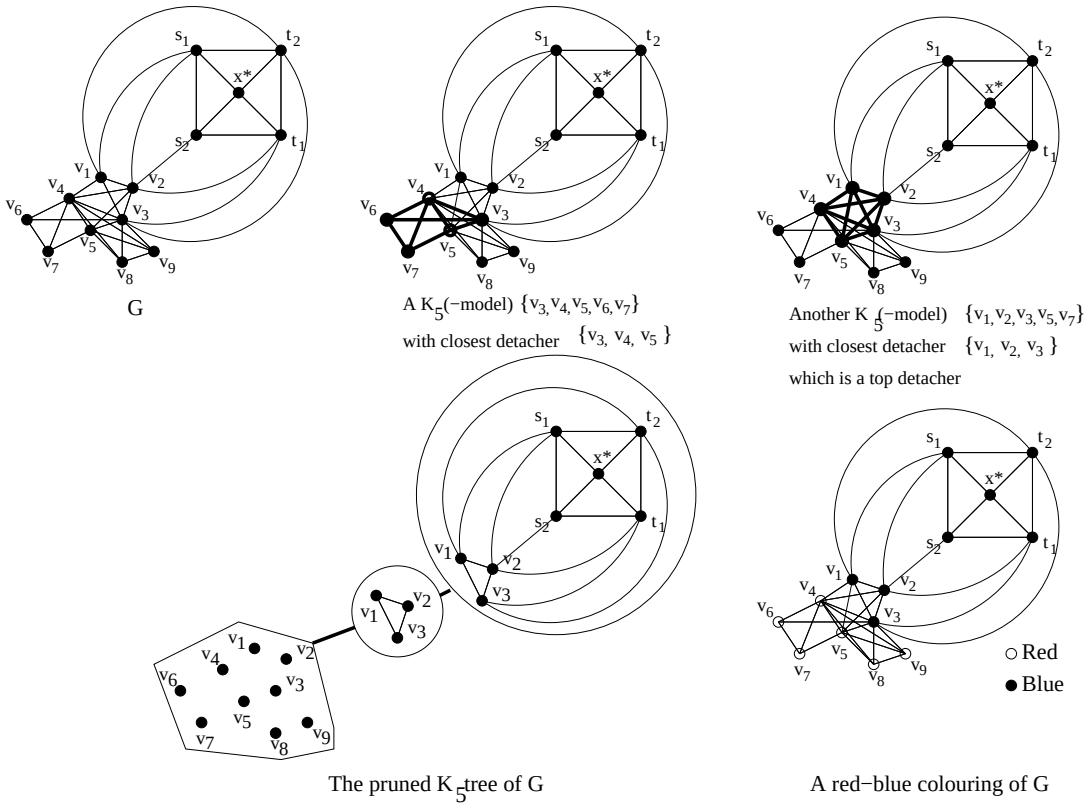


Figure 4–3: (a) A graph G with a set $A = \{s_1, t_1, s_2, t_2, x^*\}$ of vertices (b) A K_5(-model) in G with closest detacher $\{v_3, v_4, v_5\}$ (c) Another K_5(-model) in G, this one with closest detacher $\{v_1, v_2, v_3\}$ which is also a top detacher (d) A pruned K_5-tree of G (d) a red-blue colouring of G

Note that if X and Y are top detachers then A and (all of) X are in the same decomponent of $G - Y$ (otherwise, we contradict Lemma 4.3.2), leading to the following remark.

Remark 4.4.2. *The $\mathcal{D}$-block tree $[T, \mathcal{G}]$ where $\mathcal{D}$ is the set of all top detachers is a star.*

We call the center of this star (containing A) the *root* of T.

The $\mathcal{D}$-block tree of G where the root's index is K_5-model free is sufficient to certify that G contains no K_5-model attached to A. We call this tree the *pruned K_5-tree* of G. We define this tree with a slightly weaker restriction on A.

Definition 4.4.3. The $\mathcal{D}$-block tree $[T, \mathcal{G}]$ where $\mathcal{D}$ is the set of all top detachers is a *pruned K_5-tree.*

We can now state the main algorithm of this chapter.

Algorithm 4.4.4.

Input: A 3-connected graph G and a fixed size subgraph A in G with no 3-cut separating A.

Output: Either

- a pruned K_5-tree of G, or
- "a K_5-model attached to A exists".

Running time: $O(|V(G)| + |E(G)|)$.

We refer to A as the *root subgraph* or simply *root* of G (as it is always contained in the root graph node of a pruned K_5-tree).

4.5 Pruned $(3, 3)$-tree

Before we begin our discussion of Algorithm 4.4.4, we note that we can build a more interesting tree decomposition (that is not always a star) in case an attached K_5-model does not exist. Indeed, since the label G_r of the root of a pruned K_5-tree is K_5-minor free, we can apply the main result of our previous section (Algorithm 3.0.55) to G_r to further decompose it along $(3, 3)$-cuts into graphs which are planar or L. Furthermore, by construction, the set $\mathcal{F}$ of 3-cuts obtained from the union of $(3, 3)$-cuts of G_r and 3-cuts in cutting nodes of this pruned K_5-tree are laminar. We call this $\mathcal{F}$-block tree the *pruned $(3, 3)$-tree.*

Thus, as we just discussed, we can combine Algorithm 4.4.4 and 3.0.55 into the following algorithm.

Algorithm 4.5.1.

Input: A 3-connected graph G^* and an fixed size subgraph A in G^* with no 3-cut separating A.

Output: Either

- a pruned $(3,3)$-tree of G, or
- "a K_5-model attached to A exists".

Running time: $O(|V(G)| + |E(G)|)$.

All internal nodes of the pruned $(3,3)$-tree are either planar or L. We now delve into the details of Algorithm 4.4.4.

4.6 Main algorithm

4.6.1 Overview

In this section, we give an overview of the main result of this chapter, Algorithm 4.4.4, a linear time algorithm which finds a pruned K_5-tree for 3-connected graphs with no attached K_5-model.

In each iteration, our recursive algorithm either determines an attached K_5-model exists or recursively applies itself on a set of smaller 3-connected graphs. Having solved the problem on these smaller graphs, it uses the solution to these new problems to construct the solution to the old problem quickly. Our reductions will reduce the problem so much that the entire algorithm runs in linear time provided the reduction and construction subroutines take linear time.

Many reductions are similar to those used in our linear time K_5-minor recognition algorithm.

One reduction consists of deleting a stable set of vertices of degree at least four such that for every pair of neighbours of every vertex deleted we leave a set of four common

neighbours, where these sets are disjoint (note that this implies that the neighbours of a deleted vertex form the center of a clique subdivision which combined with the edges to the deleted vertex leaves a clique minor of order at least 5). When doing so, we also add edges to make the neighbourhood of each deleted vertex a clique. It is not hard to prove that given an attached K_5-model in the resultant graph, we can easily find one in G.

Furthermore, we can easily extend a pruned K_5-tree for the smaller graph to a decomposition tree for G. For deleted vertices of degree at least 5, we simply add the deleted vertex to the K_5-model node containing the clique minor given by the subdivision whose centers are its neighbours. This will also work for a vertex of degree 4 if the clique on its neighbourhood in the smaller graph extends to a K_5-model.

In our second reduction, we contract the edges of a matching M with the properties that:

(i) the resultant minor G_M of G is 3-connected, and

(ii) every vertex of G has degree bounded by a constant d specified below.

We will then solve the problem on G_M and exploit this solution to find the solution in G. If an attached K_5-model exists in G_M, it is easy to see that the uncontraction of this attached K_5-model is an attached K_5-model in G. If the algorithm returns a pruned K_5-tree of G_M, then after the uncontraction we must check if there are new K_5-models, and find new top detachers. In doing so, we exploit the fact that each graph node in the decomposition tree for G_M is planar or small, much as was done in Chapter 3.

We make a third reduction if there is a larger stable set S of vertices of degree 3 and each of which has the same neighbourhood as at least 4 vertices in $V(G) - S$. In this case, we reduce by deleting each vertex in the stable set, replacing it with a triangle on its neighbourhood. If an attached K_5-model K exists in this smaller graph, an attached K_5-model (not necessarily obtained from K) also exists in G. Given a pruned K_5-tree for the smaller graph, for each deleted vertex v, we either

(i) add it to a K_5-model node,

(ii) place it in the face of the graph node formed by the triangle on its neighbourhood, or

(iii) add it and its neighbourhood as a graph node adjacent to the cutting node containing its neighbours (such a cutting node exists since $N(v)$ is a $(3,3)$-cut in $G - S$ due to 4 other vertices of degree 3 with neighbourhood $N(v)$).

We can perform a similar reduction if we find a large stable set S of degree 4 vertices, each of which has the same neighbourhood as at least 4 vertices in $V(G) - S$.

We make our final reduction if we find a small dense minor of G. In this case, we recurse on this minor. We use this recursive solution to find a large set S of special "irrelevant" vertices (vertices in K_5-model nodes whose removal does not destroy the corresponding K_5-model) and delete them in G. We recurse on the resulting graph. If an attached K_5-model exists in $G - S$ then our choice of S guarantees one also exists in G. Otherwise, as in our first reduction, we can add S to the pruned K_5-tree of $G - S$ to obtain the pruned K_5-tree of G.

For our final reduction, we use Frank, Ibaraki and Nagamochi's algorithm [28] to determine which edges of the small minor are "irrelevant" (which we then translate into "irrelevant" vertices of G). Their algorithm finds a subset of at most $4n$ edges of an input 3-connected graph G which yield a 3-connected subgraph H in which for every pair of vertices the set of 3-cuts which separate them are the same. Specifically, we apply the following corollary to the main result of [28] in which they find a graph G_k with at most kn edges in linear time.

Algorithm 4.6.1. [28]

Input: A 3-connected graph G.

Output: A 3-connected subgraph H of G with at most $4|V(G)|$ edges in which for every pair of vertices the set of 3-cuts which separate them are the same as in G.

Running time: $O(|V(G)|)$

Corollary 4.6.2. *Let G be a graph and let G_4 be defined as in [28]. Then G and G_4 have the same set of $(3,3)$-cuts and thus the shape of their pruned K_5-trees is the same (that is, the trees are the same and the vertex set inside corresponding nodes are the same).*

4.6.2 Formal description

Having described the algorithm informally we are now ready to present a formal specification. Throughout the chapter, $d = 100000$ and let $\varepsilon = d^{-6}$. Our algorithm uses a number of subroutines with these specifications. We label them to indicate where they can be found in this chapter.

The first subroutine is used to determine which reduction to make.

Algorithm 4.7.1.

Input: A 3-connected graph G.

Output: One of the following.

1. A graph F with at most $\frac{16|V(G)|}{d}$ vertices, and at least $2000\frac{16|V(G)|}{d}$ edges, and a subdivision of F in G such that every edge is subdivided at most once, and the vertices which are the midpoints of a subdivided edge are a stable set of G.
2. A matching M in the subgraph of G induced by the vertices of degree at most d which has size at least $d^4\varepsilon|V(G)|$.
3. A stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree at least 5 and at most d such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct.
4. A stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 4 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct.
5. A stable set of at least $\frac{16}{d}|V(G)|$ degree 3 vertices in G.
6. $8|V(G)|$ edges in G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.9.1.

Input:

- A graph G,
- a root A in G,
- a graph F with at most $\frac{16|V(G)|}{d}$ vertices, and at least $2000\frac{16|V(G)|}{d}$ edges, and a subdivision of F in G such that every edge is subdivided at most once, and the vertices which are the midpoints of a subdivided edge are a stable set of G. (i.e., output (1) of Algorithm 4.7.1.)

Output:

- A 3-connected graph F' with $|E(F')| = |E(F)|$ and $|V(F')| \leq 15|V(F)|$.
- A subdivision of F' in G such that every edge is subdivided at most once, and the vertices in the center of the subdivided edges are a stable set of G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.9.5.

Input:

- A graph G,
- a root A in G,
- A 3-connected graph F' with $|E(F')| = |E(F)|$ and $|V(F')| \leq 15|V(G)|$.
- A subdivision of F' in G such that every edge is subdivided at most once, and the vertices in the center of the subdivided edges are a stable set of G.

Output: Either "a K_5-model attached to A exists" (in G) or

- A set R of $27\varepsilon' n$ vertices in K_5-models of G, and
- a graph H obtained from G by deleting these vertices.

Algorithm 4.10.1.

Input:

- A graph G,

- a root A in G,
- a matching M in the subgraph of G induced by the vertices of degree at most d which has size at least $d^4\varepsilon|V(G)|$.

Output: Either

- a matching N in G of size at least $4\varepsilon n$ among vertices of degree at most ε^{-1} such that G/N is 3-connected, or
- a set of at least εn laminar 3-cuts in G and for each cut X, a component C of $G - X$ of size at most ε^{-1} disjoint from A and a 3-connected graph H obtained by removing all such components C and adding a triangle onto each such cut X.
- a set of at least εn laminar 4-cuts in G and for each cut X, a component C of $G - X$ of size at most ε^{-1} disjoint from A and a 3-connected graph H obtained by removing all such components C and adding some edges between vertices of each such cut X.

Running time: $O(|V(G)| + |E(G)|)$.

The following subroutines are needed for our post-recursion steps.

Algorithm 4.11.1.

Input:

- A graph G,
- a root A in G,
- a stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree at least 5 and at most d such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct, and
- a pruned K_5-tree of $H = G - S$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.14.1.

Input:

- A graph G,
- a root A in G,
- a stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 4 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct, and
- a pruned K_5-tree of $H = G - S$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.12.1.

Input:

- A graph G,
- a root A in G,
- a stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 3 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct, and
- a pruned K_5-tree of $H = G - S$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.15.1.

Input:

- A graph G,
- a root A in G,
- a set of at least $4|V(G)|$ edges F in G, none of which have endpoints in different components of $G - X$ for any 3-cut X of G, and
- a pruned K_5-tree of $H = G - F$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.16.1.

Input:

- A graph G,
- a root A in G,
- a matching N in G such that $H = G/N$ is 3-connected,
- a pruned K_5-tree of H.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Algorithm 4.18.1.

Input:

- A graph G,
- a root A in G,
- a set of at least εn laminar 3-cuts or 4-cuts in G and for each cut X, a component U_X of $G - X$ of size at most ε^{-1} disjoint from A
- a 3-connected graph H obtained by removing all components U_X and adding some edges between the vertices of X for each X, and
- a pruned K_5-tree of H.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Having completely specified our subroutines, we now describe the main algorithm of this chapter.

Description and analysis. [of Algorithm 4.4.4]

1. If G has fewer than 20 vertices, solve the problem by brute force. Otherwise, proceed as follows.

2. Run Algorithm 4.7.1.
   (a) If it returns **output (1) a graph F and a subdivision of F in G**,
      i. run Algorithm 4.9.1 on the output and run Algorithm 4.9.5 on the graph F' (a small dense minor of G) it outputs.
      ii. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".
      iii. If a set R and a graph H are returned, recurse on H.
      iv. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".
      v. If a pruned K_5-tree of H is returned, return it as a pruned K_5-tree of G.
   (b) If it returns **output (2), a matching M**,
      i. run Algorithm 4.10.1 to obtain either a submatching N or a graph H and a set $\mathcal{F}$ of cuts and for each cut $X \in \mathcal{F}$ a component U_X of $G - X$ of size at most ε^{-1}.
      ii. If we obtain a submatching N,
         A. build $H = G/N$ and recurse on H.
         B. Run Algorithm 4.16.1 on the output and return its output.
      iii. If we obtain a graph H and a set $\mathcal{F}$ of cuts,
         A. recurse on H, run Algorithm 4.18.1 on the output and return its output.
   (c) If it returns **output (3), a stable set S of vertices of degree at least 5**,
      i. build H, the graph obtained from G by adding a clique on the neighbourhood of each vertex in S and removing S.
      ii. Recurse on H.
      iii. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".

iv. If a pruned K_5-tree of H is returned, run Algorithm 4.11.1 and return its output.

(d) If it returns **output (4), a stable set S of vertices of degree 4**,

i. build H, the graph obtained from G by adding a clique on the neighbourhood of each vertex in S and removing S.

ii. Recurse on H.

iii. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".

iv. If a pruned K_5-tree of H is returned, run Algorithm 4.14.1 and return its output.

(e) If it returns **output (5), a stable set S of vertices of degree 3**, is returned,

i. build H, the graph obtained from G by adding a clique on the neighbourhood of each vertex in S and removing S.

ii. Recurse on H.

iii. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".

iv. If a pruned K_5-tree of H is returned, run Algorithm 4.12.1 and return its output.

(f) If it returns **output (6), a set F of edges**, is returned,

i. run Algorithm 4.6.1 on G to delete all but $4|V(G)|$ edges and recurse on the resulting graph H,

ii. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".

iii. If a pruned K_5-tree of H is returned, run Algorithm 4.15.1 and return its output.

□

4.7 Finding the reductions

In this section we take our first step towards finding a structure which allows us to perform one of the reductions described in the previous section. In doing so, we use the following observation:

If G contains a subdivision of F in which every edge is subdivided exactly once and the midpoints of the subdivided edges form a stable set, then there is a model of F in G such that

(i) each vertex image is a star, and

(ii) each edge in the model has at least one endpoint which is a center of the star.

Algorithm 4.7.1.

Input: A 3-connected graph G.

Output: One of the following.

1. A graph F with at most $\frac{16|V(G)|}{d}$ vertices, and at least $2000\frac{16|V(G)|}{d}$ edges, and a subdivision of F in G such that every edge is subdivided at most once, and the vertices which are the midpoints of a subdivided edge are a stable set of G.
2. A matching M in the subgraph of G induced by the vertices of degree at most d which has size at least $d^4\varepsilon|V(G)|$.
3. A stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree at least 5 and at most d such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct.
4. A stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 4 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct.
5. A stable set of at least $\frac{16}{d}|V(G)|$ degree 3 vertices in G.
6. $8|V(G)|$ edges in G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis. We present an algorithm which has two steps. It first finds a maximal matching M amongst the vertices of degree at most d using a greedy algorithm. Obviously, the vertices of M, together with the vertices of degree greater than d form a hitting set B for the edges of G. If this matching has more than $d^4\varepsilon|V(G)|$ edges, then we can return it as output (2).

Otherwise, we turn to the second step, in which we attempt to find a subdivision of a graph F as in (1) whose centers are in B. We assign a vertex to the pair of vertices which will be the endpoints of the edge of F it corresponds to. To do so, for each vertex of $G - B$ in turn, we try to assign it to a pair of its neighbours which have not yet been assigned four other vertices of $G - B$.

By our bound on the number of edges of G, $|B| \leq \frac{16|V(G)|}{d}$, so if we find $2000\frac{16|V(G)|}{d}$ vertices of $G - B$ which can serve as the midpoint of distinct subdivided edges, we return output (1).

Otherwise, there are at least $\frac{48}{d}|V(G)|$ vertices we failed to assign. If at least a third of these vertices have degree greater than 4, we return output (3). If at least a third of these vertices have degree 4, we return output (4). Otherwise, we return output (5). □

If our preprocessing step returns either output (3), (4) or (5), we are ready to perform the appropriate reduction as described in the last section. If the preprocessing step returns output (2) we need to massage the matching to find a matching N which is not much smaller such that contracting on N yields a 3-connected minor of G. To do so, we use the techniques developed in the previous chapter. If we return output (1) we need to preprocess further. We would essentially like to preprocess by recursively applying our algorithm to the model of F in which the vertex images are stars and the edges of the model all have at least one endpoint which is the center of a star. However, this graph may not be 3-connected, and so again we need to do some massaging. This is described in the next section.

4.8 Pre-recursion: Deleting vertices

If obtain either output (3), (4) or (5) (i.e., a stable set S), we simply delete the vertices of S and recurse on the resulting graph.

4.9 Pre-recursion: Contracting stars

In this section, we describe the algorithm with finds the graph H to recurse on when we obtain output (1) from Algorithm 4.7.1.

4.9.1 Maintaining Connectivity

In this section, we describe an algorithm which takes output (1) (a large set of edges inducing stars) from Algorithm 4.7.1 and produce a new 3-connected graph F'. F' has as many edges and at most 15 times as many vertices as F and a subdivision of F' in G such that every edge is subdivided at most once, and the vertices in the center of the subdivided edges are a stable set of G. (Our main algorithm will then recurse on F'.)

Let N_0 be the vertices in the middle of a subdivided edges of F and C be all other vertices of the subdivision of F in G. Let N_1 be the remaining vertices of G. We may think of output (1) of Algorithm 4.7.1 as a set of edges inducing a union of disjoint stars S_0 whose contraction in $G - N_1$ yield F. We add an arbitrary edge incident to each vertex in N_1 to form a new set S of disjoint stars whose contraction in G yield F and let $N = N_0 \cup N_2$. This ensures uncontracting S in F yields a 3-connected graph (namely G). Then C is the center of these stars (some of which may be trivial) and N is the set of non-centers.

Formally, we describe the following algorithm.

Algorithm 4.9.1.

Input:

- A graph G,
- an induced 4-wheel in G, and
- a graph F with at most $\frac{16|V(G)|}{d}$ vertices, and at least $2000\frac{16|V(G)|}{d}$ edges, and a subdivision of F in G such that every edge is subdivided at most once, and the vertices

which are the midpoints of a subdivided edge are a stable set of G. (i.e., output (1) of Algorithm 4.7.1.)

Output:

- A 3-connected graph F' with $|E(F')| = |E(F)|$ and $|V(F')| \leq 15|V(F)|$.
- A subdivision of F' in G such that every edge is subdivided at most once, and the vertices in the center of the subdivided edges are a stable set of G.

We note that for any cut of F, the union of the images of the vertices in this cut is a cut in G. Since G is 3-connected, it follows that there is no 1-cut or 2-cut which consists of only vertices of N. Our algorithm proceeds in two steps, it first uncontracts some edges which ensure that there are no 1-cut or 2-cuts consisting of only vertices of C and then performs further uncontractions to ensure that there are no 2-cuts consisting of one vertex of N and one vertex of C.

In both steps, we begin by constructing an auxiliary graph with vertex set the current set of centers of stars and with edges representing connections we may have lost due to the contractions we have performed.

In the first step we have an edge for each such connection. That is, for each vertex u which is in some star of which it is not the center, we add an edge between every pair of neighbours of u. This gives us our auxiliary graph H.

To determine which connections are important, we apply Frank, Ibaraki and Nagamochi's algorithm [28] to H in order to reduce it to a graph H' with at most $5|C|$ edges with no new cuts of size at most 5.

Let D be the added edges of $E(H')$ which were not originally from G. We build a set U of vertices which we would like to uncontract. For each edge e of D, add a vertex u with both endpoints of e in its neighbourhood which is in some star of which it is not the center. So we set T_0 to be the stars induced by $E(S) - U$.

After these uncontractions the following holds:

Lemma 4.9.2. *There is no cutset $X \subseteq C$ of size at most 2 in G/T_0*

Proof. Suppose G/T_0 contained such a cutset. Then each component U_i of $G/T_0 - X$ contains at least one vertex of C as otherwise, U_i only contains vertices of N each of which would have degree at most 2 since N is a stable set in G. But all vertices of G/T_0 not in C have degree at least 3 as G is 3-connected.

Since H is 3-connected, so is H' and there is at least one path between U_1 and U_2 in $H' - X$. Let P be a shortest path in $H' - X$ from a vertex of $U_1 \cap V(H')$ to a vertex of $U_2 \cap V(H')$. Since P is a shortest path, it contains exactly one edge from $U_1 \cap V(H')$ to $U_2 \cap V(H')$. But this edge corresponds to a vertex in G/T_0 which connected U_1 and U_2 in $G/T_0 - X$. Contradiction. □

Now we can combine this with our remark stating that no cutset of size at most 2 in G/T_0 is contained entirely in N.

Corollary 4.9.3. *Every 2-cut of G/T_0 contains exactly one vertex of C.*

In other words, the procedure we described above takes an input S and outputs T_0 with the property in Corollary 4.9.3. We formally describe this subroutine before reusing it on T_0 instead of S.

Algorithm 4.9.4.

Input:

- A graph G,
- a set of edges inducing a union of disjoint stars S with centers C and non-centers N in G

Output: A subset T_0 of S with $|T_0| > |S| - 10|C|$ and such that every 2-cut of G/T_0 contains exactly one vertex of that is the center of a star (in T_0).

Description and analysis. 1. For each u in some star of which it is not the center, add an edge between every pair of neighbours of u.

Call the resulting graph H.

2. Frank, Ibaraki and Nagamochi's algorithm [28] to H with connectivity $k = 5$ to obtain a graph H'.
3. Build $D = E(L) - E(G)$.
4. Let U be the empty set.
5. For each $e \in D$, add to U a vertex u with both endpoints of e in its neighbourhood which is in some star of which it is not the center.
6. Build $T_0 = E(S) - U$ and return T_0.

□

In the second step, we build our auxiliary graph with respect to T_0 rather than S and build T instead of T_0.

Now the same argument used in the first step combined with Corollary 4.9.3 applied to G/T_0 shows that G/T is 3-connected. Indeed, all 2-cuts of G/T consist of one center c and one non-center n of a star in T_0. Any such 2-cut $X = \{c, n\}$ was a 1- or 2-cut X' in G/T_0 as all vertices of $G/T_0 = G/T/(E(T_0) - E(T))$ have degree at least 3 and non-centers of a star form a stable set. Note that the non-center n of X contracts to a center of T_0 in G/T_0. Since the only cuts of size at most 2 in G/T_0 contains one center and one non-center, it follows that the center c of X is a non-center of S. But this implies that both c and n were non-centers of S in G. Since they form a 2-cut in G/T, they also form a 2-cut in G contradicting the 3-connectivity of G.

We now formally describe Algorithm 4.9.1.

Description.

1. Run Algorithm 4.9.4 with input G and S to obtain a subset T_0 of S.
2. Run Algorithm 4.9.4 with input G and T_0 to obtain a subset T of T_0.
3. Return T.

4.9.2 Small dense minors

We are now ready to use Algorithm 4.9.1 as a subroutine to find the graph H we recurse on using output (1) of Algorithm 4.4.4.

Let $\varepsilon' = \frac{15 \cdot 16}{d}$

Algorithm 4.9.5.

Input:

- A graph G,
- a root A in G,
- A 3-connected graph F' with $|E(F')| = |E(F)|$ and $|V(F')| \leq 15|V(G)|$.
- A subdivision of F' in G such that every edge is subdivided at most once, and the vertices in the center of the subdivided edges are a stable set of G.

Output: Either "a K_5-model attached to A exists" (in G) or

- A set R of $27\varepsilon' n$ vertices in K_5-models of G, and
- a graph H obtained from G by deleting these vertices.

In this case, this pre-recursion actually includes a recursive call to our main algorithm, Algorithm 4.4.4.

Description and analysis. 1. Run Algorithm 4.4.4 on F'.

2. If "a K_5-model attached to A exists" is returned, return "a K_5-model attached to A exists".
3. If a pruned K_5-tree of F' is returned,
   (a) initialize R to be the empty set.
   (b) For each K_5-model node t of the pruned K_5-tree with parent cutting node s,
      i. add all vertices of $F'_t - F'_s$ to R.
   (c) Return $H = G - R$.

□

For our algorithm to run in linear time, we will need H to be small compared to G. More precisely, we need H to have at most $(1-2\varepsilon')n$ vertices. We now prove this.

Lemma 4.9.6. *H has at most $(1-2\varepsilon')n$ vertices.*

Proof. Let F' be the 3-connected graph output by Algorithm 4.9.1. Since F (the input to Algorithm 4.9.1) has at most $\frac{16|V(G)|}{d}$ vertices, F' has at most $\frac{15\cdot 16|V(G)|}{d} = \varepsilon' n$ vertices.

By Lemma 1 of [56], in the pruned K_5-tree of F', there are at most $64\varepsilon' n + 9\varepsilon' n = 73\varepsilon' n$ vertices whereas F' has at least $100\varepsilon' n$ subdivided edges. So at least $27\varepsilon' n$ of these edges lie in a K_5-model node. The vertices that subdivide these edges are added to R and removed from G. So H has at most $(1-2\varepsilon')n$ vertices as required.

□

4.10 Pre-recursion: Contracting a matching

In this section, we describe Algorithm 4.10.1 which takes output (2) (a large induced matching) from Algorithm 4.7.1 and produce either a submatching N whose contract yields a 3-connected graph H or a set of 3-cuts or 4-cut in G, each of which has a decomponent of size at most ε^{-1} not containing A.

Algorithm 4.10.1.

Input:

- A graph G,
- a root A in G,
- a matching M in the subgraph of G induced by the vertices of degree at most d which has size at least $d^4\varepsilon|V(G)|$.

Output: Either

- a matching N in G of size at least $4\varepsilon n$ among vertices of degree at most ε^{-1} such that G/N is 3-connected, or

- a set of at least εn laminar 3-cuts in G and for each cut X, a component C of $G - X$ of size at most ε^{-1} disjoint from A and a 3-connected graph H obtained by removing all such components C and adding a triangle onto each such cut X.
- a set of at least εn laminar 4-cuts in G and for each cut X, a component C of $G - X$ of size at most ε^{-1} disjoint from A and a 3-connected graph H obtained by removing all such components C and adding some edges between vertices of each such cut X.

Running time: $O(|V(G)| + |E(G)|)$.

This reduction is similar to the one in the previous chapter (in Section 3.4.2). If we fail to find a submatching, we return a large set of 3-cuts on which we decompose G. The only difference in this case is that we cannot simply return a K_5-model if we find one. However, we also no longer need an edge inside each 3-cut and can also return 4-cuts attached to a K_4-minor, which is needed in some cases.

We now provide the formal details of Algorithm 4.10.1.

Description and analysis. Let $S = f(M)$ the vertices obtained by contracting M and $H = G/M$. By Lemma 3.3.10 applied to H and S, either

1. T has at least $|S|/15$ leaves, or
2. there is a subset $S' \subseteq S$ of vertices of size at least $|S|/15$ no two of which are in a 2-cut of H.

In the second case, we return $f^{-1}(S')$ as N. Thus, we may assume T has at least $|S|/15 \geq 30d^2\varepsilon n > 4\varepsilon n$ leaves.

Since the 2-block tree for H has at least $4\varepsilon n$ leaves, it has at least $2\varepsilon n$ leaves such that the leaf L has at most ε^{-1} vertices (since the graphs corresponding to these leaves are disjoint, except for the cut Y).

Consider a leaf of T containing at most ε^{-1} vertices of H cut off by Y from the rest of H and the corresponding subgraph L of G obtained by uncontracting the edges of M' in it. We distinguish two cases.

Case 1: If $f^{-1}(Y)$ has only three vertices then we simply note that $f^{-1}(Y)$ is a 3-cut containing an edge separating L from a component of $G - f^{-1}(Y)$ containing the rest of G. If for more than half the cuts Y, $|f^{-1}(Y)| = 3$, we return this family $f^{-1}(Y)$ for each such Y and the corresponding leaf L of size at most ε^{-1}. This family contains at least $|S|/30 > \varepsilon n$ cuts and are laminar since the cuts in cutting nodes of T are laminar

Case 2: If $f^{-1}(L)$ contains a 3-cut then we proceed as above and return this 3-cut (instead of $f^{-1}(Y)$) and the component of $G - f^{-1}(Y)$ completely contained in L.

Case 3: If $f^{-1}(Y)$ has four vertices and $f^{-1}(L)$ contains no 3-cut then, since G is 3-connected and no vertex of degree at most five in G has neighbours in two different matching edges we see that $f^{-1}(L)$ has at least 7 vertices.

We claim there are two vertex disjoint paths between the two matching edges wz and xy of $f^{-1}(L)$. If some vertex a separates $\{x, y\}$ from $\{w, z\}$ in $f^{-1}(L)$ then either $\{w, x, a\}$ or $\{x, y, a\}$ is a 3-cut in $f^{-1}(L)$, a contradiction. So by Menger's theorem, there are two vertex disjoint paths from $\{x, y\}$ to $\{w, z\}$. Without loss of generality, these paths are from x to w and y to z.

We now use brute force to determine there are also vertex disjoint paths in $f^{-1}(L)$, one from x to z and the other from w to y (we solve this instance of 2-DRP since $f^{-1}(L)$ has size at most ε^{-1}). If so, we remove $f^{-1}(L) - f^{-1}(Y)$ and add edges so $f^{-1}(Y)$ forms a clique. If not, since $f^{-1}(L)$ contains no 3-cuts, it is actually planar (since the auxiliary graph $f^{-1}(L)^*$ to this new instance of 2-DRP is planar). In this case, we remove $f^{-1}(L) - f^{-1}(Y)$ and add edges so w, x, y, z is a 4-cycle and add a vertex x_L adjacent to w, x, y, z.

If for more than half the cuts Y, we are in this case, we return this family $f^{-1}(Y)$ for each such Y and the corresponding leaf L of size at most ε^{-1}. This family contains at least $|S|/30 > \varepsilon n$ cuts and are laminar since the cuts in cutting nodes of T are laminar

□

4.11 Post-recursion: Adding vertices of degree at least 5

In this section, we give the details of Algorithm 4.14.1.f

Algorithm 4.11.1.

Input:

- A graph G,
- a root A in G,
- a stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 4 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct, and
- a pruned K_5-tree of $H = G - S$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

We start by showing that when we perform reduction (4) and determine that a K_5-model attached to A exists in H then a K_5-minor attached to A also exists in G.

Lemma 4.11.2. *Let G be a 3-connected graph, A an induced 4-wheel in G, x a degree $\ell > 4$ vertex in G and K is a K_ℓ-subdivision in G with centers $N(x)$ and edges subdivided exactly once.*

Let H be the graph obtained from G by deleting x and adding a clique onto $N(x)$.

If H has a K_5-minor L attached to A then so does G.

The proof formalizes the idea that if a vertex is attached to a K_5-model K then it is in a K_5-model node of either the K_5-model node for K or a K_5-model node with a closer detacher.

Proof. Choose an arbitrary K_5-minor K' obtained by deleting vertices from K.

Suppose the lemma is false then in particular, K' is not attached to A.

Since G is 3-connected, by Theorem 4.2.2, there is a closest detacher X of size 3 for K'.

$V(K') \cap V(H)$ is contained in some component U of $H - X$ which does not completely contain some vertex image of L (otherwise, X is a detacher for L contradicting our assumption that L is attached).

Thus, all vertex images of L intersecting $U \cup X$ also intersect X. We now change the vertex images of L intersecting X so they use edges of G rather than H. To do so, we only need to find a K_3-minor attached to X in U.

This is easy to find as we can use K' as an attached minor since X is a closest detacher (it has size greater than 3 which is more than what we need).

More precisely, since X is a closest detacher of K', we can find 3 paths P_1, P_2, P_3 in $H[U \cup X]$ from X to the centers of K' (i.e., $N(x)$). P_1, P_2, P_3 are also paths in G as they do not use edges between neighbours of x. If none of P_1, P_2, P_3 contains a subdivided edge of K' then

This re-routed minor we obtained from L is a K_5-minor attached to A in G, a contradiction. □

Lemma 4.11.3. *Deleted vertices x of degree at least 5 are in K_5-minors nodes of G.*

Proof. Their neighbourhood is a clique in H so $N(x)$ is a clique of size and least 5 in H and thus appears in a K_5-model node. $N(x)$ is the center of the (same) complete graph minor in G and x has $\deg(x) > 4$ path to it so no detacher separates X from this complete graph minor. Therefore, x is in the same K_5-model node. □

Lemma 4.11.4. *Adding vertices x of degree at least 5 and deleting the clique on their neighbourhood does not destroy any detachers (i.e., every detacher is still the closest detacher to some K_5-minor).*

Lemma 4.11.5. *Let G be a 3-connected graph, A an induced 4-wheel in G, x a degree $\ell > 4$ vertex in G and K is a K_ℓ-subdivision in G with centers $N(x)$ and edges subdivided exactly once.*

Let H be the graph obtained from G by deleting x and adding a clique onto $N(x)$.

If H has a top K_5-model L with closest detacher Y then so does G.

Proof. Choose an arbitrary K_5-minor K' obtained by deleting vertices from K.

Suppose the lemma is false then in particular, K' is not a K_5-minor with closest detacher Y. K' has a closest detacher X further from A since the centers of K' is a K_5 (and thus a K_5-minor K'_H) in H.

$V(K'_H)$ is contained in some component U of $H - X$ which does not completely contain some vertex image of L (otherwise, X is a detacher for L contradicting our assumption that Y is the closest detacher of L).

Thus, all vertex images of L intersecting $U \cup X$ also intersect X. We now change the vertex images of L intersecting X so they use edges of G rather than H. To do so, we only need to find a K_3-minor attached to X in U.

This is easy to find as we can use K' as an attached minor since X is a closest detacher (it has size greater than 3 which is more than what we need). □

4.12 Post-recursion: Adding degree 3 vertices

In this section, we assume that we removed a stable set of degree 3 vertices and added a triangle onto either neighbourhood to obtain H. We now add back these vertices given a pruned K_5-tree $[T, \mathcal{H}]$ for H (and remove edges in their neighbourhoods that we added).

Algorithm 4.12.1.

Input:

- A graph G,
- a root A in G,
- a stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 3 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct, and

- a pruned K_5-tree of $H = G - S$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

We further assume that for each removed vertex v, there are three (remaining) vertices, each adjacent to a pair of neighbour of v.

If any such vertex has all neighbours in the K_5-minor free graph H_r indexing the root of T then we simply add these vertices to the root (and remove any edges we added in their neighbourhood).

We claim H_r is still K_5-minor free. Suppose we add back v and $H_r + v$ has a K_5-model K. Then $\{v\}$ is not a vertex image of K since it only has degree 3. But then we can replace v in any vertex image X by $X \cap N(v)$ in K to obtain a K_5-model in H_r, a contradiction.

We add all vertices with all neighbours in H_r first.

If not all neighbours of v are in H_r, since $N(v)$ is a triangle in H, $N(v)$ is completely contained in some (K_5-minor) node H_t of T. We simply add v to (the list of vertices of) this K_5-model node.

We claim G_t, the graph obtained from $H_t + v$ by removing some edges of $N(v)$ is still a K_5-model node with the same closest detacher.

Lemma 4.12.2. *Let G_t be the graph obtained from $H_t + v$ by removing some edges of $N(v)$. Then G_t is still a K_5-model node with the same closest detacher (i.e., G_t contains an attached K_5-model with the same detacher).*

Proof. Let $N(v) = \{x, y, z\}$. Suppose K is a K_5-model in H with closest detacher H_s. Then we can choose K so three vertex images of K, say X_1, X_2, X_3 all intersect H_s (or H_s is not the closest detacher). $N(v)$ is not H_s (or v has all neighbours in H_r). Suppose $|N(v) \cap H_s| \leq 1$.

If two of X_1, X_2, X_3 intersect $N(v)$, say at x, y we can remove the edge between them in $N(v)$ and still obtain (the same) K_5-model. This is a K_5-model in G_t (if we contract vz).

If at most two vertex images intersect $N(v)$, again we obtain the same K_5-model by remove an edge in $N(v)$ no contained in a vertex image.

So three vertex images intersect $N(v)$ and at most one of them is X_1, X_2 or X_3. Without loss of generality, $x \in X_3, y \in X_4, z \in X_5$. Recall there are three vertices in H, v_{xy} adjacent to x to y, v_{yz} adjacent to y to z and v_{zx} adjacent to z to x. Since, $|N(v) \cap H_s| \leq 1$, all three vertices are also in H_t.

If v_{xy} is not in $X_1 \cup X_2$ then we can put v_{xy} in X_3 (or X_4) and v in X_5 to obtain a K_5-model with closest detacher H_s in G.

If v_{yz} is not in $X_1 \cup X_2$ then we can put v_{yz} in X_4 (or X_5) and v in X_3 to obtain a K_5-model with closest detacher H_s in G.

If v_{zx} is not in $X_1 \cup X_2$ then we can put v_{zx} in X_5 (or X_3) and v in X_4 to obtain a K_5-model with closest detacher H_s in G.

So $\{v_{xy}, v_{yz}, v_{zx}\} \subseteq X_1 \cup X_2$. Without loss of generality, two of them are in X_1. If $v_{xy}, v_{yz} \in X_1$ and $v_{yz} \notin H_s$, we can put v_{yz} in X_5 and v in X_3 to obtain a K_5-model with closest detacher H_s in G. If $v_{xy}, v_{yz} \in X_1$ and $v_{xy} \notin H_s$, we can put v_{xy} in X_4 (or even X_3) and v in X_5 to obtain a K_5-model with closest detacher H_s in G.

A symmetric argument shows we obtain a K_5-model with closest detacher H_s in G when $v_{zx}, v_{yz} \in X_1$ and when $v_{xy}, v_{zx} \in X_1$ (we did not use the fact that X_3 intersects H_s in our argument).

Thus, in all cases where $|N(v) \cap H_s| = 1$, we obtain a K_5-model with closest detacher H_s in G.

Now suppose $|N(v) \cap H_s| = 2$. Say, $x, y \in H_s$ and therefore $xy \in E(H_t)$. Again, we contract the edge between v and $N(v) - H_s$ to form a triangle on $N(v)$ so again the same K_5-model exists in G_t. □

Similarly, we can show if H contains a K_5-model attached to A then so does G.

4.13 Locally bounded treewidth

For the remaining cases of our post-recursion algorithms, we exploit the "bounded local treewidth" of planar graphs, essentially using a stronger version of Theorem 2.6.46. This method was initially used by Baker[5] to give a PTAS for finding the maximum sized independent set in a .

Lemma 4.13.1. *[5] A planar graph G with diameter d has treewidth at most $3d$ and we can find a tree decomposition of width $3d$ for G in $O(dn)$.*

Definition 4.13.2. We say a class of graphs have *locally bounded treewidth* if there exists a function f such that for any graph in this class of diameter d has treewidth at most $f(d)$.

We use Eppstein's construction of layered graphs [26] to solve optimization problems on planar graphs.

Lemma 4.13.3. *([26] Lemma 5.2) Let G be a connected planar graph, $v \in V(G)$ and w a constant. Let T be the breadth-first search (BFS) tree rooted at v. If G_i is the subgraph induced by vertices in levels $iw, iw+1, iw+2, \ldots, (i+3)w-1$ in T (i.e., vertices whose distance from v is between iw and $(i+3)w-1$) then the following holds.*

(1) We can construct all G_i (with $(i+3)w-1$ less than the diameter of G) in $O(m+n)$.

(2) For every vertex $u \in V(G)$, the subgraph G' of G induced by vertices at distance at most w from u is contained in some subgraph G_i.

(3) Every vertex of G is included in at most three subgraphs G_i.

(4) The total size of all G_i is at most $3(m+n)$.

(5) For all i, the treewidth of G_i is at most $3w$.

For completeness, we repeat the proof of this lemma here.

Proof. To construct the G_i's, we simply run breadth first search starting at v. This proves (1).

If u is at distance j from v then j is between $(i+1)w$ and $(i+2)w-1$ (inclusively) for some i. So by definition G_i contains all vertices with distance w of u and (2) holds.

A vertex u at distance j only appears in the three graphs $G_{\lfloor j/w \rfloor + k}$ for k in $\{-2, -1, 0\}$. So (3) holds. Thus every edge also appears in at most three graphs G_i so (4) holds.

Finally each graph G_i is a subgraph of G' formed by removing all vertices at distance greater than $(i+3)w - 1$ from v and contracting all vertices at distance at most $iw - 1$ from v into a single vertex v. All vertices in G' are with distance $3w$ of G' so by Lemma 4.13.1, G' has bounded treewidth and so does G. □

4.13.1 Previous work

The notion of locally bounded treewidth was first introduced by Baker [5] to give a polynomial time approximation scheme for the maximum independent set and other optimization problems on planar graphs.

Her algorithm use a construction different for the one in Lemma 4.13.3 and builds disjoint graphs by removed the kth level of a BFS tree (rather than the overlapping graphs of Lemma 4.13.3). There are k ways of choosing which set of levels to remove, at least one of which only decreases the size of the maximum stable set by a factor of $(k-1)/k$. She proves Lemma 4.13.1 then finds the maximum stable on each remaining component.

Eppstein [26] built on this technique by using the improved construction in Lemma 4.13.3, allowing him to solve graph isomorphism and other similar problems (counting the number of isomorphic copies, finding all copies, count the number of induced copies of a connected graph, etc) exactly in linear time (linear in the size of the input and output).

In fact, he characterized all graphs of locally bounded treewidth [26, 27] ("locally bounded treewidth" was originally named the "diameter-treewidth property" due to the diameter based construction) thus extending his results to all graphs excluding some "apex graphs". *Apex graphs* are planar graphs with a fixed number of vertices (with any neighbourhood) added.

4.13.2 Locally bounded treewidth for face-vertex distance and K_5-minor free graphs

We wish to solve problems on our root graph nodes which is K_5-minor free. Hajiaghayi [36] showed that such graphs have locally bounded treewidth.

Lemma 4.13.4. *[36] Let G be a K_5-minor free graph. Then G has locally bounded treewidth.*

However, we will not be able to use this result directly as we will need "distances" to be calculated in the face-vertex incidence graph (of the graph nodes of the $(3,3)$-block tree of the root graph node of the pruned K_5-tree).

We first see why we can switch from distance in G to distances in $FV(G)$ for a planar graph G. Given G, we can add a vertex inside every face adjacent to all vertices on that face to obtain G'. The resulting graph is planar and thus, by Lemma 4.13.1, subgraphs induced by vertices are within distance d of each other have treewidth bounded by $3d$. Therefore, subgraph of G where vertices are within $d/2$ faces of each other have treewidth bounded by $3d$.

Corollary 4.13.5. *A planar graph G whose face-vertex incidence graph has diameter at most $d/2$ has treewidth at most $3d$.*

In particular, we can replace the graph G_i in Lemma 4.13.3 by vertices within face-distance iw and $(i+3)w-1$ from v. We now explicitly define layers to use the face-vertex distance. We then extend this definition to K_5-minor free graphs.

We need to look in the face-vertex incidence graph in order to find cuts of size X at most 3. In G, there may be no i for which X is contained in G_i (the graph induced by vertices whose distance from v is between iw and $(i+3)w-1$ for some v and some i). However, in $FV(G)$, the following lemma shows all 3-cuts of G corresponds to cuts of size at most 6 in $FV(G)$. In particular, there always exists an i for which X appears in $FV(G)_i$ (for $w > 3$).

Lemma 4.13.6. *Let X be a 3-cut in a 3-connected planar graph G. Then all vertices of X are within distance 2 of each other in $FV(G)$.*

Proof. Since X is a cut, there is a closed curve intersecting only the vertices of X and no other vertex or edge of G (since embedding G in the plane and removing the points corresponding to the vertices of X yields a disconnected drawing). The faces and vertices that this curve crosses induce a cycle (of length 6) in the $FV(G)$. Therefore, two vertices of X are at distance at most two from each other by going through the shortest side of the cycle. □

We now wish to move across graph nodes. We've seen in Section 2.6.6 that the clique sum of two graphs of bounded treewidth yields a graph of treewidth no greater than the two graphs composing it.

Thus, even if we recombine all graph nodes, the graphs G_i defined in Lemma 4.13.3 still have bounded treewidth. This property is sufficient for our reduction.

4.13.3 Definitions

We formalize some concepts introduced in this section.

Definition 4.13.7. Let G be a planar graph and $u, v \in V(G)$. The *face-vertex distance* is the distance between u and v is the distance between u and v in $FV(G)$ (i.e., the number of edges in a shortest path from u to v in $FV(G)$). We denote this value by $d_G(u, v)$. When clear from the context, we omit the subscript G.

Definition 4.13.8. Let G be a connected planar graph, $v \in FV(G)$ and $w > 10$ a constant. Let T be the breadth-first search (BFS) tree rooted at v in $FV(G)$. The *ith level* is the set of vertices at the ith level in the BFS tree (i.e., all vertices at distance i from v in $FV(G)$).

By *layer* i (or the *ith layer*), we mean the subgraph G_i induced by vertices in levels $iw, iw + 1, iw + 2, \ldots, (i + 3)w - 1$ in T (i.e., vertices whose distance from v is between iw and $(i + 3)w - 1$) then the following holds.

Lemma 4.13.9. *If $|X| \leq 3$ is a cut in G then there exists a layer G_i such that X appears in G_i.*

Proof. This follows immediately from Lemma 4.13.6 as the vertices of X are within distance 2 of each other (and any vertices within distance $w > 10$ from each other is contained in some layer) □

4.13.4 Layers in 3-connected K_5-minor free graphs

We extend the notion of layers explicitly to K_5-minor free graph. We first define the analogue of a face-vertex incidence graph. We still denote this graph $FV(G)$. We choose the definition of $FV(G)$ so that the distance function d_G in this graph satisfies the following property.

There exists a function f such that for all K_5-minor free graphs G, for all k and for all $v \in V(G)$, the graph induced by all vertices within distance k of v (i.e., $G[\{u|d_G(v,u) \leq k\}|]$) has treewidth at most $f(k)$.

For technical reason, we first define the face-vertex incidence graph of L (denoted $FV(L)$) to be the graph in Figure 4–4.

Definition 4.13.10. $FV(L)$ is the graph in Figure 4–4.

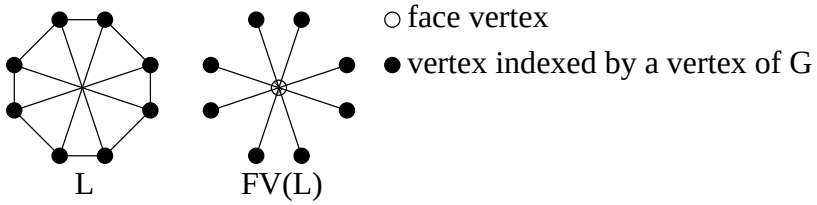


Figure 4–4: L and $FV(L)$

Definition 4.13.11. Let G be a 3-connected K_5-minor free graph. The *face-vertex incidence graph* denoted $FV(G)$ is the graph obtained from the disjoint union of $FV(G_t)$ for each graph node t of the $(3,3)$-block tree of G by

- relabelling all vertices with neighbourhood $V(G_s)$ for some cutting node s to G_s, and

- replacing all vertices labelled the same (either as a vertex of G or a newly created vertex from the previous step) to a single vertex whose neighbourhood is the union of the neighbourhoods of replaced vertices.

Let d_G be the distance between two vertices in $FV(G)$.

We immediately obtain the following corollary from Corollary 4.13.5 and the fact that the clique sum of two graphs of bounded treewidth yields a graph of treewidth no greater than the two graphs composing it (seen in Section 2.6.6).

Corollary 4.13.12. *A 3-connected K_5-minor free graph G whose face-vertex incidence graph has diameter at most $d/2$ has treewidth at most* $\max(8, 3d)$.

We can now use $FV(G)$ to define a *k-layering* which is essentially Eppstein's construction where we replace the usual distance function in a graph G by distances in $FV(G)$.

Definition 4.13.13. By a *k-layering of G, v*, we mean the subgraphs G_i where G_i is induced by vertices in levels $ik, ik+1, ik+2, \ldots, (i+3)k-1$ in the BFS tree T of G rooted at v (i.e., vertices whose distance from v is between ik and $(i+3)k-1$ in G).

By a *k-layering of G, v in $FV(G)$*, we mean the subgraphs G_i where G_i is induced by vertices in levels $ik, ik+1, ik+2, \ldots, (i+3)k-1$ in the BFS tree T of $FV(G)$ rooted at v (i.e., vertices whose distance from v is between ik and $(i+3)k-1$ in $FV(G)$).

The graph G_i is called the *ith layer* of the k-layering (or k-layering in $FV(G)$).

When the root vertex v is not of particular importance, we omit it refer to a *k-layering of G* to mean a k-layering of G, v for a arbitrary $v \in V(G)$ (and we refer to a *k-layering of G in $FV(G)$* to mean a k-layering of G, v for a arbitrary $v \in V(G)$).

Note the only different between a k-layering of G, v and a k-layering of G, v in $FV(G)$ is the graph in which we compute distances.

We now describe how to build $FV(G)$ in linear time when G is 3-connected.

Algorithm 4.13.14.

Input: A K_5-minor free graph G.

Output: $FV(G)$

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis. 1. Build the block tree $[T, \mathcal{G}]$ of G.

2. For each block G_t, build $FV(G_t)$ and set $H_t = FV(G_t)$.
3. For each cutting node s in postorder(G).
4. (a) Set $FV(G_s)$ to be a star centered at a new vertex v_s with neighbours $V(G_s)$.
   (b) For each neighbour t of s, find the vertex adjacent to all of G_s in $FV(G_t)$ (corresponding to the face of the triangular face G_s) and change the label of that vertex to v_s (so they are now labelled the same).
5. Build H, the disjoint union of all graph nodes.
6. For each cutting node G_s,
7. (a) For each $v \in G_s$, identify all vertex in H corresponding to v in H (i.e., delete all these vertices and replace them by a single vertex whose neighbourhood is the union of the neighbourhood of deleted vertices).
8. Return H.

Note that we can perform the identification operation by simply adding a spanning tree on all vertices we wish to identify and then contract this spanning tree. If we add edges for all spanning trees first and then perform all contractions at once, we can complete the last step of the algorithm in linear time. □

Having built $FV(G)$, we can easily build a layering

Algorithm 4.13.15.

Input: A K_5-minor free graph G and an integer k.

Output: A k-layering of G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis. 1. Run Algorithm 4.13.14 to build $FV(G)$.

2. Run BFS from any vertex of $v \in FV(G)$ to compute distances from v to all other vertices in $FV(G)$.
3. Return all layers (subgraphs G_i of G induced by vertices at distance $ik, ik+1, ik+2, \ldots, (i+3)k-1$).

□

4.13.5 Extending layers to non-3-connected K_5-minor free graphs

We can easily extend the definition of $FV(G)$, our distance function and therefore layers to K_5-minor free graphs that are not 3-connected. We do so using the block tree and strong 2-block tree. We simply merge their graph nodes in the same way (it is even simpler as there is no special vertex to add to each cutting node before hand since they no longer correspond to faces).

Definition 4.13.16. Let G be a 2-connected K_5-minor free graph. The *face-vertex incidence graph* denoted $FV(G)$ is the graph obtained from the disjoint union of $FV(G_t)$ for each graph node t of the strong 2-block tree of G by identifying all vertices labelled the same.

Let G be a connected K_5-minor free graph. The *face-vertex incidence graph* denoted $FV(G)$ is the graph obtained from the disjoint union of $FV(G_t)$ for each graph node t of the block tree of G by identifying all vertices labelled the same.

Again, we obtain the same bound on the treewidth of such graphs.

Corollary 4.13.17. *A K_5-minor free graph G whose face-vertex incidence graph has diameter at most $d/2$ has treewidth at most* $\max(8, 3d)$.

4.13.6 Red-blue colouring

We now use ideas introduced in this section to find a pruned K_5-tree of a graph G with locally bounded treewidth. We reformulate the problem of finding the graph nodes of a graph G as a *red-blue colouring* problem where the root node is blue and K_5-model node, with the closest detacher separating them removed, are red.

In this section, we show how to solve this reformulated problem in linear time if G has locally bounded treewidth. We then use this method in the post-recursion step of the remaining reductions of our main algorithm.

Definition 4.13.18. A *red-blue colouring* of (G, A, R) is a 2-colouring of G using colours red and blue with the following properties.

- All vertices in R are red.
- The subgraph induced by blue vertices is connected.
- A is blue or all vertices are red.
- Every component of red vertices (called "red components") is adjacent to at most 3 blue vertices.
- The graph obtained from the blue subgraph by adding a clique onto each cut is K_5-minor free.
- Subject to the above, the number of blue vertices

We now describe how an algorithm for obtaining an red-blue colouring for any one layer (which is of bounded treewidth by Lemma 4.13.1) is used to obtain an red-blue colouring for all of G.

Algorithm 4.13.19.

Input:

- A graph G in a class of graphs such that $FV(G)$ has locally bounded treewidth,
- a root A,
- a set R of vertices in G (to be pre-coloured red)

Output: An red-blue colouring of (G, A, R).

Description. 1. Add a vertex v to G adjacent to A.

2. Run algorithm 4.13.15 to build a k-layering $G_0, \ldots, G_\ell$ of G, v (in $FV(G)$).
3. Remove v from G and set $R_{\ell+1}$ to be empty.
4. For i from $\ell - 1$ to 1,

5. (a) Use Algorithm 4.13.20 to obtain a red-blue colouring c of $(G_i, \{x_i^*\}, R_{i+1} \cup (R \cap V(G_i)))$.
   (b) Let R_i be the red vertices of c in $V(G_{i-1}) \cap V(G_{i+1})$ (i.e., all red whose distance from A is between $(i+1)k$ and $(i+2)k$ in the original graph G).
   (c) Let R_i be the set of vertices coloured red in G_{i+1}.
6. Let c be the colouring where any vertex in R_i for some i is red and all other vertices are blue.
7. Return c.

We now show that the colouring c obtain from this algorithm is indeed an red-blue colouring of G.

Analysis. Let c^* be the red-blue colouring of (G, R).

Suppose c and c^* differ. Clearly, all vertices coloured red by c^* are also coloured red by c as otherwise, we found a closer detacher to some red component using our iterative approach contradicting the optimality of c^*.

Let x be a vertex at maximum distance from A on which c and c^* disagree. Then c colours x red and c^* colours it blue. Let G_j be the layer containing all vertices at distance 2 from x in the $FV(G)$. Let K be the red component of c containing x. Let X be the closest 3-cut separating A from K in G. Then by Lemma, X appears in G_j (since $c^*(x)$ is blue). So x should be coloured blue by c. Contradiction. □

To complete the description of our algorithm, we present Algorithm 4.13.20 for finding an red-blue colouring in one layer (which has bounded treewidth). We present an algorithm using a extended monadic second order logic formulation.

Algorithm 4.13.20.

Input:

- A graph G of treewidth ω,

- a root A,
- a set R of vertices in G (to be pre-coloured red)

Output: An red-blue colouring of (G, A, R).

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis. We use an extended monadic second order logic formula and then simply apply Theorem 2.6.47 to obtain a linear time algorithm. We encode the set R and A as labels. We use the shorthand $\mathrm{Red}(x)$ to mean $x \in X_1$, $\mathrm{Red}(X)$ to mean $\forall x \in X, x \in X_1$ and $\mathrm{Blue}(X)$ to mean $\forall x \in X, x \notin R$.

We also need a variant to the predicate K_5-model(X) so that we can test adjacency in the graph where we add cliques on blue detachers. To do so, we simply replace every instance of the subformula $\mathrm{Adj}(x, y)$ by $(\mathrm{Adj}(x, y) \vee xy \in E_1)$ (where E_1 is a set of edges). We call the resulting predicate K_5-model-aux(X)

$$
\begin{aligned}
\theta(X_1) &= \exists E_1(\theta_1(X_1) \wedge \theta_2(X_1) \wedge \theta_3(X_1, E_1) \wedge \theta_4(E_1)) \\
\theta_1(X_1) &= \mathrm{Red}(R) \wedge \forall X\, \mathrm{Blue}(X) \rightarrow \exists Y(X \subseteq Y) \wedge \mathrm{Blue}(Y) \wedge (A \subseteq Y) \wedge \mathrm{Conn}(Y) \\
\theta_2(X_1) &= \nexists u_1, u_2, u_3, u_4, X, v_1, v_2, v_3, v_4\, \mathrm{Red}(X) \wedge (\{u_1, u_2, u_3, u_4\} \subseteq X) \wedge \mathrm{Conn}(X) \\
&\quad \wedge \mathrm{Blue}(\{v_1, v_2, v_3, v_4\}) \\
&\quad \wedge \mathrm{Adj}(u_1, v_1) \wedge \mathrm{Adj}(u_2, v_2) \wedge \mathrm{Adj}(u_3, v_3) \wedge \mathrm{Adj}(u_4, v_4) \\
\theta_3(X_1, E_1) &= \nexists X\, \mathrm{Blue}(X) \wedge K_5 - \mathrm{model} - \mathrm{aux}(X) \\
\theta_4(E_1) &= (xy \in E_1) \rightarrow (\exists u_1, u_2, u_3, X, z\, \mathrm{Red}(X) \wedge (\{u_1, u_2, u_3\} \subseteq X) \wedge \mathrm{Conn}(X) \\
&\quad \wedge \mathrm{Blue}(\{x, y, z\}) \\
&\quad \wedge \mathrm{Adj}(u_1, x) \wedge \mathrm{Adj}(u_2, y) \wedge \mathrm{Adj}(u_3, z))
\end{aligned}
$$

We set the weight function f_1 to be identically 1 on $V(G)$. Our EMS problem consists of minimizing $|X_1|_1$ subject to $\theta(X_1)$. □

Finally, we give an explicit algorithm for transforming a red-blue colouring into a pruned K_5-tree and vice-versa.

Algorithm 4.13.21.

Input: A 3-connected graph G, root A, and red-blue colouring c.

Output: A pruned K_5-tree of G.

Description and analysis.

1. Build B, the set of blue vertices and R, the set of red vertices coloured by c.
2. Build a copy H of G. We alter H throughout the algorithm.
3. For each component red component U,
   (a) Find X_U, the set of (blue) neighbours of U. Since G is 3-connected and c is an red-blue colouring, $|X| = 3$.
   (b) Add edges to H so X_U is a clique.
   (c) Build a cutting node labelled by H and a graph node labelled by $H[U \cup X]$. Add an edge between these two new nodes.
   (d) Remove U from H
4. Build a root node r labelled by H.
5. Add an edge between r and all cutting nodes we built and return the resulting decomposition as a pruned K_5-tree of G.

□

Algorithm 4.13.22.

Input: A 3-connected graph G, root A, and pruned K_5-tree $[T, \mathcal{G}]$ of G.

Output: A red-blue colouring of G.

Description and analysis. Build a colouring c of $V(G)$ where all vertices in the index of root node of T blue and all other vertices are red. Return c. □

4.14 Post-recursion: Adding degree 4 vertices

In this section, we assume that we removed a stable set of degree 4 vertices from G and added a clique onto each neighbourhood to obtain H. For each of the vertex v we removed, we left 6 vertices in H, one adjacent to each pair of neighbours in $N(v)$ (these vertices may also have other neighbours in $N(v)$). We now add back these vertices given a pruned K_5-tree $[T, \mathcal{H}]$ for H (and remove edges in their neighbourhoods that we added).

Algorithm 4.14.1.

Input:

- A graph G,
- a root A in G,
- a stable set S of at least $\frac{16}{d}|V(G)|$ vertices of degree 4 such that for every $v \in S$, we can choose in $V(G) - S - N(v)$ four common neighbours for every pair of neighbours of v such that these choices are all distinct, and
- a pruned K_5-tree of $H = G - S$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

We proceed as in the case for degree 3 vertices and distinguish between the cases where $N(v)$ is completely contained in the root graph nodes of the pruned K_5-tree of H and the case where it is not.

The second case is easier for degree 4 vertices and we begin by showing we can add those vertices. In the first case, $N(v) \cup \{v\}$ together with the 6 vertices adjacent to a pair of neighbours of v form a K_5-model. In this case, we need to find the closest detacher to this new K_5-model (previously, the root was planar). To find all new top detachers, we use the fact that the root graph index has locally bounded treewidth.

We now prove the necessary lemma for the second case.

Lemma 4.14.2. *Suppose v is a degree 4 vertex in G and $N(v)$ is not completely contained in the root graph node H_r of the pruned K_5-tree $[T, \mathcal{H}]$ of H. Let H_t be the unique K_5-model node containing $N(v)$. Suppose further that there are 10 vertices in H_t, each adjacent to a pair of neighbours of v.*

If G_t is the graph obtained from $H_t + v$ by removing some edges of $N(v)$ then G_t contains a K_5-model with closest detacher H_s where s is the only neighbour of t in T.

Proof. Let K be a K_5-model with vertex images $\{X_1, X_2, X_3, X_4, X_5\}$ in H_t with closest detacher H_s. If K' be the K_5-subdivision with centers $N(v) \cup \{v\}$ and subdivided edges the 10 vertices each adjacent to a pair of neighbours of v (edges incident to v are not subdivided).

Suppose the lemma is false and let L be a K_5-model in H_t with closest detacher H_s. Then the closest detacher X of K' (from A in G) is not H_s. Let U be the component of $G - X$ which completely contains a vertex images of K' but not H_s. Since X is not a closest detacher to L, all vertex images of L intersecting U also intersect X.

We aim to replace these at most 3 vertex images in L to form a K_5-model in G_t. We only need to show $U \cup X$ can be contracted into a triangle onto X (i.e., we only need to show there is a K_3-model attached to X in $U \cup X$). But this is easy by Lemma 4.3.4 since K' is a K_5-model attached to X, which is more than what we need.

□

We now consider the set of vertices S where $N(v)$ is completely contained in the root graph node H_r for each $v \in S$. As stated earlier, the guarantees a K_5-subdivision with centers $N(v) \cup \{v\}$ and therefore a K_5-model in a previously K_5-minor free graph H_r. So we need to find the closest detacher to this new K_5-model.

For each $v \in S$, since we added a clique onto $N(v)$ in H, all vertices if $N(v)$ are within distance 2 of each other. In particular, they all appear in some layer. Furthermore, by

Corollary 2.6.6, adding v to all layers where $N(v)$ appears does not increase the treewidth of those layers (beyond 5).

Thus, we can add back all removed vertices to a layer without increasing its treewidth (above 5).

Let G_r be the graph obtained from H_r by adding back the vertices of S and deleting some edges between neighbours of vertices in S. We now describe the algorithm which builds a pruned K_5-tree of G_r.

Algorithm 4.14.3.

Input:

- A K_5-minor free graph H_r, and
- a graph G_r obtained from H_r by adding a stable set S of degree 4 vertices and deleting some edges between vertices of $N(v)$ for each $v \in S$ such that for each $v \in S$, $N(v)$ is the center of a K_4-subdivision in G_r.

Output: A pruned K_5-tree of G_r.

Description and analysis.

1. Run Algorithm 4.13.22 on (G_r, A, S) to obtain an red-blue colouring c of G_r.
2. Run Algorithm 4.13.21 on c and G_r to obtain a pruned K_5-tree of G_r and return it.

□

Lemma 4.14.4. *Let X be a closest detacher of K from A in G and U be the (unique) component of $G - X$ completely containing a vertex image of K but none of A. Then $G[U \cup X]$ has a K_3-model attached to X (i.e., if we contract all edges in $G[U \cup X]$ without both endpoints in X then we obtain a triangle on X).*

Similarly, we can show if H contains a K_5-model attached to A then so does G.

4.15 Post-recursion: Adding $(3,3)$-connectivity preserving edges

In this section, we describe a linear time algorithm which takes the pruned K_5-tree for a graph H and add a set of edges F that do not cross 3-cuts of G (i.e., both endpoints are on the same side of any 3-cut) and outputs a pruned K_5-tree of the resulting graph G.

Algorithm 4.15.1.

Input:

- A graph G,
- a root A in G,
- a set of at least $4|V(G)|$ edges F in G, none of which have endpoints in different components of $G - X$ for any 3-cut X of G, and
- a pruned K_5-tree of $H = G - F$.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

We do not need to do anything if an edge of F is contained in a K_5-minor node (if all of F is in K_5-minor nodes, we return the pruned K_5-tree of H as a pruned K_5-tree of G). Thus, we start by deleting any edge of F that does not have both endpoint in the root node H_r of our pruned K_5-tree for H.

To add edges of F to H_r, we again exploit the locally bounded treewidth of H_r. We first give an main algorithm and fill in the details for the subroutines later. Recall $d(u, v)$ is the distance between u and v in the face-vertex incidence graph.

We start by building the $(3,3)$-block tree of the root node. Since no edge of F crosses a 3-cut, all edges have both endpoints in some (planar) graph node of this $(3,3)$-block tree. So we restrict our attention to adding edges to a (3-connected) planar graph P (rather than K_5-minor free graphs).

If an edge e of F has both endpoints in the same face P then $P + e$ is still planar. Conversely if endpoints of e are in different faces, by Whitney's theorem, adding e creates

a K_5-model. Thus, our algorithm proceeds by first adding a maximal subset of F which preserves the planarity of P and then finds the top detachers for the newly created K_5-models.

The following remark is an immediate corollary of Wagner's theorem for K_5.

Remark 4.15.2. *If G is a 3-connected planar graph we add an edge uv with endpoints not in the same face then there is a K_5-model of $G + e$ (containing e).*

In fact, we partition F into three sets F_1, F_2, F_3 and add them in three rounds (each depending on the graph created in the previous round).

- F_1 is a maximal set of edges of F we can add to faces of planar graph nodes (of the $(3,3)$-block tree of H_r) which preserving their planarity.
- F_2 is the set of edges of $F - F_1$ whose endpoints are at distance at least 6 in the face-vertex incidence graph of their corresponding planar graph nodes (of the $(3,3)$-block tree of $H_r + F_1$).
- F_3 is the set of edges of $F - F_1$ whose endpoints are at distance 4 in the face-vertex incidence graph of their corresponding planar graph nodes (of the $(3,3)$-block tree of $H_r + F_1$).

Algorithm 4.15.3.

Input:

- A K_5-minor free graph H_r,
- a root set A, and
- a set of edges F not in $E(H_r)$.

Output: A pruned K_5-tree for G_r, the graph obtained from H_r by adding all edges in F to H_r.

Description. 1. Build the $(3,3)$-block tree of H_r

2. For each graph node t of the $(3,3)$-block tree, build F_t, the set of edges of F with both endpoints in $H_{r,t}$.

3. For each graph node t of the $(3,3)$-block tree,
   (a) Run Algorithm 4.15.4 to find a maximal subset F'_t of F_t such that $H_{r,t} + F_t$ is planar.
   (b) Add F'_t to $H_{r,t}$ (and H_r).
4. Call the resulting graph H'_r (to avoid confusion with H_r).
5. Build F_1, the union of all F'_t (for all graph node t).
6. Build F_3, the set of edges u, v of $F - F_1$ with $d_{H'_r}(u, v) = 4$.
7. Build $F_2 = F - F_1 - F_3$.
8. For each graph node t of the $(3,3)$-block tree, build $F_{2,t}$, the set of edges of F_2 with both endpoints in $H_{r,t}$.
9. For each graph node t of the $(3,3)$-block tree,
   (a) Run Algorithm 4.15.7 to find an red-blue colouring of $H'_t + F_{2,t}$.
10. Combine all red-blue colouring obtained this way into an red-blue colouring of $H''_t = H'_t + F_2$.
11. Run Algorithm 4.13.21 to obtain the pruned K_5-tree of $H'_r + F_2$ from c'. Let H''_r be the root node of this pruned K_5-tree.
12. Remove from F_3 all edges that do not have both endpoints in H''_r.
13. Run Algorithm 4.15.9 on H''_r, F_3 to obtain a red-blue colouring c'' of H''_r.
14. Run Algorithm 4.13.21 to obtain the pruned K_5-tree of $H''_r + F_3$ from c''.
15. Combine all colourings by colouring red any vertex that is red in any red-blue colouringÀpply Algorithm 4.13.21 to turn this combined red-blue colouring into a pruned K_5-tree.
16. Return the resulting pruned K_5-tree as a pruned K_5-tree of G.

4.15.1 Adding planar edges (F_1)

We describe Algorithm 4.15.4 which takes a 3-connected planar graph $P = G_t$ and a set of edges F'_t not present in P such that both endpoints of any edge in F'_t lie on a face of P

and finds a maximal subset of F_t' that can be added to P while preserving the planarity of P.

We can group the edges of F_1 by the faces containing the endpoint (each edge is only contained in one face as P is planar). Thus, we can find a maximal subset of planarity preserving edges to add to each face in turn. In other words, we only need be able to find a maximal subset of edges to add to a cycle C. We now describe the algorithm to add these edges.

Algorithm 4.15.4.

Input:

- A cycle $C = \{c_1, \ldots, c_n\}$ (with vertices labelled in this order in C),
- a set of edges $E = \{e_1, \ldots, e_k\}$ between vertices of C (not initially present in C.

Output: A maximal subset E' of E where no two edges cross in C (i.e.,).

Running time: $O(\ell + k)$

A greedy quadratic time algorithm which tries to add one edge at a time, checking against the list of already added edges produces the correct output. To speed this algorithm up to a linear time algorithm, we use a stack and consider the edges to add sorted according to their endpoints in the cycle.

Description and analysis. 1. Represent each edge by a pair of indices in C:

   Construct a pair ℓ_i, r_i for each edge e_i with $\ell_i < r_i$ and $e_i = c_{\ell_i} c_{r_i}$.

   We refer to ℓ_i as the left endpoint of e_i and r_i as the right endpoint of e_i.

2. Radix sort the pairs ℓ_i, r_i in increasing order of left endpoints and subject to this, in decreasing order of right endpoints.
3. Initialize F to be empty.
4. Initialize an stack S of indices (to an empty stack).
5. For each pair ℓ, r (in the above sorted order),
6. (a) Pop S until S is empty or $\top(S) > \ell$.

(b) If S is empty or $r \leq \top(S)$, add (ℓ, r) to F and push r onto S.

7. Return the edges corresponding to F.

□

4.15.2 Adding far non-planar edges (F_2)

We describe Algorithm 4.15.7 which takes a 3-connected planar graph $P = G_t$ and a set of edges F_t'' not present in P with far apart endpoints and finds a pruned K_5-tree of $P + F_t''$. We show that if the endpoints of an edge in F_t'' are sufficiently far apart (distance at least 6 in $FV(P)$), it suffices to colour the vertex furthest from A red (without adding the edge).

We first prove a lemma about the cut separating them from A. Recall d is the distance between two vertices in the face-vertex incidence graph.

Lemma 4.15.5. *Let G be a 3-connected planar graph and A a root face of G.*

Suppose $u \in V(G)$ and $v \in V(G)$ are at distance at least 6 from each other in the face-vertex incidence graph of G and no 3-cut separates u from v.

If a 3-cut X with $u \in X$ separates A from v then $d(u, A) \leq d(v, A) - 4$.

Proof. Suppose not then such an X exists and $d(u, A) \leq d(v, A)$. By Lemma 4.13.6, $d(u, x) \leq 2$ for all $x \in X$ and $d(X, v) \geq 4$. Let $x \in X$ minimize $d(x, v)$ and subject to this, minimize $d(x, A)$.

So $d(u, A) \leq d(x, A) + 2$ and $d(v, A) = d(v, x) + d(x, A)$ (since X separates v from A). Combining these, we get that $d(u, A) \leq d(x, A) + 2 = d(v, A) - d(v, x) + 2 = d(v, A) - 4$ as required. □

We now show that we obtain the same red-blue colouring if we add an edge and if we pre-colour the further endpoint of that edge red.

Lemma 4.15.6. *Let G be a 3-connected planar graph and A a root face of G.*

Suppose $u \in V(G)$ and $v \in V(G)$ are at distance at least 6 for each other in the face-vertex incidence graph of G, labelled so $d(u, A) \leq d(v, A)$, and no 3-cut separates u from v.

Let R be a subset of vertices of G (to be pre-coloured red).

Then the red-blue colouring c of $(G + uv, A, R + v)$ is the same as the red-blue colouring c' of $(G, A, R + v)$.

Proof. Suppose not and c, c' differ. So there is an x such that $c(x)$ is red but $c'(x)$ is blue.

c and c' differ on one of u or v (if u and v are both coloured red by c' then they are in the same red component in G since no 3-cut separates them). Since $v \in R + v$, c and c' differ on u. But again, since no 3-cut separates u from v, u is adjacent to the red component of c' containing v. Let X (which contains u) be the closest detacher to the red component of v in c'. Then X is also a closest detacher in $G + uv$ (since uv does not cross this cut). So u is coloured blue in both c and c', a contradiction. □

Thus, we can add all of F_2 using the following algorithm.

Algorithm 4.15.7.

Input:

- A planar graph P,
- a set of edges F_2 in P,

Output: A red-blue colouring of $P + F_2$.

Description and analysis.

1. Set R to be the empty set.
2. Compute the distance from A to every other vertex in $FV(P)$.
3. For each edge of F_2, add the vertex furthest from A to R (breaking ties arbitrarily).
4. Obtain an red-blue colouring for (P, A, R) using Algorithm 4.13.19 and return it.

□

4.15.3 Adding close non-planar edges (F_3)

We describe Algorithm 4.15.9 which takes a 3-connected planar graph $P = G_t$ and a set of edges F_t''' not present in P such that the endpoints of any edge in F_t''' and finds a pruned K_5-tree of $P + F_t'''$.

In this case, we simply impose the extra condition that at least one endpoint of each edge need to be coloured red (without adding the actual edge), the resulting red-blue colouring is an red-blue colouring of P. We modify Algorithm 4.13.20 to add this extra condition.

Lemma 4.15.8. *Let P planar graph and A a "root" in P.*

If uv is an non-edge in P where $d(u, v) = 2$ (in $FV(G)$) then at least one of u or v is coloured red in a red-blue colouring of $(P + uv, A, \emptyset)$.

Proof. Suppose not. Then both u and v are in the 3-cut X separating the K_5-model created by adding uv from A. But decomposing on X in G yield (two) planar decomponents (all of which contain a clique on X), which is a contradiction to either of them containing a K_5-model. □

Algorithm 4.15.9.

Input:

- A K_5-minor free graph H_r,
- a root set A, and
- a set of edges F not in $E(H_r)$.

Output: A pruned K_5-tree for G_r, the graph obtained from H_r by adding all edges in F to H_r.

Description and analysis. We need to modify Algorithm 4.13.20 to handle these extra constraint. To do so, we leave $\theta_1(X_1), \theta_2(X_1), \theta_3(X_1)$ unchanged and add a new condition $\theta_5(X_1)$ (and change $\theta(X_1)$ to take this new constraint into account).

$\exists E_1(\theta_1(X_1) \wedge \theta_2(X_1) \wedge \theta_3(X_1, E_1) \wedge \theta_4(E_1) \vee \theta_5(X_1))$

$\theta_5(X_1) \quad = \quad \forall uv \in F_t''' \, \mathrm{Red}(u) \vee \mathrm{Red}(v)$

We still set the weight function f_1 to be identically 1 on $V(G)$. Our EMS problem still minimizes $|X_1|_1$ subject to $\theta(X_1)$. □

4.16 Post-recursion: Uncontracting a matching

Let H be a graph obtained by contracting a matching M of a graph G via a mapping f. In this section, we wish to obtain the pruned K_5-tree of G from the pruned K_5-tree of H. We again use local treewidth and the face-vertex incidence graph of the root H_r of the pruned K_5-tree of H. We obtain a red-blue colouring of $f^{-1}(H_r)$ and transform this red-blue colouring and the pruned K_5-tree of H into a pruned K_5-tree of G.

For each leaf graph node t of the pruned K_5-tree of H into a single vertex, build a vertex v_t adjacent to all neighbours of H_t (i.e., $N(v_t) = V(H_s)$ where s is the parent cutting node s of t in the pruned K_5-tree of H). Let P be the graph obtained from H_r by adding all such vertices v_t. Let R be this set of added vertices. Equivalently, P is obtained from the red-blue colouring of H corresponding to pruned K_5-tree of H by contracting all red components to a single vertex. Note that P is K_5-minor free as K_4 is K_5-minor free. In the following, we only uncontract edges of the matching M but not vertices in R. So we extend f to P by setting $f(v_t) = v_t$ for all newly added vertices v_t.

4.16.1 Overview

We build a 20-layering of P in $FV(P)$. Since each layer G_i has bounded treewidth and $f^{-1}(G_i)$ has treewidth at most twice the treewidth of G_i, $f^{-1}(G_i)$ also has bounded treewidth. We can thus start from the highest numbered layer and obtain a red-blue colouring of the uncontraction of that layer and successively uncontract layers while pre-colouring the red vertices coloured when we uncontracted the previous layer.

Instead of explicitly stating the algorithm above, we can also note that, since treewidth at most doubles, $f^{-1}(P)$ has locally bounded treewidth. We could redefine distance between two vertices in $f^{-1}(P)$ be the distance between their images in H_r. Then we can build a 10-layering of $f^{-1}(P)$ in $FV(P)$ (with the distance function we just defined) and find a red-blue colouring as before. All 3-cuts are still within some layer (since the contracting of a 3-cut is within some layer of the 20-layering of H_r).

We now provide the formal details.

4.16.2 Details

We describe the following main algorithm of this section.

Algorithm 4.16.1.

Input:

- A graph G,
- a root A in G,
- a matching N in G such that $H = G/N$ is 3-connected,
- a pruned K_5-tree of H.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Again, we provide the output in the form of a red-blue colouring of G.

Description and analysis.

1. Build the red-blue colouring of H from the pruned K_5-tree of H.
2. Contract every red component of the pruned K_5-tree of H to a single vertex. Call the union of these vertices R and the resulting graph P.
3. Uncontract M in P and call the resulting graph G'.

   As we've discussed above, G' has bounded local treewidth.
4. Run Algorithm 4.13.19 on (G', A, R) to obtain a red-blue colouring c' of G'.

5. Build a red-blue colouring c of G as following. $c(v) = c'(v)v \in V(G') - R$, $c(v) = \text{Red}$ otherwise.
6. Run Algorithm 4.13.21 on c and return its output (as a pruned K_5-tree of G).

This algorithm runs in linear time since every step corresponds to an algorithm which runs in linear time.

Every vertex not in $V(G') - R$ is coloured red since the contraction of any 3-cut is a 3-cut (so uncontracting M cannot create a closer detacher to a K_5-model). Every vertex in $V(G')$ is coloured correctly by □

4.17 Post-recursion: Adding vertices in K_5-model nodes

In this section, we describe an algorithm for obtaining a pruned K_5-tree of G given a pruned K_5-tree for $H = G - R$ where R is a set of vertices we know will appear in K_5-model nodes of a pruned K_5-tree of G.

Algorithm 4.17.1.

Input:

- A graph G,
- a root A in G,
- a set R of $27\varepsilon' n$ vertices in K_5-model nodes of the pruned K_5-tree of G, and
- a graph H obtained from G by deleting these vertices.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis.
1. Build an red-blue colouring c of H.
2. Extend c to a colouring c' of G by colouring all vertices in R red.
3. Run Algorithm 4.13.21 on G, A, c' to obtain a pruned K_5-tree of G and return it.

□

4.18 Post-recursion: Adding small components

In this section, we describe how to add a set of component U_x, each separated by a cut X of size 3 or 4 to H and obtain a pruned K_5-tree of the resulting graph G from a pruned K_5-tree of H.

Algorithm 4.18.1.

Input:

- A graph G,
- a root A in G,
- a set of at least εn laminar 3-cuts or 4-cuts in G and for each cut X, a component U_X of $G - X$ of size at most ε^{-1} disjoint from A
- a 3-connected graph H obtained by removing all components U_X and adding some edges between the vertices of X for each X, and
- a pruned K_5-tree of H.

Output: A pruned K_5-tree of G.

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis.

1. Build the red-blue colouring of H from the pruned K_5-tree of H.
2. Extend this red-blue colouring to an red-blue colouring of G by adding back one component U_X at a time and using a brute force (we can do so since each component has size at most ε^{-1}).
3. Transform the red-blue colouring of G from the previous step into a pruned K_5-tree of G and return it.

□

CHAPTER 5
Applications

In this chapter, we describe how the certifying $(3,3)$-block tree obtained from our main algorithm of Section 3 (Algorithm 3.0.55) can be used to solve optimization problems for K_5-minor free graphs in linear time.

In Section 5.1, we show how this algorithm can be used to solve maximum independent set. In Section 5.2, we use it to solve subgraph isomorphism.

5.1 Maximum independent set

We give a linear time PTAS (a $1+\frac{1}{k}$-approximation algorithm) for the maximum independent set problem restricted to K_5-minor free graphs.

We follow the same steps as Baker's algorithm (here we use the layered construction we described in Section 4.13 based on Eppstein's, but layers based on Baker's construction would also suffice). Our algorithm simply tries to delete every kth level. The resulting components then each have bounded treewidth by Corollary 4.13.17 and we can find the maximum independent set in each graph. There exists some choices of layers whose deletion reduces the size of the maximum stable set in the graph by a factor of at most $\frac{1}{k}$.

Algorithm 5.1.1.

Input: A K_5-minor free graph G.

Output: A stable set S of size at least $(1-1/k)$ the size of a maximum stable set in G.

Running time: $O(k(m+n))$

Description. 1. Build $FV(G)$. Let v be an arbitrary vertex in G.

2. For $i=1,\ldots,k$,

3. (a) Delete levels $i, i+k, i+2k, \ldots$.

(b) Now the graphs G_i induced by levels $i+jk+1, \ldots, i+(j+1)k$ all have bounded treewidth by Lemma 4.13.4.

We find the maximal independent set in each of these graphs using Theorem 2.6.47 and the extended monadic second order logic formulation given in Section 2.6.10. Build the union of these stable sets to form a stable set S_i of G (there are no edges between these stable sets as they are in different components).

4. Return the largest of the S_i's we computed.

Analysis. This algorithm clearly runs in time $O(k(m+n))$ as each step corresponds to an algorithm we have already shown to run in linear time.

Suppose S^* is an optimal solution. Then S^* is the union of the vertices of S^* in levels $i, i+k, i+2k, \ldots$ (for all i).

Thus, there is an i for which deleting vertices of S^* in levels $i, i+k, i+2k, \ldots$ reduces the size of S^* by at most $(1-\frac{1}{k})|S^*|$. The resulting set is a stable set in G_i and by the optimality of S_i, $|S_i| \geq (1-\frac{1}{k})|S^*|$. So the stable set returned by the algorithm is within $1-\frac{1}{k}$ of the optimum, as required. □

5.2 Subgraph isomorphism

In this section, for each graph H, we give a linear time algorithm (with running time depending on H) to determine if an input K_5-minor free graph contains H as a subgraph.

Algorithm 5.2.1. k FACE-VERTEX SUBGRAPH CONTAINMENT

Input: A K_5-minor free graph G and a K_5-minor free graph H where the diameter of $FV(H)$ is at most k.

Output: A subgraph of G isomorphic to H if it exists. "There is no subgraph of G isomorphic to H" otherwise.

As a special case, we can determine if any fixed H is a subgraph of an input graph G. However, as we remark at the end, even though this improves the running time on the best

known result [21] for K_5-minor free graph, it does not actually require building a $(3,3)$-block tree (or our extension of the "face-vertex incidence graph").

Algorithm 5.2.2. H SUBGRAPH CONTAINMENT

Input: A K_5-minor free graph G.

Output: A subgraph of G isomorphic to H if it exists. "There is no subgraph of G isomorphic to H" otherwise.

Basically, we build a $|V(H)|$-layering so that if a copy of H appears in G, this copy is contained in some layer. We then solve the same problem in each layer by exploiting their bounded treewidth.

Description and analysis.

1. Apply Algorithm 4.13.15 to build a k-layering of G (using distances in $FV(G)$).
2. For each layer G_j, determine if H is a subgraph of G_j (using Algorithm 5.2.3).
3. If we find a subgraph isomorphic to H in any layer, return that subgraph (as a subgraph of G).

   Otherwise, return "There is no subgraph of G isomorphic to H".

□

Algorithm 5.2.3. H SUBGRAPH CONTAINMENT

Input: A graph G of bounded treewidth.

Output: A subgraph of G isomorphic to H if it exists. "There is no subgraph of G isomorphic to H" otherwise.

We simply give a LinEMSOL formulation for this problem.

$$\theta = \exists v_1, \ldots, v_{|V(H)|} \forall ij \in E(H) v_i v_j \in E(G)$$

We close this section, with two possible modifications to our algorithm.

Remark 5.2.4. *We can remove the explicit restrictions on G and H and instead allow our algorithm to return a K_5-model in G, K_5-model in H, or "$FV(H)$ has diameter grater than k" as 3 extra outputs. In this case, we can simply preprocess the input accordingly.*

Remark 5.2.5. *For a fixed H, instead of a k-layering, we can build a $|V(H)|$-layering using distances in G (instead of $FV(G)$). In particular, we do not need to first build any $(3,3)$-block tree in this case (by simply running BFS from any vertex in G). In this case, the entire algorithm (i.e., Algorithm 5.2.2) still runs in linear time.*

In fact, a more general statement can be made that is originally due to Baker [5] and extended by Eppstein.

Theorem 5.2.6. *[5, 26] There is a linear time PTAS (a $1+O(1/k)$-approximation algorithm) for the following problems in any minor closed family of graphs with locally bounded treewidth.*

- *maximum independent set,*
- *minimum vertex cover,*
- *maximum H-matching, and*
- *minimum dominating set.*

See [5] for a complete list of problems for which such a PTAS exists. Combined with Eppstein's characterization of graph of locally bounded treewidth, we see these problems have linear time PTAS on K_5-minor free graphs.

Theorem 5.2.7. *A family of minor closed graphs have locally bounded treewidth if and only if there exists an apex graph H such that H is not a minor of any graph in the family.*

We recall the definition of an apex graph.

Definition 5.2.8. A graph G is an *apex graph* if there exists a vertex v such that $G - v$ is planar.

(K_5 is an apex graph since K_4 is planar.) In the next section, we see to how improve on the running time of these algorithm. We obviously do not improve their dependence on the size of the graph n (as this is best possible) but instead reduce its dependence on k.

5.3 Improving linear time PTASs

In this section, we improve on Theorem 5.2.6. Eppstein used Bodlaender's [8] algorithm as a subroutine to construct the tree decomposition of each layer. The hidden constants in running time of Bodlaender's algorithm depend heavily (it is super-exponential) on the treewidth of the graph (and thus in this case, the diameter of the graph). We follow [21] and reduce this dependence by instead using Baker's algorithm for constructing the tree decompositions for planar graphs of fixed diameter.

Theorem 5.3.1. *[5] We can construct a tree decomposition of a planar graph of diameter d of width at most $3d$ in time $O(dn)$.*

We can easily extend this algorithm from planar graphs to K_5-minor free graphs by building a $(3,3)$-block tree for the K_5-minor graph and combining the tree decomposition for its graph nodes (obtained from Theorem 5.3.1) on cuts of $(3,3)$-cuts.

Thus, we can build all tree decompositions needed in all steps in $O(kn)$. The dominant term in the running time of our algorithm is now due to the algorithm for solving the problem in question (e.g., dominating set, maximum independent set, etc) on a graph of bounded treewidth (which is exponential in k). Demaine et al. have shown this term (and therefore our total running time) to be $O(k \cdot 2^{2k}n)$ for all problems listed in Theorem 5.2.6.

5.4 Bounded intersection tree decomposition

We now present further applications of the $(3,3)$-block tree. We first present a slightly more general $\mathcal{F}$-block tree that we use instead.

In both the last two chapters, we started by reduce the problem in question to a key problem in a 3-connected graph and solved this key problem. The certificate to the problem in Chapter 3 can be extended to any graph by first building a block tree, then building a 2-block tree for each graph node of the block tree and finally building $(3,3)$-block tree for each graph node of the 2-block tree.

In this section, we describe and use the *bounded intersection tree decomposition* which combines all cuts of interest to us into a single tree decomposition. We simply let $\mathcal{F}$ be all cuts listed above, all 1-cuts of G, a laminar family of 2-cuts in each block and all $(3,3)$-cut in each 2-connected component.

Definition 5.4.1. A *bounded intersection tree decomposition* of a connected graph G is an $\mathcal{F}$-block tree where $\mathcal{F}$ consists of 1-cuts, 2-cuts and $(3,3)$-cuts and every graph node is either planar or L.

Thus, by Wagner's theorem implies every connected K_5-minor free graph has a bounded intersection tree decomposition. We use the following algorithm to build a bounded intersection tree decomposition of a K_5-minor free graph G.

Algorithm 5.4.2.

Input: A connected graph G.

Output: Either a bounded intersection tree decomposition of G or a K_5-minor in G.

Note this algorithm can also take a general graph G as input and output.

Description and analysis.

- Build the block tree $[T, \mathcal{G}]$ of G.
- For each graph node (block) G_t,
  - build the 2-block tree $[T', \mathcal{G}']$ of G_t.
  - For each graph node G'_t of the 2-block tree of G_t
    - Run Algorithm 3.0.55 on G'_t
    - If a K_5-model is returned, return the corresponding K_5-model in G.
    - Otherwise, we obtain a $(3,3)$-block tree of G'_t.
  - Take the disjoint union of all $(3,3)$-block tree obtained (for all graph nodes of the 2-block tree of G'_t) and all cutting nodes of G_t in its 2-block tree.
  - For each cutting node G'_s of the 2-block tree of G_t, for each neighbour u of s in $[T', \mathcal{G}']$, add an edge from G'_s an arbitrary node of the $(3,3)$-block tree of G'_u containing all of G'_s.

The resulting decomposition is a bounded intersection tree decomposition of G_t.

- Take the disjoint union of all bounded intersection tree decompositions obtained (for all graph nodes (blocks) of the block tree of G) and all cutting nodes of G in its block tree.
- For each cutting node G'_s of the 2-block tree of G_t, for each neighbour u of s in $[T', \mathcal{G}']$, add an edge from G'_s an arbitrary node of the $(3,3)$-block tree of G'_u containing all of G'_s.

  The resulting decomposition is a bounded intersection tree decomposition of G, which we return.

Since the total size of all block trees, all 2-block trees and all $(3,3)$-block trees is linear in the size of G and we can build them in linear time, the entire algorithm runs in linear time.

To add all new edges from all cutting nodes to an arbitrary graph node containing all vertices of the cutting node, we simply build a lookup table for all vertices listing the nodes containing them and look up the corresponding entry for each cutting node. □

5.5 k-Realizations

We now use the bounded intersection tree decomposition to solve the k-Realizations problem. The k-Realizations problem (k fixed) has the following form.

Problem 5.5.1 (k-realizations)**.**

Input: *A graph G and a set X of k vertices of G.*

Output: *All partitions $\Delta = (D_1, \ldots, D_\ell)$ of X such that there is a set of disjoint trees $\{T_1, \ldots, T_\ell\}$ with $D_i \subseteq V(T_i)$ (we return this information as a look up table with entries indexed by partitions Δ).*

5.5.1 Previous work and connection to graph minors

This problem can be solved in $O(n^3)$ time in general using Robertson and Seymour's seminal results. In [55], Reed et al. showed that this problem can be solved in linear time

on planar graphs. Our decomposition allows us to extend this result to graphs with no K_5-minor.

Theorem 5.5.2. *[55] For any fixed k, k-Realizations can be solved in linear time if the input graph is planar.*

This problem appears in the theory of graph minors as a polynomial time algorithm for solving k-Realizations can be used to determine if G contains a fixed graph H as a minor in polynomial time. We follows Reed's [53] treatment of this reduction.

Problem 5.5.3. *[k-DRP]*

Input: *A graph G and $2k$ vertices $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$.*

Output: *"Yes" if there exist k vertex disjoint paths, one from s_i to t_i for each i, and "no" otherwise.*

Lemma 5.5.4. *If we can solve $2k$-Realizations then we can solve k-DRP in the same running time.*

Proof. Given an instance $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ of k-DRP, solve an instance $G, \{s_1, \ldots, s_k, t_1, \ldots,$ of $2k$-Realizations and return whether the partition $((s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k))$ is realizable. □

Remark 5.5.5. *An algorithm for k-DRP implies an algorithm for the more general problem where we allow the paths to share endpoints but insist that they are otherwise disjoint. We simply make multiple copies of any vertex which appears more than once in $s_1, \ldots, s_k, t_1, \ldots, t_k$.*

Lemma 5.5.6. *If we can solve k-DRP in polynomial time then we can solve H-subdivision containment in polynomial time where $|E(H)| = k$*

Proof. Given an instance G of H-subdivision containment, we try all $O(n^{|V(H)|})$ possible choices for centers of H in G and ask for paths corresponding to edges of H. □

Now combined with Theorem 2.6.28, we see that a polynomial time algorithm for k-Realizations for all fixed k implies a polynomial time algorithm for H-minor containment for all fixed H (by testing for all subdivisions in $Z(H)$, the set of graphs such that if H is a minor of a graph G then G contains some graph in $Z(H)$ as a subdivision).

k-Realizations can also be use to solve other similar problems that we now list.

Problem 5.5.7. *[k-rooted routing]*

Input: *A graph G and a set X of $2k$ vertices.*

Output: *All labellings of X into $s_1, \ldots, s_k, t_1, \ldots, t_k$ for which $G, (s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ is a "Yes" instance of k-DRP.*

Problem 5.5.8. *[k-vertex disjoint trees]*

Input: *A graph G, a set X of k vertices in G and a partition $\Delta = (D_1, \ldots, D_\ell)$ of X.*

Output: *"Yes" if there exist ℓ vertex disjoint trees $T_1, \ldots, T_\ell$ with $D_i \subseteq T_i$ for each i, and "no" otherwise.*

Lemma 5.5.9. *If we can solve $2k$-Realizations then we can solve k-rooted routing and k-vertex disjoint paths in the same running time.*

Again, we only need to look at the appropriate part of the solution.

For planar graphs, Reed et al. [55] gave a linear time algorithm which we use as a subroutine (their paper refers to k-Realizations as the *disjoint trees problem*).

Theorem 5.5.10. *[55] For fixed k, there is a $O(|V(G)|)$ algorithm to solve k-Realizations when the input graph G is planar.*

If k is not fixed, this problem is NP-hard even G is planar. In fact, even k-DRP is NP-hard when G is planar.

5.5.2 Solving k-realizations on K_5-minor free graphs

To solve this problem, we construct the bounded intersection tree decomposition $[T, \mathcal{S}]$ for G.

If there is a leaf $t \in T$ which contains none of X, we can simply contract all vertices in G_t but not its parent s (cutting node) to a vertex without changing the answer to the problem. In fact, since any cutting node has size at most 3, we can remove this leaf vertex and if it is the only child of s, replace G_s by a clique without changing the answer to the problem.

In fact, we can build the minimal subtree T' of T whose leaves contain only vertices of X and root T at an arbitrary node r of T'. We then delete all vertices in any component C of $T - V(T')$ that do not appear elsewhere in the tree, and add a clique on the neighbourhood of each deleted component.

By our definition of a bounded intersection tree decomposition, the graph nodes of the resulting tree $[T', \mathcal{S}']$ are still planar or L.

Thus, $[T', \mathcal{S}']$ is the bounded intersection tree decomposition of a graph G' with the same realizable partitions as G for the set X.

Since T' has at most k leaves, nodes in T' have degree at most $3k$. Thus, for each graph node t of T', we can solve an instance of q-realizations with $q \leq 4k$ where the terminals consist of all vertices of X in G_t and all vertices in a cutting node incident to G_t.

We then replace each G_t by H_t, the smallest graph with the same realizable partitions as G_t (with the same set of terminals). The size of H_t is bounded by a function of $4k$ (independent of the size of G_t) so we now have a tree decomposition $\mathcal{H}$ of some graph H of bounded tree width.

Furthermore, by our choice of H_t, the same partitions are realizable in G and H. Indeed, given a partition $\Delta = (D_1, \ldots, D_\ell)$ of X realized by $T_1, \ldots, T_\ell$ in G, we replace the part of $T_1, \ldots, T_\ell$ in G_t for each $t \in T'$ by a forest connecting the same endpoints (in X and vertices in the intersection of G_t and its neighbours). This gives us a realization of Δ in H. Similarly, given a $\Delta = (D_1, \ldots, D_\ell)$ of X realized by $T_1, \ldots, T_\ell$ in H, we replace the part of $T_1, \ldots, T_\ell$

in G_t for each $t \in T'$ by a forest connecting the same endpoints (in X and vertices in the intersection of H_t and its neighbours). This gives us a realization of Δ in G.

The above remark allows us to obtain a solution for H with terminals X using the tree decomposition $[T', \mathcal{H}]$ and return it as a solution for G.

Algorithm 5.5.11.

Input: A K_5-minor free graph G and a set X of k vertices of G.

Output: All partitions $\Delta = (D_1, \ldots, D_\ell)$ of X such that there is a set of disjoint trees $\{T_1, \ldots, T_\ell\}$ with $D_i \subseteq V(T_i)$ (we return this information as a look up table with entries indexed by partitions Δ).

Running time: $O(|V(G)| + |E(G)|)$.

Description and analysis.

1. Build a bounded intersection tree decomposition $[T, \mathcal{G}]$ of G.
2. Let Y be the set of all graph nodes containing a vertex of X.
3. Build T', the smallest subtree of T containing all of Y.

   This can be done by rooting T at an arbitrary root $r \in Y$ and using dynamic programming to prune the tree.
4. For any leaf t of T' that is not a leaf of T, add a clique in G_s corresponding to each child (cutting) node s of t.

   Also add all these cliques to G to form a graph G'. Call the resulting decomposition $[T', \mathcal{G}']$.
5. For each graph node t in T',
   (a) let S_t be the union of all vertices in cutting nodes incident to t.
   (b) Run the algorithm of Reed et al. [55] on $G'_t, S_t \cup (X \cap G'_t)$ to obtain all realizable instances.
   (c) Replace G'_t by H_t, the smallest graph with the same realizable partitions as G'_t (we can pre-compute all H_t in advance for all subsets of realizable instances on at most $4k$ vertices).

6. Now $[T', \mathcal{H}]$ is a tree decomposition of bounded width.

   Use dynamic programming or an MSO formulation to solve k-realizations using $[T', \mathcal{H}]$.

   Return the realizable instances as the realizable instances of G.

□

CHAPTER 6
Concluding remarks

In this thesis, we gave a linear time algorithm for building two tree decompositions, the $(3,3)$-block tree of a 3-connected K_5-minor free graph and the pruned K_5-tree of a 3-connected graph with no attached K_5-model. They are used to solve two problems arising from theory of graph minors, K_5-minor recognition and 2-disjoint rooted paths.

We then use these algorithm to speed up existing algorithms and extend their inputs from planar graphs to K_5-minor free graphs. One direction for future work is to explore further applications of these tree decompositions, as discussed in Chapter 5.

Another question which remains open is whether our linear time algorithm for solving 2-DRP can be modified into one which returns the desired paths when they exist. One possibility is to modify Algorithm 4.4.4 for restricted attached K_5-model to return an attached K_5-model when it exists. However, currently, in many of the reductions we make, we are unable to obtain an attached K_5-model for the input given an attached K_5-model obtained recursively.

Similarly, we can think of modifying our algorithm to linear time recognition algorithms such as graphs excluding a graph H which can be drawn in the plane with a single pair of edges crossing, which also admit a decomposition similar to Wagner's theorem for K_5-minor free graphs.

Finally it remains open is whether we can also build the $(3,3)$-block tree of any 3-connected graph. As we have shown such a tree decomposition is unique for any 3-connected graph, except for $K_{3,3}$ (which has two decompositions). Although we can build such a tree decomposition recursively in polynomial time, is it also possible in linear time?

REFERENCES

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

[2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

[3] S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23:11–24, 1989.

[4] T. Asano. An approach to the subgraph homeomorphism problem. *Theoretical Computer Science*, 38:249–267, 1985.

[5] B.S. Baker. Approximation algorithms for np-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994.

[6] H. L. Bodlaender. Dynamic programming algorithms on graphs of bounded tree-width. In T. Lepisto and A. Salomaa, editors, *15th International Colloquium on Automata, Languages and Programming*, volume 317, pages 105–118. Springer Verlag, 1988.

[7] H. L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11:631–643, 1990.

[8] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[9] H.L. Bodlaender, T. Kloks, and D. Kratsch. Treewidth and pathwidth of permutation graphs. *SIAM Journal on Discrete Mathematics*, 8:606, 1995.

[10] H.L. Bodlaender and D.M. Thilikos. Treewidth for graphs with small chordality. *Discrete Applied Mathematics*, 79(1):45–61, 1997.

[11] H.L. Bodlaender, J. Van Leeuwen, R. Tan, and D.M. Thilikos. On interval routing schemes and treewidth. *Information and Computation*, 139(1):92–109, 1997.

[12] J. Boyer and W. Myrvold. Stop minding your p's and q's: A simplified o (n) planar embedding algorithm. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 140–146. Society for Industrial and Applied Mathematics, 1999.

[13] H.J. Broersma, E. Dahlhaus, and T. Kloks. A linear time algorithm for minimum fill-in and treewidth for distance hereditary graphs. *Discrete Applied Mathematics*, 99(1):367–400, 2000.

[14] J. Bruno, K. Steiglitz, and L. Weinberg. A new planarity test based on 3-connectivity. *Circuit Theory, IEEE Transactions on*, 17(2):197–206, 1970.

[15] B. Courcelle. The monadic second-order logic of graphs, ii: Infinite graphs of bounded width. *Theory of Computing Systems*, 21(1):187–221, 1988.

[16] B. Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

[17] B. Courcelle. The monadic second-order logic of graphs iii: tree-decompositions, minor and complexity issues. *ITA*, 26:257–286, 1992.

[18] B. Courcelle, J. A. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory of Computing Systems*, 33:125–150, 2000.

[19] H. de Fraysseix and P.O. de Mendez. Pigale-public implementation of a graph algorithm library and editor. *SourceForge project page http://sourceforge. net/projects/pigale*, 2002.

[20] H. De Fraysseix and P. Rosenstiehl. A depth-first-search characterization of planarity. *Ann. Discrete Math*, 13:75–80, 1982.

[21] E. Demaine, M. Hajiaghayi, N. Nishimura, P. Ragde, and D. Thilikos. Approximation algorithms for classes of graphs excluding single-crossing graphs as minors. *Journal of Computer and System Sciences*, 69(2):166–195, 2004.

[22] E. D. Demaine, M. Hajiaghayi, and K. Kawarabayashi. Algorithmic graph minor theory: Decomposition, approximation and coloring. In *Proc. 46th Ann. IEEE Symp. Found. Comp. Sci.*, pages 637–646, 2005.

[23] G. Di Battista and R. Tamassia. On-line graph algorithms with spqr-trees. *Automata, Languages and Programming*, pages 598–611, 1990.

[24] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.

[25] GA Dirac. On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universit*
"at Hamburg, volume 25, pages 71–76. Springer, 1961.

[26] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 632–640. Society for Industrial and Applied Mathematics, 1995.

[27] D. Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3):275–291, 2000.

[28] A. Frank, T. Ibaraki, and H. Nagamochi. On sparse subgraphs preserving connectivity properties. *J. Graph Theory*, 17(3):275–281, 1993.

[29] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

[30] F. Gavril. Algorithms on clique separable graphs. *Discrete Mathematics*, 19(2):159–165, 1977.

[31] G. Gottlob, R. Pichler, and F. Wei. Monadic datalog over finite structures of bounded treewidth. *ACM Transactions on Computational Logic (TOCL)*, 12(1):3, 2010.

[32] M. Grohe. Logic, graphs, and algorithms. *Logic and Automata–History and Perspectives*, pages 357–422, 2007.

[33] J. Gustedt. Efficient union-find for planar graphs and other sparse graph classes* 1. *Theoretical computer science*, 203(1):123–141, 1998.

[34] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In Joe Marks, editor, *Graph Drawing, Colonial Williamsburg, 2000*, pages 77–90. Springer, 2001.

[35] B. Haeupler and R.E. Tarjan. Planarity algorithms via pq-trees. *Electronic Notes in Discrete Mathematics*, 31:143–149, 2008.

[36] M.T. Hajiaghayi. Algorithms for graphs of (locally) bounded treewidth. Master's thesis, University of Waterloo, 2001.

[37] R. Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, 1976.

[38] F. Havet, G. Quercini, and B. Reed. Personal communication, 2007.

[39] J. Hopcroft and R.E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 3:135–158, 1973.

[40] J. Hopcroft and R.E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[41] John E. Hopcroft and Robert E Tarjan. Efficient algorithms for graph manipulation. Technical report, Stanford University, Department of Computer Science, Stanford, CA, USA, 1971.

[42] HA Jung. Eine Verallgemeinerung des n-fachen Zusammenhangs f
"ur Graphen. *Mathematische Annalen*, 187(2):95–103, 1970.

[43] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proceedings of the sixteenth annual ACM symposium on Theory of com puting*, STOC '84, pages 266–272, New York, NY, USA, 1984. ACM. For maximal stable set (need to convert using the line graph, mentioned on page 1). Sequential algorithm stated on page 2.

[44] A. Kézdy and P. McGuinness. Sequential and parallel algorithms to find a K_5 minor. In *Third Annual Symposium on Discrete Algorithms*, pages 345–356. Springer, 1992.

[45] T. Kloks. Treewidth of circle graphs. *Algorithms and Computation*, pages 108–117, 1993.

[46] S. Kreutzer. Algorithmic meta-theorems. *Parameterized and Exact Computation*, pages 10–12, 2008.

[47] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematica*, 16:271–283, 1930.

[48] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs. International Symposium, Rome, Italy*, pages 215–232, 1966.

[49] L. Lovász. Graph minor theory. *American Mathematical Society*, 43(1):75–86, 2006.

[50] S. Mac Lane. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460–472, 1937.

[51] K. Paton. An algorithm for the blocks and cutnodes of a graph. *Communications of the ACM*, 14(7):468–475, 1971.

[52] L. Perkovic and B. Reed. An improved algorithm for finding tree decompositions of small width. *International Journal of Foundations of Computer Science*, 11(3):365–371, 2000.

[53] B. Reed. *Rooted Routing via Graph Minors.* Manuscript.

[54] B. Reed. Tree width and tangles: A new connectivity measure and some applications. *Surveys in combinatorics*, 241:87–162, 1997.

[55] B. Reed, N. Robertson, L. Schrijver, and P. Seymour. Finding disjoint trees in planar graphs in linear time. In *Graph Structure Theory*, pages 295–302. AMS, 1993.

[56] B. Reed and D. Wood. Fast separation in a graph with an excluded minor. In *Euro-Conference on Combinatorics, Graph Theory and Applications*, pages 45–50, 2005.

[57] B.A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 221–228. ACM, 1992.

[58] N. Robertson and P. D. Seymour. Graph minors. XIII: the disjoint paths problem. *J. Comb. Theory Ser. B*, 63(1):65–110, 1995.

[59] N. Robertson and P. D. Seymour. Graph minors. XVI. excluding a non-planar graph. *Journal of Combinatorial Theory, Series B*, 89:43–76, 2003.

[60] N. Robertson and P.D. Seymour. Graph minors. v. excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, 1986.

[61] J. Lagergren S. Arnborg and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[62] P.D. Seymour. Disjoint paths in graphs. *Discrete Mathematics*, 29:293–309, 1980.

[63] W.K. Shih and W.L. Hsu. A new planarity test. *Theoretical Computer Science*, 223(1-2):179–191, 1999.

[64] Y. Shiloach. A polynomial solution to the undirected two paths problem. *J. ACM*, 27(3):445–456, 1980.

[65] R. Sundaram, K.S. Singh, and C. Pandu Rangan. Treewidth of circular-arc graphs. *SIAM Journal on Discrete Mathematics*, 7(4):647–655, 1994.

[66] R. Tamassia. *Handbook of graph drawing and visualization*. Chapman & Hall/CRC, 2007.

[67] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[68] R.E. Tarjan. Decomposition by clique separators. *Discrete mathematics*, 55(2):221–232, 1985.

[69] T. Tholey. Solving the 2-disjoint paths problem in nearly linear time. In *STACS 2004*, volume 39, pages 51–78, 2004.

[70] T. Tholey. Improved algorithms for the 2-vertex disjoint paths problem. *SOFSEM 2009: Theory and Practice of Computer Science*, pages 546–557, 2009.

[71] C. Thomassen. 2-linked graph. *European Journal of Combinatorics*, 1:371–378, 1980.

[72] WT Tutte. A theory of 3-connected graphs. *Indag. Math*, 23:441–455, 1961.

[73] K. Wagner. Über eine Eigenschaft der ebenen Komplexe. *Math. Ann.*, 114:570–590, 1937.

[74] K. Wagner. Über eine Erweiterung eines Satzes von Kuratowski. *Deutsche Mathematik*, 2:280–285, 1937.

[75] S.H. Whitesides. Algorithm for finding clique cut-sets. *Info. Proc. Lett.*, 12(1):31–32, 1981.

[76] H. Whitney. Non-separable and planar graphs*. *Classic Papers in Combinatorics*, pages 25–48, 1987.

[77] S. G. Williamson. Depth-first search and Kuratowski subgraphs. *J. ACM*, 31(4):681–693, 1984.

[78] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 681–690. ACM, 2006.