Utrecht University

Department of Information
and Computing Sciences

Disjoint Paths and Directed Steiner Tree on Planar Graphs with Terminals on Few Faces

Master's Thesis in Computing Science

Hilde Verbeek

Supervisors:

dr. Jesper Nederlof

prof. dr. Hans L. Bodlaender

Co-supervisors:

Sukanya Pandey

Krisztina Szilagyi

Defended on 27 July 2022

Abstract

In the field of parameterized complexity, the main topic of interest is to make normally hard problems more tractable. A problem is in the complexity class FPT (fixed-parameter tractable) if it permits an algorithm with a running time bounded by $f(k) \cdot \text{poly}(n)$, where n is the conventional input size and k is some defined parameter of the input. The consequence of a problem being in FPT, is that it can be solved in polynomial time if the parameter is fixed, despite the problem being NP-hard generally. Many different parameters can be considered for fixed-parameter tractability: usually, this is a natural parameter arising from the problem formulation, such as the weight of a Steiner tree or the size of a dominating set, in their respective problems. Other parameters may be, for example, the output size, structural properties of an input graph (eg. treewidth).

A particularly interesting parameter is the *face cover number*, on problems whose input consists of a planar graph G and a set of terminal vertices $K \subseteq V(G)$. The face cover number of such an input is then the minimum number of faces needed in a planar embedding of G, such that every terminal in K is incident on at least one of those faces. While FPT graph problems already tend to perform better on planar graphs, this parameterization can create a further improvement for certain instances, where many terminals are incident on few faces.

Two problems are discussed in this thesis. Firstly, the Disjoint Paths problem is considered, which asks for a set of pairwise vertex-disjoint paths between given pairs of terminals in a graph. While it is a classically (very) hard problem, it has been shown to be fixed-parameter tractable. Different approaches to solving the problem are discussed, ending with a parameterization by the face cover number. The second considered problem is that of Directed Steiner Tree, which asks for a directed subtree of minimum weight in a graph, that connects all given terminals. The classic Dreyfus-Wagner algorithm for Steiner Tree is discussed, and it is shown that it can be modified for both the directed variant and one parameterized by the face cover number (and both). Next, a $2^{O(k)} \cdot n^{O(\sqrt{k})}$ algorithm by Kisfaludi-Bak et al. is explained, parameterized by the face cover number, and adapted to directed graphs.

Contents

Chapter 1

Introduction

A central objective in the study of complexity is to find optimal running times for algorithmic problems. It is commonly believed that problems in the class NP-hard cannot be solved in polynomial time, which creates a significant barrier in the design of practically applicable algorithms. A common workaround for this is to settle for suboptimal solutions, through the use of polynomial-time approximation schemes and other optimization methods.

The field of parameterized complexity attempts to break this barrier while still finding exact solutions to problems. A problem is said to be *fixed-parameter tractable* (FPT) with regard to some input parameter k, if it permits a running time that is polynomial when the value of k is fixed. In certain applications, this may actually yield useful exact algorithms when prerequisite knowledge tells us this parameter is upper-bounded by a constant.

Several different parameters may be considered for parameterized algorithms, depending on the domain. Sometimes, there are obvious candidates for parameters that arise from the problem description. In other cases, algorithms can be parameterized by the output size or less obvious parameters of problem, or structural properties of an input graph such as its *treewidth*, a measure of a graph's similarity to a tree.

In designing algorithms for graph problems, the structure of the input graph is an important factor to consider. As a general rule, it can be said that graph problems are easier to solve on planar graphs than they are in general. This difference is exhibited by certain structural properties of planar graphs, such as the fact that they are sparse and that their treewidth is bounded by the square root of the number of vertices (this bound is linear on general graphs) [11]. Other advantages of planar graphs also exist: for example, the topology of a planar embedding can give clues to designing good solutions in ways that are impossible on general graphs.

Certain graph problems contain in their input a special set of vertices called *terminals*, which need to be represented in the output in some way. It is natural to look for algorithms that are parameterized by the number of terminals. This allows for algorithms that run in polynomial time when the number of terminals is bounded by a constant, but the disadvantage is that the choice of this constant does have an exponential influence on the running time. This means that it may still be impractical to use the algorithm for inputs with many terminals, even if there exists a constant upper bound on them. A related parameter, specifically on planar graphs, is the *face cover number* of the terminals, which is the minimum number of faces needed in any planar embedding of the graph, such that all terminals are on the border of at least one of those faces.

The face cover number has some advantages as a parameter compared to the number of terminals. For one, using the face cover number means there is no longer a running time dependence on the number of terminals itself, meaning the problem has the same asymptotic running time regardless of the number of terminals, as long as those terminals are incident on the same number of faces. Secondly, having an ordering of terminals along a face's border can aid in algorithm design. The third advantage is from a practical standpoint: in certain applications,

it is natural to have groups of terminals clustered together in the plane.

This thesis discusses two well-known graph problems, their generic parameterized algorithms and parameterization by the face cover number of the terminals.

The Disjoint Paths problem is a very difficult problem that is closely related to the theory of graph minors. Given a graph and a set of pairs of terminals, the problem asks to find pairwise vertex-disjoint paths between all terminal pairs. Its practical interest is mainly in networking and VLSI (very large scaled integrated) chip design: here, it is asked to connect terminals on a flat chip using non-intersecting wires [16, 45]. Most interest in the problem, however, is purely theoretical: in their series of publications about graph minors, Robertson and Seymour tied Disjoint Paths to the problem of testing whether or not one graph is a minor of the other. An FPT algorithm for the latter problem could prove useful in the classification of graphs, as their work showed, making it a natural objective to find an FPT algorithm for Disjoint Paths itself.

Disjoint Paths is NP-complete, but Robertson and Seymour did manage to find an FPT algorithm for the problem, parameterized by the number of terminals [39]. This algorithm can be seen as a major breakthrough, but unfortunately the parameter dependence of its running time is so large that it is in no way practical for real-world use. Nevertheless, the algorithm has inspired much subsequent work in the field. Notably, algorithms for this problem on planar graphs perform significantly better than those designed for general graphs; parameterization by the face cover number, though, has received little attention over time.

The second problem discussed in this thesis is Steiner Tree. This is a very widely studied problem that asks, for a weighted graph and a set of terminals, a minimum-weight subtree of the graph that connects all terminals. The problem has applications in circuit and network design, with several variants for the problem existing for specific domains. For example, in the Euclidean Steiner Tree problem, the terminals are embedded in a plane and need to be connected by a set of segments of minimum total length, without a pre-existing graph structure [10]. Rectilinear Steiner Tree is similar, with the added caveat that all segments need to be horizontal or vertical [18]. A variant that receives attention in this thesis, is Directed Steiner Tree, in which the input graph is directed and contains a root vertex, and the output tree needs to be rooted in said vertex.

Steiner Tree was one of Karp's original NP-complete problems [20]. Parameterized algorithms for the problem are widely researched, with one of the most famous algorithms for it, by Dreyfus and Wagner, being FPT parameterized by the number of terminals [13]. Conveniently, this algorithm can easily be adapted to use the face cover number [5], as well as to the directed variant.

Structure of the thesis. In Chapter 2, some preliminary definitions and theorems are given that are used throughout the thesis. Disjoint Paths and Steiner Tree are discussed in Chapters 3 and 4 respectively. Both chapters start with a general background of the problem and discuss common algorithms, before going over their parameterizations by the face cover number. Some algorithms are discussed in detail, whereas for others only a broad summary of their techniques is given. The chapter on Steiner Tree adapts its two discussed algorithms to directed graphs. The final chapter revisits both problems, summarizing my findings and discussing some open questions.

Chapter 2

Preliminaries

The problems discussed in this thesis work with both directed and undirected graphs. For a graph G, the set $V(G)$ denotes its vertices; if it is undirected, its edges are denoted by $E(G)$ and if it is directed, its arcs are denoted by $A(G)$. In problems with a graph G as input, it can be assumed that $n = |V(G)|$ and $m = |E(G)|$ or $|A(G)|$ unless stated otherwise.

For convenience, I sometimes abuse notation for certain operations on graphs. For example, $G \cup H$ with G and H being graphs, denotes the union of the two graphs, ie. a graph G' with $V(G') = V(G) \cup V(H)$ and (assuming the undirected case) $E(G') = E(G) \cup E(H)$. Similarly, $G - v$ denotes the graph G with vertex $v \in V(G)$ removed from it; the same can be done to edges or arcs, sets of vertices, edges or arcs or entire subgraphs. In the case when a vertex is removed from a graph, its incident edges or arcs are removed too.

As a general rule, these abuses of notations are only applied when their meaning is immediately clear from the context.

2.1 Fixed-parameter tractability

Many problems contain in their input, besides the regular input size, an extra parameter that is usually denoted by the letter k. Such a problem is called a *parameterized problem* if it is stated with the value of k fixed. Generally, many possibilities for parameters are available for a problem. Some examples include the output size or the number of terminals in the input. Less obvious parameters are, for example, structural properties of the input graph such as its treewidth (see Section 2.3) or the face cover number of the terminals (see Section 2.2).

The aim of parameterized complexity, is to find polynomial-time solutions for parameterized versions of problems when the problems are normally NP-hard and thus do probably not allow a polynomial-time solution. That way, if it is somehow known beforehand that the value of the parameter is fixed, it is possible to actually solve the problem in polynomial time.

A parameterized problem with parameter k and input size n belongs to the class **FPT** (*fixed-parameter tractable*) if there exists an algorithm for it with a running time of the form $f(k) \cdot \text{poly}(n)$, meaning that the its (polynomial) dependence on n is itself unaffected by the value of k. The class **XP** (which stands for *slicewise polynomial*), on the other hand, contains problems with a running time of the form $n^{f(k)}$. While XP problems do run in polynomial time for fixed k, the running time does grow faster for higher k.

2.2 Face cover number in planar graphs

Both problems discussed in this thesis have a set of terminal vertices $K \subseteq V(G)$ in their input. A natural parameter to consider then is $k = |K|$, and both problems do permit an FPT algorithm parameterized by this. Another parameter that may be considered, when G is planar, is the *face cover number*, defined as follows:

Definition 2.1 (Face cover number)**.** *Let G be a planar graph and $K \subseteq V(G)$ a set of terminal vertices. The* face cover number *$\gamma(G, K)$ of K in G is defined as the lowest integer p, such that there exists a planar embedding of G that contains faces $F_1, ..., F_p$ in which every vertex of K is incident on at least one face of $F_1, ..., F_p$.*

The face cover number can be a powerful parameter, because an algorithm parameterized by it could run in the same asymptotic time for any amount of terminals, provided that those terminals are incident on a fixed number of faces. Moreover, it may be useful for real-world applications which operate on a flat surface with many terminals clustered together.

Determining the terminal face cover number of a graph is NP-complete, but there exists an algorithm that for given k, either determines that a graph permits no face cover of at most k faces, or outputs a set of at most k faces in a planar embedding covering all terminals. This algorithm runs in $c^k \cdot n$ time for some constant c, meaning that it is in FPT [6]. Thus, for FPT problems parameterized by the face cover number, it can be assumed that this algorithm is used to determine a face cover beforehand and the algorithms as a whole is still FPT.

2.3 Tree decompositions and treewidth

A powerful tool in the design of graph algorithms is a *tree decomposition*, which is a tree-like representation of the graph's structure. Tree decompositions allow one to develop dynamic programming algorithms, that consider only small parts of the graph at a time; the more "tree-like" a graph is, the lower the running time of these algorithms. Formally, a tree decomposition is defined as follows:

Definition 2.2 (Tree decomposition)**.** *A* tree decomposition *of an undirected graph G is a tree T and vertex sets (called* bags*) $\{X_t\}_{t \in V(T)}$ that satisfies the following properties:*

1. *for every vertex $v \in V(G)$, there exists some node $t \in V(T)$ with $v \in X_t$;*
2. *for every vertex $v \in V(G)$, the nodes $\{t \in V(T) : v \in X_t\}$ form one connected component in T;*
3. *for every edge $\{u, v\} \in E(G)$, there exists a node $t \in V(T)$ such that $\{u, v\} \subseteq X_t$.*

The width *of T is defined as*

$$tw(T) = \max_{t \in V(T)} |X_t| - 1.$$

Furthermore, the *treewidth* of a graph is defined as follows.

Definition 2.3 (Treewidth)**.** *The* treewidth *$tw(G)$ of a graph G is the minimum width over all tree decompositions of G.*

In parameterized problems, I denote the treewidth of a graph by w when it is used as a parameter.

It can be said, in a sense, that the lower a graph's treewidth is, the more the graph looks like a tree. Trees and forests have treewidth 1, and outerplanar graphs have treewidth 2 [11]. Conversely, a complete graph of n vertices has treewidth $n - 1$, as it is complex and not at all like a tree. Planar graphs have treewidth at most $O(\sqrt{n})$.

2.3.1 Computing tree decompositions

In FPT problems parameterized by the input graph's treewidth, it is generally assumed that a tree decomposition is supplied with the input; the running time is then specifically dependent on the width of said tree decomposition. Creating a tree decomposition of minimum width is not trivial.

Computing the treewidth of a graph is in FPT: while it is NP-complete to compute the exact treewidth, it is possible to determine whether or not a graph has treewidth less than a fixed value w. Both exact and approximation algorithms are used for computing treewidth. An algorithm by Bodlaender that runs in $w^{O(w^3)} \cdot n$ time, determines whether or not the input graph has a treewidth of at most w, and if so, returns a tree decomposition of width w [8]. Similarly, an algorithm by Korhonen gives a $(2w + 1)$-approximation and returns a corresponding tree decomposition, in $2^{O(w)} \cdot n$ time [25].

2.3.2 Nice tree decompositions and dynamic programming

In the development of tree decomposition algorithms, it is often preferable to have a tree decomposition that fulfills specific properties that aid in the basic operations of the algorithm. Such a tree decomposition, called a *nice* tree decomposition, is defined as follows:

Definition 2.4 (Nice tree decomposition)**.** *A tree decomposition T on graph G is called* nice *if it fulfills the following properties:*

1. *T is rooted in a node r, with $X_r = \emptyset$;*
2. *for all leaves ℓ of T, $X_\ell = \emptyset$;*
3. *every node has at most two children;*
4. *if a node t has two children t_1 and t_2, then $X_t = X_{t_1} = X_{t_2}$;*
5. *if a node t has one child t', then $X_t = X_{t'} \cup \{v\}$ or $X_{t'} = X_t \cup \{v\}$, for some vertex $v \in V(G)$.*

In a nice tree decomposition according to the formulation above, four different types of nodes can be distinguished:

- **Leaf:** a node t without any children, and $X_t = \emptyset$;
- **Introduce v:** a node t with one child t', such that $X_t = X_{t'} \cup \{v\}$;
- **Forget v:** a node t with one child t', such that $X_{t'} = X_t \cup \{v\}$;
- **Merge:** a node t with two children t_1 and t_2, such that $X_t = X_{t_1} = X_{t_2}$.

There is a simple linear-time algorithm to turn a tree decomposition into a nice tree decomposition of equal width. A nice tree decomposition has at most $O(n)$ nodes, with n being the number of vertices in the original graph.

In a nice tree decomposition, it is possible to define a subgraph that is represented by each node:

Definition 2.5 (Subgraph in a tree decomposition)**.** *In a nice tree decomposition T on the graph G, define for each $t \in V(T)$ the tree T_t as the subtree of T with t as its root. Then V_t is the set of vertices that are introduced in T_t, and G_t is $G[V_t]$. In other words, G_t is the part of G that has been introduced "up to this point" in node t.*

An addition to nice tree decompositions, that is sometimes useful, is an "introduce edge" node for some edge in the original graph. While this does not change any of the normal properties of tree decompositions, it can be used to treat new edges individually in computing solutions. An introduce edge node would be inserted as a child of one of its endpoints' forget nodes; with it, it is possible to define E_t for every node t, which is the set of edges that have been introduced in T_t. Then, the graph G_t becomes (V_t, E_t).

Nice tree decompositions are useful for designing dynamic programming algorithms for graph problems. Usually, the crux of the algorithm is to store for every node t and some state dependent on X_t a sort of partial solution on the graph G_t. Only the vertices of X_t are used as part of the state; X_t can be considered the "working area" while the rest of G_t has been "finished". For a broader introduction to tree decompositions, as well as some examples of tree decomposition algorithms, see Chapter 7 of [11].

2.4 Separations

In some situations, it is useful to split a graph into smaller parts. In the algorithm explored in Sections 4.5 and 4.6, a separation in a graph is used to divide relevant parts of the problem into two subproblems, as part of a recursive algorithm. Formally, a separation can be defined as follows:

Definition 2.6 (Separation)**.** *A* separation *in a connected graph G is a pair (Y, Z) with $Y, Z \subseteq V(G)$, such that G contains no paths from vertices in $Y \setminus Z$ to vertices in $Z \setminus Y$ or vice versa.*

The order *or size of the separation is $|Y \cap Z|$.*

In the case of the recursive algorithm, it is important to ensure that both halves of the separation contain a significant part of the graph's features. Such a balance in a separation can also be formalized:

Definition 2.7 (Balanced separation)**.** *In a connected graph G, an α-*balanced separation *is a separation (Y, Z) such that for a given weight function $\omega : V(G) \to \mathbb{R}^+$ on the graph's vertices, $\omega(Y \setminus Z) \leq \alpha \cdot \omega(V(G))$ and $\omega(Z \setminus Y) \leq \alpha \cdot \omega(V(G))$ (abusing notation to obtain total weights of vertex sets).*

Connecting the notion of a balanced separation with tree decompositions gives the following helpful theorem. This proves not only that a balanced separation of size bounded by the treewidth exists in a graph, but also that a tree decomposition can be used to find such a separator.

Theorem 2.1 (Lemma 7.20 from [11])**.** *Let G be a connected graph with treewidth w and let $\omega : V(G) \to \mathbb{R}_{\geq 0}$ be a weight function on G's vertices. G admits a $\frac{2}{3}$-balanced separation (A, B) of size $w + 1$.*

2.5 Graph minors

An important concept in graph theory is that of a *minor*, which is similar to a subgraph but has more powerful properties. Formally, a graph minor is defined as follows:

Definition 2.8 (Graph minor)**.** *A graph H is a* minor *of a graph G (corr. G is a* major *of H) if it can be obtained from G through a series of vertex deletions, edge deletions and edge contractions.*

A minor model *for G and H is a function that maps vertices from G to their corresponding vertices in H.*

The power of graph minors lies in the fact that certain families of graphs can be defined by which graphs are and are not minors of the graphs in said family. As a well-known example, it is known that planar graphs do not include K_5 and $K_{3,3}$ as minor, where K_5 is the complete graph of five vertices and $K_{3,3}$ is the complete bipartite graph with three vertices in each half. Another example is the following theorem:

Theorem 2.2 (Planar Excluded Grid Theorem; Theorem 7.23 from [11])**.** *Let G be a planar graph with treewidth at least $\frac{9}{2}t$ for $t \geq 0$. Then G contains the $t \times t$ grid as a minor. Moreover, for a value ϵ and fixed t, there exists an algorithm that for a graph G either returns the $t \times t$ grid as a minor model of G, or determines that G has treewidth less than $(\frac{9}{2} + \epsilon)t$ and returns a corresponding tree decomposition instead. This algorithm runs in $O(n^2)$ time.*

The power of this theorem lies in the fact that the existence of a grid as minor in a graph G can be used to make certain statements about G itself. Take, for instance, the Vertex Cover problem, which asks for a set of vertices in a graph such that every edge has an endpoint in this set. The following two facts can be easily verified with regard to vertex covers:

Observation 2.1.

1. *If G contains a vertex cover of size k, and H is a minor of G, then H also contains a vertex cover of size at most k.*
2. *A $t \times t$ grid does not contain a vertex cover smaller than $\lfloor t^2/2 \rfloor$ vertices.*

A corollary of this is the following:

Corollary 2.1. *Let G be a planar graph of treewidth $\frac{9}{2}t$ or greater. Then G does not contain a vertex cover smaller than $\lfloor t^2/2 \rfloor$ vertices.*

This concept is known as *bidimensionality*, as it exhibits a correlation between two properties of a graph. This can be exploited in designing an algorithm: an algorithm that determines if a graph has a treewidth less than a given value, could use the excluded grid theorem to either output a minor model for a grid, or return a tree decomposition of the graph. If a large grid minor exists in the graph, this immediately proves that the graph cannot have a vertex cover smaller than the input value and false can be returned. Otherwise, the algorithm conveniently returns a tree decomposition which can be used to determine a result using dynamic programming.

Chapter 3

Disjoint Paths

The Disjoint Paths problem (DPP) is a well known problem in parameterized complexity and graph theory. It asks to find a number of paths in a graph, connecting given pairs of terminals, in a way such that these paths do not intersect. While there are parameterized algorithms for the problem, it remains an extremely difficult problem to solve with few practical exact algorithms for it. Formally, the problem is stated as follows:

Disjoint Paths
Given: undirected graph G, terminals $\mathcal{P} = \{\{s_1, t_1\}, ..., \{s_k, t_k\}\}$
Asked: pairwise vertex-disjoint paths $P_1, ..., P_k$ such that for each i, P_i connects s_i to t_i

Some variants of the problem include Edge-Disjoint Paths [46], in which the paths can have vertices in common but not edges, and a variant on directed graphs. It is obvious that a solution for (vertex-)Disjoint Paths is immediately a solution for Edge-Disjoint Paths and that a Directed Disjoint Paths algorithm can also be used to solve the undirected variant. In fact, the directed variant is much harder than the undirected variant, as it is NP-hard even when $k = 2$ is fixed, on general graphs [15]. Another variant is Shortest Disjoint Paths, in which the edges of the graph have a weight and the solution should have minimum total weight [24]. The Disjoint Trees problem asks to find trees instead of paths, where instead of the input containing terminal pairs, there are multiple sets of terminals each of which needs to be connected by a tree [34]. These variants are not discussed in detail in this chapter, as the main theoretical and practical interest in DPP lies in the undirected vertex-disjoint version.

While the problem is very hard to solve in general, it is somewhat easier to solve on planar graphs. Most of the improvements have thus been on planar graphs. The reason for this is that topological properties of an embedded planar graph can be used to one's advantage in finding a solution: two paths not intersecting in a planar graph translates well to two curves not intersecting in $\mathbb{R}^2$.

This chapter discusses some of the techniques used to solve DPP, mainly on planar graphs. Some of the algorithms are explained in detail whereas other, more complicated approaches, are summarized cursorily. Section 3.1 goes over the theoretical background of the problem and the algorithm by Robertson and Seymour, with Section 3.1.1 explaining the irrelevant vertices technique, which is the core of their algorithm and a large inspiration for later research on the problem. Section 3.2 goes over an algebraic approach by Schrijver for solving DPP on directed planar graphs. Section 3.3 explains a tree decomposition algorithm for DPP, which is important for the irrelevant vertices technique. The irrelevant vertices technique is revisited in Section 3.4, which goes over an algorithm that applies it to planar graphs. Section 3.4.4 summarizes an algorithm that takes this a step further, by combining irrelevant vertices with Schrijver's algebraic approach to achieve the best-known running time for planar DPP yet. Lastly, Section 3.5 investigates the case of the problem where all terminals are incident on few faces.

3.1 Robertson and Seymour's algorithm

An important feat in the field of parameterized complexity and graph theory is the so-called Graph Minors project by Robertson and Seymour. In a series of 23 papers spanning from 1983 to 2004, they set out to prove Wagner's conjecture [40], also known as the Robertson-Seymour theorem, and managed to do so, introducing many important concepts along the way. The theorem is as follows:

Theorem 3.1 (Robertson-Seymour theorem)**.** *Any infinite sequence of graphs contains two graphs such that one that is a minor of the other; that is, the class of all graphs is well quasi-ordered by the minor relation.*

A consequence of this theorem is that every minor-closed family of graphs $\mathcal{G}$ (that is, every minor of a graph of $\mathcal{G}$ is also in $\mathcal{G}$) can be characterized by a finite set of "forbidden minors", which are not minors of any graphs in $\mathcal{G}$.

An early related theorem is Wagner's theorem [47] (equivalently Kuratowski's theorem [26]) which states that a graph is planar if and only if it does not contain the graphs K_5 and $K_{3,3}$ as minors. Here K_5 is the complete graph of five vertices, and $K_{3,3}$ is the complete bipartite graph with three vertices in both halves. It is easy to see that K_5 and $K_{3,3}$ are not planar themselves; together they form the set of forbidden minors for planar graphs. Essentially, the Robertson-Seymour theorem is a generalization of Wagner's theorem, extending the concept of forbidden minors to *all* minor-closed graph families.

In proving Wagner's conjecture, Robertson and Seymour have introduced several concepts that have become important in the field. Concepts like tree decompositions, excluded grids and the irrelevant vertex technique (all of which are used in this thesis) have found part of their origin and notoriety in the Graph Minors project.

The disjoint paths problem played a significant role in the Graph Minors project. In order to devise an FPT algorithm for Minor Testing, which tests whether or not one graph is a minor of another, Robertson and Seymour needed to prove the tractability of Disjoint Paths, which was already known to be NP-complete. They achieved this with a $f(k)n^3$ time algorithm using the *irrelevant vertex technique* (explored in Section 3.1.1) which has formed the basis for some proceeding DPP algorithms [39]. Neither the function f itself nor even its computability were known at the time of publication. It was later shown that $f(k)$ must be at least the very unfortunate value of $2^{2^{2^{2^{\Omega(k)}}}}$ [22], meaning that while it is technically FPT, it is in no way practically useful. A simple algorithm enumerating all subsets of edges of the input graph could solve DPP in $2^{|E|}$ time, which would be more practical on every realistic input despite being considered a "worse" running time.

As for ludicrous running times, Disjoint Paths is not the only offender in the Graph Minors project. The consequence of proving the fixed-parameter tractability of Minor Testing is that for every finite set of forbidden minors $\mathcal{F}$, it is possible to determine in polynomial time whether or not a given graph belongs to the family of graphs characterized by $\mathcal{F}$. The best-known algorithm for this problem, originating in the Graph Minors project and resulting from the DPP algorithm, runs in $f(k)n^2$, where $f(k)$ is "a stack of powers of 2 of height at most 5, with k^{1000} on top" [31].

Robertson and Seymour have become notorious for these running times, and give a good example of what has become colloquially known as "galactic algorithms" [28], which have an impossibly high running time despite belonging to a "good" complexity class. The Graph Minors project was one of the most important applications of parameterized complexity in the field's early days; however, while the main goal of parameterized complexity is to improve the computational tractability of hard problems, this is clearly not the case in the algorithms by Robertson and Seymour. Nevertheless, the theoretical significance of the Graph Minors project was great, and the algorithm for DPP has laid the foundations for much faster algorithms (particularly on planar graphs) which are discussed in this chapter.

3.1.1 Irrelevant Vertices Technique

The irrelevant vertex technique is a method of solving parameterized problems on large graphs when there exists an algorithm that is parameterized by the graph's treewidth. The main part of the technique is to prove that if a graph G has treewidth greater than $g(k)$, with k being the parameter of the problem, then it must be possible to identify some vertex v in the graph that is "irrelevant" to the solution. In the case of disjoint paths, some non-terminal vertex v being irrelevant means that $(G, \mathcal{P})$ is a yes-instance of DPP if and only if $(G - v, \mathcal{P})$ is also a yes-instance. The algorithm then consists of two phases:

1. as long as G has treewidth greater than $g(k)$, identify irrelevant vertices and remove them;
2. when G's treewidth has reached $g(k)$ or less, solve the problem using a dynamic programming algorithm on its tree decomposition.

In a sense, it is a way to convert an FPT algorithm parameterized by the treewidth to an FPT algorithm using another parameter, provided that there is a provable link between the treewidth and this other parameter. Proving such a link is often tedious and relies on graph minor theorems such as the excluded grid theorem (Theorem 2.2). This way, it is shown that a graph of large enough treewidth must have some part with a given structure, and this structure can be shown to contain an irrelevant vertex. Section 7.8 of [11] contains an overview of the irrelevant vertex technique and its application to Planar Vertex Deletion, which asks if it is possible to make a graph planar by removing some vertices from it.

In the case of disjoint paths, proving the irrelevance of a vertex comes down to proving that any solution can be re-routed to avoid this vertex, even when restricted to a specific minor of the graph. The algorithm by Robertson and Seymour works, in summary, as follows.

1. if G has treewidth greater than $g(k)$ and contains a large clique as minor, then one of this clique's vertices is redundant and can be removed;
2. if G has treewidth greater than $g(k)$ and does not contain a large clique minor, then it must contain a large planar piece as minor through the "flat wall theorem"; all paths can be routed through this wall to avoid a vertex in the middle, which can be removed;
3. if G has treewidth at most $g(k)$, then it is solved using the tree decomposition algorithm described in Section 3.3.

The function $g(k)$ was not defined at the time of its publication, and as mentioned before, did not end up being favorable for practical application: an estimated bound on the running time is $2^{2^{2^{2^{\Omega(k)}}}} \cdot n^3$ [22]; the polynomial part was later improved to be n^2 [21]. Nevertheless, this algorithm has inspired much later work, leading to somewhat practical algorithms, mostly on planar graphs.

3.2 Schrijver's cohomology algorithm

An important Disjoint Paths algorithm by Schrijver [43] works on directed planar graphs, but can be applied to undirected graphs as well after replacing every edge with two parallel arcs in opposite directions. Schrijver developed a language of "flows" of non-crossing walks between terminal pairs, that encodes the relative placement of different paths to one another in a solution. A flow does not necessarily equal a proper solution, as it can have multiple walks crossing a single arc and these walks can go in the opposite direction compared to the arc. However, the ordering of these walks along the arc does matter, which is what encodes the topology of a solution.

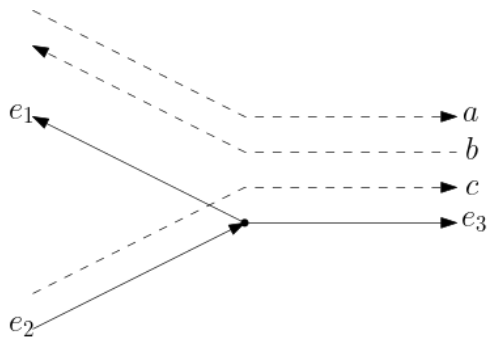


Figure 3.1: Three walks along three arcs. The walks are represented by the dashed arrows. The arc e_1 gets label ba^{-1}, e_2 gets c and e_3 gets $ab^{-1}c$.

The flow language is defined as follows. Let S and T be the sets of source and sink terminals in the problem instance. The language is a group consisting of words with generators $T \cup \{t^{-1} : t \in T\}$, with 1 being the empty word. The addition of words is performed by taking their concatenation, and removing every occurrence of the consecutive letters a and a^{-1} (or vice versa) for every sink $a \in T$. For example, $a^{-1}b \cdot b^{-1}c = a^{-1}c$. The negation of a word is obtained by replacing each letter with its inverse and reversing their order.

Now a flow is encoded by assigning to each arc in the graph a word fulfilling the following properties:

1. for every non-terminal vertex in the graph, the sum of its incident arcs' words in clockwise order, with the words of incoming arcs negated, equals 1;
2. for every sink vertex t in T, this value should be t;
3. for every source vertex s in S, this value should be $g(s)$ with $g(s)$ being the sink associated with s.

This language is best explained visually, like in Figure 3.1. Drawing different non-crossing walks along the incident arcs of a vertex, a clear ordering of the walks along each arc arises. Assigning to each walk its corresponding letter, and negating the letter for the "reverse" walks, makes it clear that the above conditions must hold for a vertex if the walks are non-crossing.

A further consequence is that every solution to the disjoint paths problem can easily be converted to its corresponding flow. To every arc in the solution, assign the letter corresponding to the path it belongs to. Every other arc gets the empty word 1.

Next, two flows are considered *cohomologous* if one can be obtained from the other by "stretching" walks across faces without any walks crossing one another, as illustrated in Figure 3.2. Algebraically this can be defined as follows:

Definition 3.1 (Cohomology)**.** *Two flows ϕ and ψ are considered* homologous *if there exists a function h associating to each face in the graph a word, satisfying the following properties:*

1. *for the outer face f, $h(f) = 1$;*
2. *for every arc a, if f_1 is the face left of a and f_2 the face right of a, $h(f_1)^{-1} \cdot \phi(e) \cdot h(f_2) = \psi(e)$.*

The equivalence relation of cohomology induces a number of homology classes.

Schrijver's algorithm consists of the following two results:

1. it is possible to enumerate, in polynomial time for fixed k, a flow for every homology class in an instance;

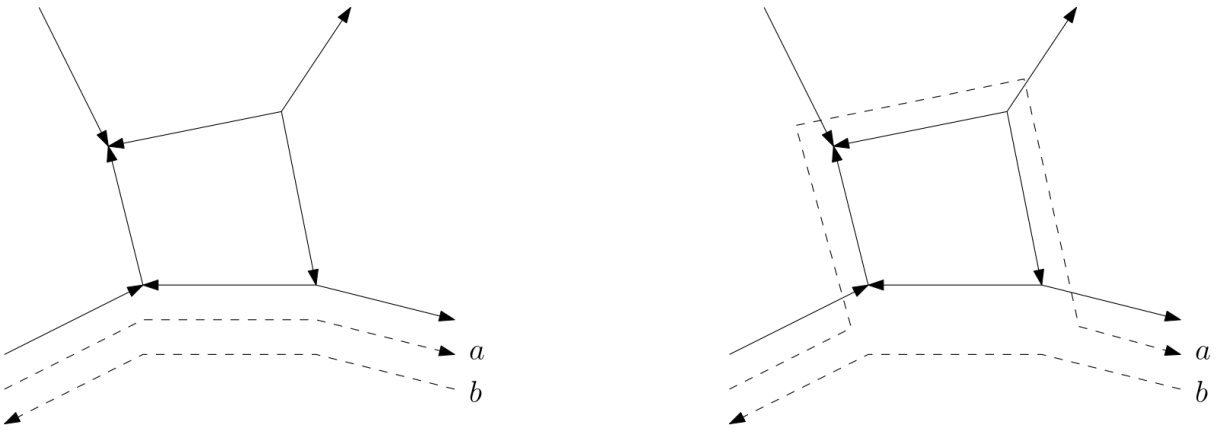


Figure 3.2: The walk a is stretched along a face f by setting $h(f) = a$. Note that the same cannot be done to b without also doing it to a, as otherwise the two walks would intersect.

2. for a given flow, it is possible in polynomial time to either find a set of disjoint paths cohomologous to it, or to determine that this is not possible.

Put together, this gives an $n^{O(k)}$ algorithm for directed planar disjoint paths, which can also be applied to the undirected variant. This puts the problem in the XP complexity class.

3.3 Dynamic programming on tree decompositions

For the second step of the irrelevant vertices technique for Disjoint Paths, it is necessary to have an FPT algorithm for the problem that is parameterized by the graph's treewidth. An algorithm by Scheffler [41] does this, and is explained in this section.

This algorithm works with "nice tree decompositions", as described in Section 2.3.2. In every node of the tree decomposition, a number of partial solutions for the problem is considered; namely, when for a tree decomposition node t, the graph G_t denotes the subgraph induced by all vertices and edges introduced in t's descendants, the considered partial solutions consist of disjoint paths inside G_t. Here some paths may fully connect two corresponding terminals to one another, while others are incomplete meaning one or both endpoints of the path are not a terminal. Moreover, all terminals in G_t are connected to some path. For these incomplete paths, the non-terminal endpoint(s) are in the set X_t, meaning they will later connect to the part of G that has not been introduced in t's subtree.

A partial solution like this, whose paths are all connected to either terminals in G_t or vertices in X_t, is called *plausible*, as there is a possibility of them later becoming full solutions to the problem. The plausible solutions that are considered in the tree decomposition's root, are all *feasible*, meaning they solve the full problem instance. This follows simply from the fact that for the root node r, $X_r = \emptyset$, and $G_r = G$; hence all paths in a feasible solution connect two corresponding terminals.

The states of the dynamic programming algorithm are distinguished by tree decomposition nodes t and functions assigning labels to all vertices in X_t, which encode the "kind" of path a vertex belongs to. For instance, if there exists a plausible solution containing a path from terminal s_i to vertex v in X_t, then this solution should be represented by a labeling that encodes that v is connected to s_i such that it may later be connected to another path that connects to t_i. For every state in the algorithm, it can simply be stored whether or not there exists a plausible solution encoded by the labeling; at the end of the algorithm, the solution(s) themselves can be determined by backtracking.

The following situations can be distinguished for vertices in X_t, and labeled accordingly:

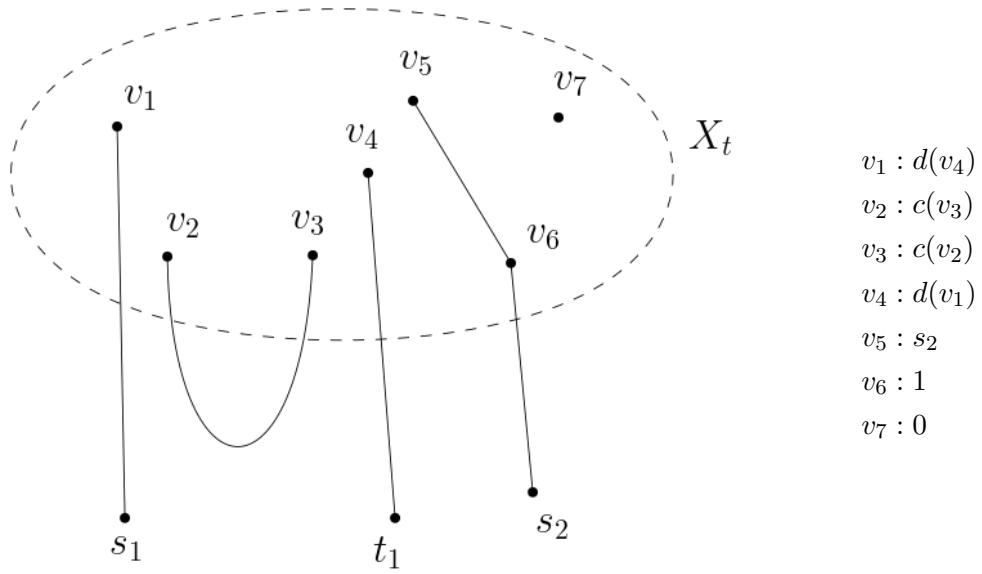


Figure 3.3: Illustration of the different vertex labels. The drawn paths represent a plausible solution corresponding to the labeling shown.

- the vertex is not part of any path in the plausible solution, and gets label 0;
- the vertex is an inner vertex (degree 2) of some path in the plausible solution, and gets label 1;
- there is some path in the plausible solution with endpoints $v \in X_t$ and terminal s_i, but the matching terminal t_i is not in G_t; v gets label s_i;
- there are paths in the plausible solution from v to s_i and from u to t_i, with $u, v \in X_t$; u gets label $d(v)$ and v gets label $d(u)$;
- there is a path in the plausible solution from v to u, neither being terminals and $u, v \in X_t$; u gets label $c(v)$ and v gets label $c(u)$.

To determine the number of different labels, the following observation is important. Note that the only times a vertex is labeled with a terminal are when said terminal is in G_t, while its corresponding terminal is not in G_t. Consider a terminal pair like this to be "separated". By the definition of tree decompositions, X_t is a separator in G, separating the subgraph $G_t - X_t$ from $G - G_t$. This means there can only be $|X_t|$ vertex-disjoint paths between $G_t - X_t$ and $G - G_t$. Therefore, there can only be at most $|X_t|$ separated terminal pairs, and thus only $|X_t|$ labels for them.

From this, it follows that there are at most $2 + 3|X_t|$ different labels that can be assigned to a vertex at node t, and thus $w^{O(w)}n$ states in total. For most nodes, the plausible solutions to states can be computed very easily. For instance, if at one node we know that vertex u gets the label s_i, and the next node introduces a vertex v with an edge to u, then either v "continues" u's path and gets the label s_i itself (and u gets label 1), or v is not part of that path and gets label 0 with v's label unchanged. So for most nodes, computing the answers to states comes down to simply checking those of a limited number of states in the previous node. The more difficult part is in the merge nodes, which combine the answers from two separate tree decomposition nodes; here the algorithm needs to go over all pairs of states in the two nodes to combine their labelings to end up with a labeling for the merge node, check whether or not this labeling is permissible, and saying it has plausible solutions if both substates do.

All in all this puts the algorithm's running time at $w^{O(w)}n$ time (or commonly $2^{O(w \log w)}n$) with w being the graph's treewidth. This is actually a tight running time, as it was proven that under the Exponential Time Hypothesis, Disjoint Paths cannot be solved faster than $2^{\Omega(w \log w)}n^{O(1)}$ [29] time (on planar graphs, this bound is $2^{\Omega(w)}n^{O(1)}$ [4]).

3.4 Irrelevant vertices on planar graphs

The irrelevant vertices technique was applied to Disjoint Paths on planar graphs by Adler et al. [2]. The technique is much the same as the algorithm by Robertson and Seymour on general graphs, but the proof for finding an irrelevant vertex is different and leads to a much better result: it is shown that on a planar graph, irrelevant vertices can be identified and removed as long as the graph has treewidth $\Omega(2^k)$, leading to a $2^{2^{O(k)}} \cdot n^2$ time algorithm. The proof is highly technical and exploits the combinatorial and topological properties of planar graphs; a summary is given in this section.

The central claim of the paper is as follows:

Theorem 3.2 (Theorem 1 from [2])**.** *Every instance of* Disjoint Paths *consisting of a planar graph G with treewidth at least $82 \cdot k^{3/2} \cdot 2^k$ $(= \Omega(2^k))$ and k pairs of terminals contains a vertex v such that every solution to it can be replaced by an equivalent one whose paths avoid v.*

This irrelevant vertex can then, as shown in the next few sections, be determined easily during the algorithm. By the excluded grid theorem (Theorem 2.2) it is possible to find a large grid minor in a planar graph, with the grid's dimension being linear to the treewidth of the graph, and a part of this grid is used which does not contain any terminals. It is then claimed that the centermost vertex of this part of the grid is irrelevant to the solution.

In this section, I attempt to summarize the proof of this irrelevance without going into too many technical details. The smaller results building up the full proof are omitted, but their respective sections in the original paper are referenced. Moreover, the original paper introduces a lot of concepts that are essential to the proof; where possible, I refrain from naming these as to make the proof more readable albeit at the cost of preciseness.

3.4.1 Concentric cycles and linkages

The first part of the proof consists of defining the working grid, as well as the disjoint paths solutions, in a more workable way. The grid is observed as being a set of concentric cycles, and a set of disjoint paths is called a *linkage*. The combination of these two concepts allows one to observe their intersection, a larger collection of paths within the cycles, and rearrange these paths in a way to avoid the irrelevant vertex at hand.

Definition 3.2 (Concentric cycles)**.** *Let $\{C_0, ..., C_r\}$ be a set of pairwise disjoint cycles in a plane graph, with each C_i being fully contained in the interior of C_{i+1}. This is called a set of* concentric cycles*.*

A set of concentric cycles is considered tight *if C_0 does not contain any chords, and for every C_i with $i > 0$, there exists no cycle that is smaller than C_i but still has C_{i-1} entirely in its interior.*

Note that a set of m concentric cycles can be obtained from a grid minor with a dimension of at least $2m - 1$: the grid naturally contains cycles (squares) centered around its middle vertices, with each vertex of a cycle corresponding to a cycle in the original graph and each edge in a cycle corresponding to some path between its endpoints. This set of concentric cycles can then be made tight by identifying chords within cycles (such that the chords do not intersect the smaller cycles) and replacing part of the cycle with the chord until this is no longer possible.

Lemma 3.1 (Lemma 1 from [2]). *Given a plane graph G, a positive integer r and a set of terminals $T \subseteq V(G)$, there exists an algorithm that either outputs a tree decomposition of G with width at most $9 \cdot (r+1) \cdot \lceil \sqrt{|T|+1} \rceil$, or returns a set of tight concentric cycles $\mathcal{C} = \{C_0, ..., C_r\}$ such that none of the terminals of T are inside C_r. This algorithm runs in $2^{(r \cdot \sqrt{|T|})^{O(1)}} \cdot n$ time.*

This lemma follows from the planar excluded grid theorem (Theorem 2.2)[1]: a sufficiently large grid is found and subdivided into $|T|+1$ equal-sized smaller grids, at least one of which does not contain any terminals.

The sets of disjoint paths are defined as follows:

Definition 3.3. *A* linkage *is a set L of pairwise vertex-disjoint paths in a graph. The* pattern *of L is the set of pairs of its paths' endpoints, and two linkages are called* equivalent *if they have the same pattern.*

Note that a linkage L is a solution to a Disjoint Paths instance if and only if it has as pattern the set of terminal pairs $\mathcal{P}$ in the instance's input.

Cheap and convex linkages

To prove the existence of an irrelevant vertex, the combination of concentric cycles and a linkage is considered. The claim is that every linkage can be "re-arranged" to avoid the innermost cycle C_0 of these concentric cycles, given that the set of concentric cycles is large enough and does not contain any terminals. To do this, a cost function is defined for linkages on concentric cycles, and it is shown that a linkage can be easily rearranged to become a linkage of minimum cost. It is then shown that a linkage of minimum cost fulfills some desirable properties, and that it must avoid the cycle C_0.

Definition 3.4 (Cheap linkage). *Let $\{C_0, ..., C_r\}$ be a set of tight concentric cycles in G. Let $\mathcal{L}$ be the family of all linkages in G and define the* cost function *$c : \mathcal{L} \to \mathbb{N}$ as follows:*

$$c(L) = \left| E(L) \setminus \bigcup_{i=0}^{r} E(C_i) \right|.$$

In other words, the cost *of a linkage is defined by the number of edges in it that do not run along any of the cycles.*

A linkage is called cheap *if there is no equivalent linkage with lower cost.*

A single path from a linkage can intersect the concentric cycles several times. To determine the properties of a cheap linkage, it is necessary to look at these individual intersecting segments of paths, rather than the paths as a whole.

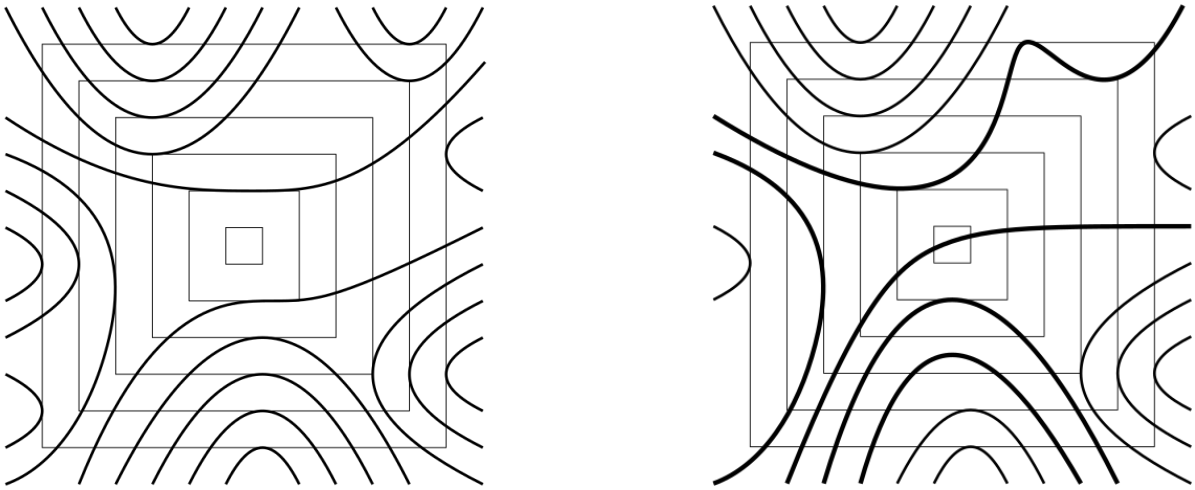
Definition 3.5 (Segments). *Let L be a linkage and $\mathcal{C} = \{C_0, ..., C_r\}$ a set of concentric cycles. The* segments *of L and $\mathcal{C}$ are the connected components of the intersection of L with C_r's interior. The* eccentricity *of a segment P is the lowest i such that P intersects C_i.*

One thus has to consider the eccentricities of the different segments of the linkage. Intuitively, a higher eccentricity is better for the linkage's cost, as that would mean the segment penetrates the cycles less deeply and therefore uses fewer edges between the different cycles.

Definition 3.6 (Convex segment and linkage). *A segment P of L and $\mathcal{C} = \{C_0, ..., C_r\}$ is called* convex *if it holds the following properties:*

1. *P does not contain a chord of C_0;*

[1] The paper actually cites two different algorithms than the one of the original planar excluded grid theorem; this achieves the same result but does improve the running time dependence on n from quadratic to linear.



(a) a convex linkage (b) a concave linkage

Figure 3.4: An example of a convex linkage and a concave linkage. In (b), the concave paths are emboldened.

2. *for every i with $1 \leq i \leq r$, the following hold:*
   (a) *P intersects the interior of C_i at most once;*
   (b) *if P contains a component in the interior of a cycle C_i, then it must also intersect C_{i-1};*
   (c) *the component in the interior of C_i must have exactly two subpaths connecting C_i and C_{i-1};*
3. *if P has eccentricity $i < r$, there must be another path "underneath" P with eccentricity $i + 1$.*

A linkage is called convex if all its paths are convex.

Intuitively, a linkage can be made convex by "pushing" all the segments as far to the outside of the cycles as possible, without making any concave bends in them.

The following claim is made:

Lemma 3.2 (Lemma 3 from [2])**.** *Let $\mathcal{C}$ be a set of tight concentric cycles and let L be a linkage. If L is cheap, then it is also convex.*

What this means, is that the authors can consider a cheap linkage and immediately assume that it is convex. The final proof for the existence of an irrelevant vertex then shows that if a linkage is convex and intersects C_0, this contradicts the cheapness of the linkage.

Parallel segments

An important part of the irrelevant vertices proof relies on finding a set of segments inside the cycles that run "parallel" to each another; a section of these segments then forms the part where an improvement in the linkage's cost can be found. It is thus important to define what it means for two segments to run parallel, and prove that a large enough class of parallel segments exists under the right conditions.

Definition 3.7 (Parallel segments)**.** *Let S_1 and S_2 be two segments, and P and P' the paths along C_r between pairs of the segments' endpoints, such that P and P' do not contain any other endpoints of the two segments. S_1 and S_2 are* parallel *if the following conditions are met:*

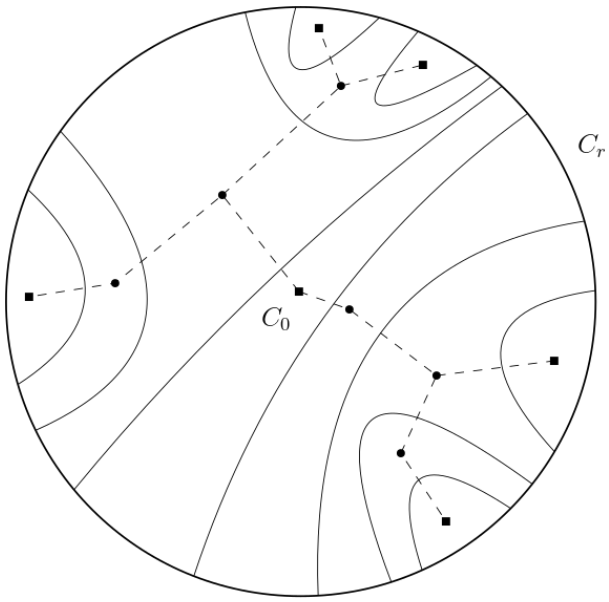


Figure 3.5: Segments inside an outer cycle, and the corresponding segment tree. The solid lines depict the segments and the outer cycle, whereas the dashed lines represent the segment tree.

1. *there is no segment with both endpoints on P or both endpoints on P';*
2. *the area enclosed by S_1, P, S_2 and P' does not contain the interior of C_0.*

Different classes of parallel segments are induced by this relation. A way of identifying and counting these classes is as follows. Consider the graph H with only C_r in it and all segments of the linkage, dissolving all vertices of degree 2. This graph is outerplanar, as every edge is either incident on the outer face or connects two vertices on the outer face. The *weak dual* T of this graph is obtained by creating a vertex for every face in H other than the outer face, and connecting two vertices if their respective faces have an edge in common.

Note that T is a tree, and that every edge in it corresponds to some segment in the linkage. Moreover, every maximal 2-path in T corresponds to an equivalence class under the parallel relation.

Take the face of H that contains the interior of C_0, and let its corresponding vertex be the root of the tree T. The following properties can be defined on T:

Definition 3.8 (Height, real height and dilation)**.**

- *The* height *of T is the length of the longest path from the root to a leaf.*
- *The* real height *of T is the maximum number of vertices with degree greater than 2 on a path from the root to a leaf, plus one.*
- *The* dilation *of T is the maximum length of all paths in T, that do not include the root or any vertices with degree three or greater.*

Now suppose that the linkage L is convex in $\mathcal{C} = \{C_0, ..., C_r\}$, which itself is tight, and the tree T is as defined. The following observations can be made about T and L:

Observation 3.1 (Observation 3 from [2])**.** *The dilation of T is equal to the size of the largest parallel class of L's segments.*

Observation 3.2 (Observation 4 from [2])**.** *The height of T is upper bounded by its real height multiplied by its dilation.*

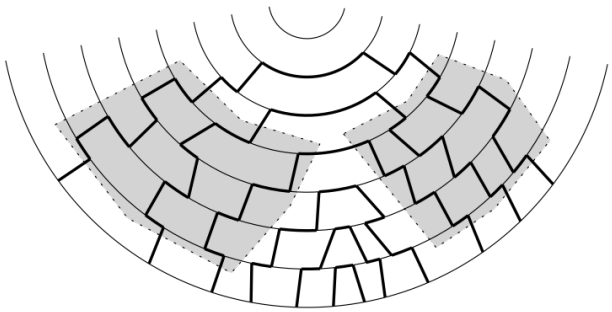


Figure 3.6: Two tilted grids of dimension 3 in a convex linkage.

Observation 3.3. *Let k be the minimum eccentricity of all paths in L. Then the height of T is $r - k$.*

Moreover, the following claim is proved:

Lemma 3.3 (Lemma 5 from [2])**.** *Suppose $L \cap C_r \neq \emptyset$ and no path from L touches C_r exactly once. Then the real height of T is at most $2 \cdot |\mathcal{P}(L)| - 3$, where $\mathcal{P}(L)$ is the number of paths in L.*

An implication of the previous statements is the following:

Corollary 3.1. *Let $\mathcal{C} = \{C_0, ..., C_r\}$ be a tight set of concentric cycles and let L be a linkage that is convex in $\mathcal{C}$, such that $L \cap C_r \neq \emptyset$ and no path in L touches C_r exactly once. If there exists a path that touches C_0, then the segments of L contain a parallel class of size at least $\frac{r}{2 \cdot |\mathcal{P}(L)| - 3}$.*

3.4.2 Tilted grids and rearrangement

Now that a proposition is given about the size of some parallel class of segments, it is still necessary to show how these segments can be rearranged. The part of the segments that is worked with, is called a *tilted grid* (defined immediately), and it is shown that one exists of sufficient size.

Definition 3.9 (Tilted grid)**.** *A* tilted grid $\mathcal{G} = (\mathcal{X}, \mathcal{Z})$ *contains two sets of r, respectively horizontal and vertical, vertex-disjoint paths in a graph, such that the following conditions are met:*

1. *for every i, j with $1 \leq i, j \leq r$, $I_{ij} = X_i \cap Z_j$ is a path;*
2. *for every j with $1 \leq j \leq r$, the paths $I_{1j}, ..., I_{rj}$ appear in order along X_j;*
3. *for every i with $1 \leq i \leq r$, the paths $I_{i1}, ..., I_{ir}$ appear in order along Z_i;*
4. *$E(I_{11}) = E(I_{r1}) = E(I_{rr}) = E(I_{1r}) = \emptyset$.*

Note that an $r \times r$ tilted grid is a topological major[2] of the $r \times r$ (normal) grid: contracting the paths I_{ij} into single vertices and the parts between the I-paths into single edges yields a normal grid.

It can be seen, like in Figure 3.6, that a tilted grid is part of some parallel segments in a convex linkage. The vertical paths of the tilted grid are all part of segments, with the horizontal paths being part of the cycles. In fact, the following claim is made:

[2] A *topological minor* of a graph is a minor that is obtained through edge contractions only; this is a stronger requirement than a regular minor and sometimes useful.

Lemma 3.4 (Lemma 6 in [2]). *Let $\mathcal{C}$ be a set of tight concentric cycles in a graph G and let L be a convex linkage. If the segments of L in $\mathcal{C}$ contain a parallel class of size k, then G contains a tilted grid of dimension $\lceil k/2 \rceil$, of which all vertical paths are part of paths in L.*

It is then shown that the parts of segments inside a tilted grid can be rearranged, if there are enough segments, to strictly utilize a smaller part of the grid. The proof of this is itself based on an earlier theorem, about reorganizing (geometric) segments inside a disk in a similar manner (see Theorem 2 from [3]).

Lemma 3.5 (Lemma 9 in [2]). *In a graph G, let L be a linkage consisting of k paths, and let $\mathcal{G}$ be an $m \times m$ tilted grid in G such that all its vertical paths are part of L. If $m > 2^k$, then G also contains a linkage L' such that:*

1. *L' is equivalent to L;*
2. *L' and L are equal in the parts of paths that fall outside of $\mathcal{G}$;*
3. *fewer of $\mathcal{G}$'s vertical paths are part of segments of L' than of L.*

In other words, L can be re-routed inside $\mathcal{G}$ to use fewer of $\mathcal{G}$'s vertical paths. As the vertical paths in a tilted grid obtained by Lemma 3.4 represent edges between cycles, which count towards a linkage's cost function, the "new" linkage is strictly cheaper than the original one.

3.4.3 Proof of irrelevant vertices

With the main ingredients summarized, it is now possible to sketch out the proof of Theorem 3.2. Obviously, this proof is not complete as it omits some of the finer details, and none of the claims in the previous sections have been fully substantiated. See [2] for full details.

Proof of Theorem 3.2 (sketch). Let k be the number of terminal pairs in the problem instance and let $r = k \cdot 2^{k+2}$. By Lemma 3.1, one can find a set of tight concentric cycles $\mathcal{C} = \{C_0, ..., C_r\}$ in the graph if the treewidth is large enough; this can be done in $2^{2^{O(k)}} \cdot n$ time.

Let L be a linkage, whose pattern is the terminals of the problem instance, that is cheap in $\mathcal{C}$ (and therefore convex). If $k = 1$, then this cheap linkage would avoid C_0 immediately (because $r > 0$), so assume $k > 1$.

For each i with $0 \leq i \leq r$, let $\mathcal{C}^{(i)} = \{C_0, ..., C_i\}$, ie. the $i+1$ innermost cycles of $\mathcal{C}$. Say that a set of concentric cycles is *touch-free* with L if no path in L touches its outermost cycle exactly once. Because $\mathcal{C}$ has $r+1$ cycles and $r+1 > k$, there must be some $\mathcal{C}^{(i)}$ that is touch-free. Let h be the greatest index of a touch-free set of cycles, and say $\mathcal{C}' = \mathcal{C}^{(h)}$.

Let L' be the linkage with paths in L that intersect $\mathcal{C}'$ and let k' be the number of paths in L'. Let $d = r - h$, and observe that $k' \leq k - d$, because every $\mathcal{C}^{(i)}$ with $i > h$ intersects at least one path that does not intersect $\mathcal{C}^{(i-1)}$.

Assume that L' is cheap and that it contains a path with eccentricity 0. By Corollary 3.1, L' and $\mathcal{C}'$ must contain a class of parallel segments of size $\frac{h}{2k'-3} > \frac{k \cdot 2^{k+2} - d}{2k - 2d} > \frac{k \cdot 2^{k+2}}{2k} = 2^{k+1}$. By Lemma 3.4, these segments must contain a tilted grid with dimension 2^k, its vertical paths being part of the segments in L'.

By Lemma 3.5, the paths inside this tilted grid can be rearranged to use strictly less than the 2^k vertical paths. Since the vertical paths in the tilted grid contain edges that count towards the cost function of linkages, this means the rearrangement can be used to create an equivalent linkage L'' with $c(L'') < c(L')$, which contradicts the fact that L' is cheap.

Due to this contradiction, it is impossible that a cheap linkage on $\mathcal{C}'$ contains a path with eccentricity 0. It follows that every linkage can be rearranged to avoid the vertices in C_0, making these vertices irrelevant. □

The algorithm follows immediately from this proof and the irrelevant vertices technique. At every step of the algorithm, the excluded grid theorem is used to either find a tree decomposition of satisfactory width, or identify a large enough set of concentric cycles in the graph that does not contain any terminals. In the latter case, the vertices in the innermost cycle are removed from the graph and the process is repeated. Otherwise, when the treewidth has reached a small enough value, the obtained tree decomposition is passed to the algorithm described in Section 3.3.

Since the graph has n vertices and every iteration of the vertex removal process removes at least one vertex, this part of the algorithm is performed at most n times and takes $2^{2^{O(k)}} \cdot n^2$ time. The treewidth is reduced to $O(2^k)$ and the tree decomposition algorithm takes $2^{O(w \log w)} \cdot n$ time, making the full running time of this algorithm $2^{2^{O(k)}} \cdot n^2$.

This is a tight bound: Adler and Krause [1] showed that a treewidth reduction beyond $O(2^k)$ is not possible, by constructing a planar graph with k terminal pairs that has a $2^k \times 2^k$ grid as minor, but requires all vertices for a solution; hence there is no irrelevant vertex. Moreover, it is theorized that the problem does not permit a $2^{o(w \log w)} \cdot n^{O(1)}$ [29] time algorithm parameterized by the treewidth (or $2^{o(w)} \cdot n^{O(1)}$ [4] on planar graphs), under the Exponential Time Hypothesis. Therefore, the existence of a faster algorithm for Planar Disjoint Paths was posed as an open question. A faster algorithm was indeed found, by Lokshtanov et al., which combines treewidth reduction with an approach similar to Schrijver's algorithm (Section 3.2). This algorithm is briefly summarized in the next section.

3.4.4 Irrelevant vertices and discrete homotopy

A recent algorithm by Lokshtanov et al. [30] combines the above result with Schrijver's algebraic method of homology. While the exact algorithm is very complicated, a rudimentary overview is as follows. First, the irrelevant vertices algorithm from Section 3.4 is used to reduce the treewidth to $2^{O(k)}$. Then, a notion similar to Schrijver's homology is defined which the authors call *discrete homotopy*. The key difference with homology is that different flows, now called *weak linkages* have to be edge-disjoint, meaning multiple walks can no longer run parallel along the same edges. However, the walks can still meet in vertices as long as they do not cross.

Next, the authors define a structure they call a *backbone Steiner tree*. This is a subtree, with the terminals as leaves, of the *radial completion* of the graph. The radial completion of a graph is obtained by adding a vertex for each existing face, and connecting it to every vertex incident on said face. The backbone Steiner tree furthermore has to satisfy some extra properties with regard to the length of its internal paths. The idea of the algorithm, is that a solution could be "pushed onto" the backbone Steiner tree by stretching paths across faces. This means that the paths are stretched around faces to be moved as close to the backbone Steiner tree as possible, resulting in a weak linkage that is discretely homotopic to the solution we started with.

It follows that one only needs to consider a limited number of weak linkages with regards to the backbone Steiner tree; if a solution exists, it will be discretely homotopic to one of said weak linkages. A modification of Schrijver's algorithm can be used to obtain a solution (if it exists) from a weak linkage in polynomial time. Moreover, the fact that the treewidth was reduced to $2^{O(k)}$ is used to further limit the number of weak linkages considered, due to the connection between a low treewidth and separators in the graph. The algorithm runs in $2^{O(k^2)} \cdot n^{O(1)}$ time in total.

3.5 Planar disjoint paths with terminals on few faces

Of interest in improving the running time of Planar Disjoint Paths, is the case in which all the terminals are incident on a limited number of faces. In this variant (to which I refer as the "few-faces variant"), the problem is parameterized by the face cover number of the

terminals rather than the number of terminals itself. This may be particularly interesting for the application of VLSI chip design, as the "terminals" for wires on a circuit tend to appear together in islands, meaning a large number of wires might be incident on a small number of islands. The flat design of a circuit and the occurrence of islands map well to a planar graph with few terminal faces.

Not much research has been done about this variant of the problem. It was proven early on by Robertson and Seymour [38] that the cases for one and two terminal faces can be solved in linear time, but larger face cover numbers were left as an open question. It was later shown by Schrijver that a parameterized algorithm exists for this problem [42]. In this section, I explain the simple algorithm for the one-face variant, and briefly discuss the techniques for two and more faces.

3.5.1 One face

When all terminals are on the border of a single face, the problem can be solved using a simple greedy algorithm in linear time. The basis of this algorithm is the following observation:

Lemma 3.6. *Let $(G, \mathcal{P})$ with $|\mathcal{P}| \geq 2$ be an instance of* Planar Disjoint Paths*, where all terminals lie on the border of a single face F. Let $\{s_i, t_i\}$ be some pair of corresponding terminals in the instance, and let P be a path between s_i and t_i that only uses vertices and edges from F's boundary. If this path P intersects no terminals other than s_i and t_i, then $(G, \mathcal{P})$ is a yes-instance if and only if $(G - P, \mathcal{P} \setminus \{\{s_i, t_i\}\})$ is a yes-instance.*

Proof. Suppose there exists a solution to the instance $(G, \mathcal{P})$, in which s_i and t_i are connected by the path P' (without necessarily $P = P'$). The path P' separates the entire graph into two parts. One of these parts (say G') contains no terminals whereas the other part contains all the terminals. Since G' contains no terminals, none of the other paths in the solution can contain vertices or edges from G', lest those paths would intersect P'. Therefore, P' could be replaced in the solution with any other $s_i - t_i$ path that contains only the vertices and edges in $G' \cup P'$. The path P fulfills these conditions, hence every solution to the instance can be replaced by an equivalent solution where s_i and t_i are connected by P. The consequence is that there exists a solution containing P if and only if there exists a solution at all, proving the lemma. □

Additionally, the following can be proved:

Lemma 3.7. *Let $(G, \mathcal{P})$ with $|\mathcal{P}| \geq 2$ be an instance of* Planar Disjoint Paths *with all terminals on the border of a face F. If there exists no pair of terminals $\{s_i, t_i\}$ such that the path between them along F's border contains no other terminals, then $(G, \mathcal{P})$ is a no-instance.*

Proof. Suppose there is no pair of "adjacent" terminals as described. Then there must be some two terminal pairs $\{s_i, t_i\}$ and $\{s_j, t_j\}$ such that they appear cyclically along F's border in the order s_i, s_j, t_i, t_j. Suppose there exists a solution to this instance where s_i and t_i are connected by the path P_i and s_j and t_j are connected by P_j. Obviously, P_i cannot contain s_j or t_j. Thus, P_i separates the graph into two parts, one of which contains s_j and the other contains t_j. It follows that P_j must intersect P_i, contradicting the fact that this is a solution to the instance. Therefore there is no solution possible to the instance. □

The algorithm follows simply from this. Make an ordering of the terminals cyclically around their common face, and identify a pair of corresponding terminals that appear adjacently in this ordering. Construct the path P around the face's border, and add it to the solution. Then, remove this path and said terminal pair from the input, and repeat the process until either no pair of adjacent terminals can be found, or a full solution is reached.

Robertson and Seymour extended Lemma 3.7 to the following:

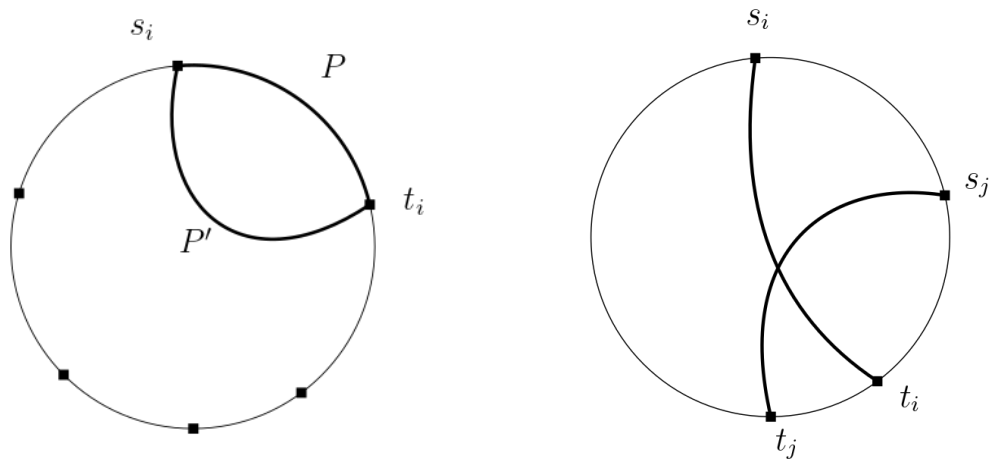


Figure 3.7: Supporting images for Lemmata 3.6 and 3.7. In both cases, the terminal face is embedded as the outer face.

Theorem 3.3 ([38]). *A* Planar Disjoint Paths *instance with all terminals on one face is a yes-instance if and only if:*

1. *there exists a pair of corresponding terminals that do not have any other terminals between them on the face's border;*
2. *each simple closed curve in $\mathbb{R}^2$ intersects G at least as often as it separates terminal pairs.*

The second condition is more or less an extension of Menger's theorem, which states the following:

Theorem 3.4 (Menger's theorem [32]). *Let x and y be two distinct vertices in a graph G. Then the size of a minimum vertex separator between x and y is equal to the number of pairwise vertex-disjoint*[3] *paths between x and y.*

It can intuitively be seen that if a graph is separated by a set of vertices X, and there need to be k disjoint paths between the two parts of the separation, then the size of X must be at least k, since every such path should use at least one unique vertex from X. As Robertson and Seymour proved, this condition as well as Lemma 3.7, are the only two conditions required to identify yes-instances of the problem.

In the above formulations, it is always assumed that the border of the terminal face is a simple cycle. This is not always the case, as the face can for instance have some appendages that are connected to the rest of the face's boundary by a single edge. It is more difficult to formulate an ordering on the terminals in this case, but the lemmata do not essentially change, nor does the algorithm. For an overview of adapting the algorithm to this case, see Section 9.4 of [44].

3.5.2 Two faces

The case of all the terminals being incident on two faces is more complicated to solve than that with one face, but still permits a linear running time. Robertson and Seymour solved the problem for one and two faces in the Graph Minors series [38], leaving more than two faces as an open question. Ripphausen-Lipa et al. solved the problem on two faces independently [35], with an algorithm that is summarized in this section.

[3] Disjoint in all vertices save for x and y

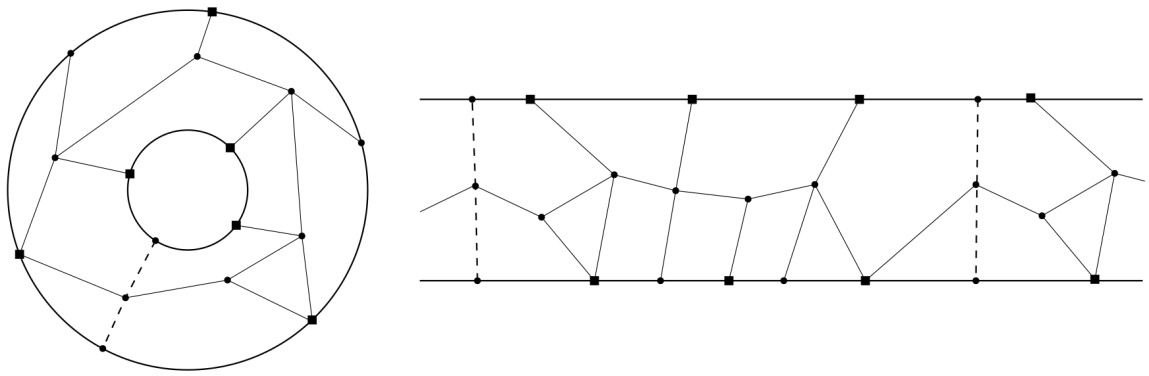


Figure 3.8: A graph G with two terminal faces, and a section of the infinite graph G^* that is obtained by "rolling out" G. The dashed segments denote the cutting path, the bottom of G^* is the outer face and the top is the inner face.

In the algorithm, the two terminal faces are labeled as the "inner face" and the "outer face" (with the outer face of course being represented by the infinite face in illustrations). The rest of the graph is shown as a band or cylinder between these two faces; formally, it is treated as an infinite graph G^* (called a band graph) that repeats itself horizontally, with the inner and outer faces at its top and bottom. To create G^*, a "cutting path" p^* between the inner and outer face is used to delimit each copy of the original graph in G^* (see Figure 3.8 for an example). By doing this, it is possible to distinguish between the different repetitions of singular vertices in the graph, which is helpful for determining the right intersections between paths.

Firstly, it can be assumed that every terminal pair has one terminal on the inner face and the other on the outer face (say, $s_1, ..., s_k$ are on the inner face and $t_1, ..., t_k$ are on the outer face). If this is not the case, the greedy algorithm from Section 3.5.1 can be used to eliminate all pairs for which this is not the case, for the conditions of the one-face algorithm still apply in this case.

The following observation is used to build a solution in the algorithm. Say that a vertex v on the border of a terminal face is *more right than* a vertex u on the same face if v appears to the right of u within some copy of the graph in G^*, or $v = u$. Following this, an $s - t$ path p is more right than an $s' - t'$ path p' if s is more right than s' and t is more right than t'. With these notions, the following statement holds:

Lemma 3.8 (Theorem 4.2 from [35])**.** *Let $P = \{p_1, ..., p_k\}$ be a set of pairwise vertex-disjoint paths, with each p_i starting in s_i on the inner face and ending in an arbitrary vertex on the outer face. Conversely, let $P' = \{p'_1, ..., p'_k\}$ be a set of pairwise vertex-disjoint paths, each p'_i starting in t_i on the outer face and ending in an arbitrary vertex on the inner face.*

There exists a solution to the problem instance if and only if, with regard to some cutting path p^* creating G^*, for every i, the end vertex of p_i is more right than t_i and the end vertex of p'_i is more right than s_i. This solution can be built from the sets P and P', by taking for every i the "left-most" segments of the two corresponding paths.

An example of how these paths are combined is shown in Figure 3.9. The algorithm is built on the mentioned observation. First, it is shown that the desired path sets can be determined using an algorithm by the same authors for the following problem: [36]

Menger Problem
Input: graph G, distinct vertices $s, t \in V(G)$
Asked: as many vertex-disjoint paths between s and t as possible

The algorithm is utilized in the following way. Say we are computing the set P, which connects terminals on the inner face with vertices on the outer face. Create a vertex s that

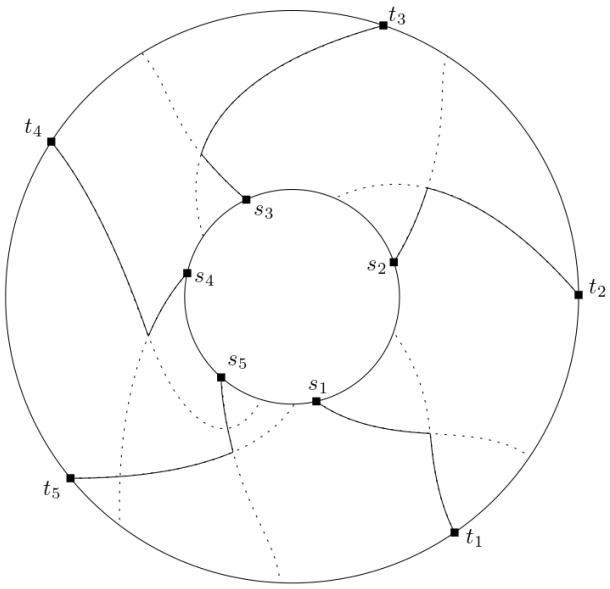


Figure 3.9: Pairs of paths being used to connect the terminals. Note that every path ends more right than (in counterclockwise direction of) its matching terminal.

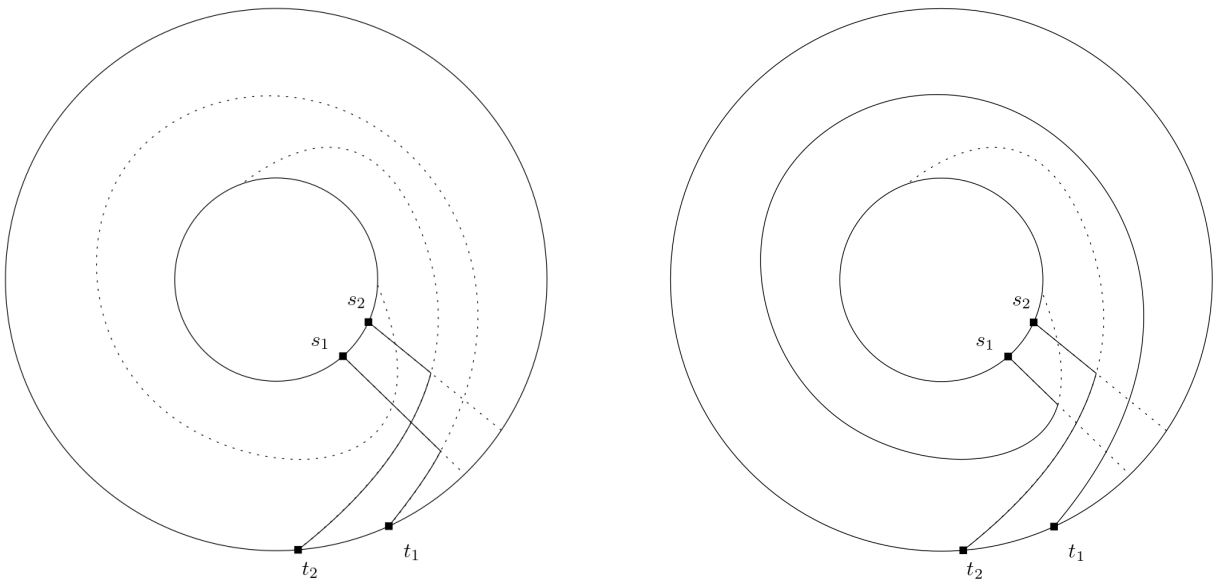
is connected to all terminals on the inner face, and a vertex t that is connected to all vertices (including non-terminals) on the outer face: the algorithm will obviously return at most k paths between inner terminals and outer vertices. It works for this application specifically, because the designed algorithm returns "right-most" (or left-most) solutions to the Menger problem, meaning that all the returned paths somewhat trend to the right, due to the nature by which vertices are considered. It is claimed that running the Menger algorithm with left-most paths for P and with right-most paths for P' returns a desirable set of paths. The algorithm runs in linear time.

The remaining ingredient of the algorithm is the cutting path p^*. Specifically, it is asked that p^* is a path between s_1 and t_1. To solve the problem, first the following simpler problem is considered:

p-Homotopic Two-Face Disjoint Paths
Input: planar graph G, terminals $\mathcal{P} = \{\{s_1, t_1\}, ..., \{s_k, t_k\}\}$ with terminals incident on two faces, path p between s_1 and t_1
Asked: pairwise vertex-disjoint paths connecting the terminal pairs, such that the path between s_1 and t_1 is homotopic to p

A solution to this problem follows immediately from Lemma 3.8. The path p can be used as the cutting path; now all found paths need to be more right than this cutting path and the found path between s_1 and t_1 will be homotopic to p.

For the regular version of the problem, this is more difficult because there is not a given cutting line. Simply using the Menger algorithm to take right-most paths is not enough, as it can happen that two resulting paths cross one another. Specifically, this happens in the following situation. Let e_i be the endpoint of a path in P that starts in terminal s_i. Then, if the vertices t_2, t_1, e_1, e_2 appear in that order along the face boundary, the two resulting paths will cross. However, the authors show that this can only happen when two corresponding paths intersect twice (after one "wraps" one or more times around the inner face), and that it can be mitigated by choosing a different intersection point of the two paths to construct their resulting path. This principle is shown in Figure 3.10. Thus, the problem can be solved in linear time by constructing the two path sets using the Menger algorithm, and then mitigating these intersection problems.



(a) not a valid solution (b) valid solution

Figure 3.10: The paths corresponding to s_1 and t_1 intersect twice, and it matters which intersection is chosen for combining the two.

3.5.3 More than two faces

For planar disjoint paths with terminals on a given, arbitrary number of faces as a parameter, an algorithm was developed by Schrijver. [42] Initially, he set out to solve the following problem:

Disjoint Homotopic Paths
Input: a planar graph G embedded in $\mathbb{R}^2$, a set of faces $I_1, ..., I_p$ in G and a set of paths $C_1, ..., C_k$ in $\mathbb{R}^2 \setminus (I_1 \cup ... \cup I_p)$ whose endpoints are on the borders of $I_1, ..., I_p$
Asked: a set of disjoint paths $J_1, .., J_k$ in G that are homotopic to $C_1, ..., C_k$

Two paths are said to be homotopic if they have the same endpoints and there exists a continuous function mapping one path to the other. In simpler terms, for each path from C_i it must be possible to "stretch" it through the space to make it equal to J_i, without it going through any of the given terminal faces.

It is shown that a solution does not exist, if it is possible to draw a closed curve D through the graph that satisfies certain properties with regards to its crossings with G and each C-curve.

A special infinite embedding G^* of the graph is made in $\mathbb{R}^2$, called the *universal covering space*, in which different homotopies of similar paths can be uniquely encoded (for instance, if P_1 and P_2 are two $u - v$ paths that are not homotopic, then their embeddings in G^* have a different ending vertex). Using the universal covering space, it is possible to test whether or not two paths are homotopic.

To solve Disjoint Homotopic Paths, first the input curves are rearranged in a way that they do not cross (but may still touch); then, using a system of linear inequalities, it is computed by what distance these curves need to be shifted in the graph in order to become disjoint. The resulting paths are then deduced from these curves.

With this algorithm for Disjoint Homotopic Paths, the regular Disjoint Paths problem is solved in a manner similar to Schrijver's algorithm on general planar graphs (Section 3.2). A number of different homotopy classes for the paths is considered, with unique representative paths for each. These representative paths are then passed to the Disjoint Homotopic Paths algorithm; if any of them gives a result, this is also a result for Disjoint Paths.

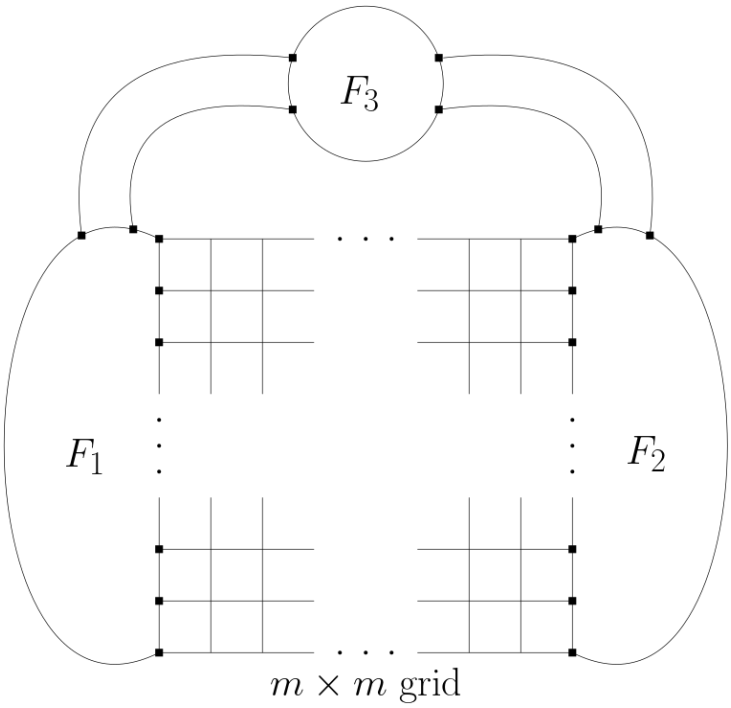


Figure 3.11: The DPP instance created in the proof of Theorem 3.5 with three terminal faces.

The algorithm is in XP, so it runs in polynomial time if the number of terminal faces is fixed. The result is also extended to work on trees rather than paths (and thus solves the Disjoint Trees problem).

3.5.4 Applying irrelevant vertices to the few faces variant

The existence of an XP algorithm parameterized by the face cover number is nice, but it does still beg the question of whether or not the problem is also in FPT which would be preferable over the XP algorithm. One potential approach to this, in light of earlier Disjoint Paths algorithms, would be to apply the irrelevant vertices technique. This section shows that this is actually not possible. The existence of an FPT algorithm for this problem remains an open question, but if it exists it should probably use a different approach than irrelevant vertices.

Recall that the irrelevant vertices technique requires a proof of the form "there exists a function $g : \mathbb{N} \to \mathbb{N}$ such that if G has a treewidth of $g(k)$ or greater, then it is possible to identify an irrelevant vertex in G". This is the part of the technique that fails for the parameterization by the face cover number, which is proven in a manner similar to the proof by Adler et al. for the tightness of the $2^{\Omega(k)}$ lower bound on planar graphs when parameterized by the number of terminals. [1] By showing that for every k and w, it is possible to construct a DPP instance, with terminals on k faces and treewidth at least w, which contains no irrelevant vertices, it is effectively proven that such a function does not exist. In turn, this means the irrelevant vertices technique cannot be applied in its current form.

Theorem 3.5. *For every $k \in \mathbb{N}$ and every function $g : \mathbb{N} \to \mathbb{N}$, it is possible to construct a* Planar Disjoint Paths *instance $(G, \mathcal{P})$ that satisfies the following properties:*

1. *the terminals of $\mathcal{P}$ can be covered by at most k faces in a planar embedding of G;*
2. *G is planar and has treewidth at least $g(k)$;*
3. *the instance permits exactly one solution, which uses all vertices in P.*

Proof. For any $k \in \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, I construct a planar DPP instance with treewidth at least $w = g(k)$ and at most k terminal faces.

Consider $k = 3$ (other values will be shown to be possible with a similar construction) and let w be any positive integer. Say $m = \lceil \frac{2}{9}w \rceil$ and construct an $m \times m$ grid. Label the vertices on the grid's left column, in order, $s_1, ..., s_m$ and those on the right column $t_1, ..., t_m$ (and add these terminal pairs to the input). Continue the left column by adding terminals s_{m+1} and s_{m+2} at the bottom and add an edge between s_{m+2} and s_1 to close the cycle, which forms the face F_1. Do the same with the rightmost column of the grid, by adding terminals s_{m+3} and s_{m+4} and adding an edge between s_{m+4} and t_1 to create the face F_2.

Create a third terminal face F_3 with the terminals t_{m+1}, t_{m+2}, t_{m+3} and t_{m+4}. For each of these terminals, create an edge between it and its corresponding terminal on F_1 or F_2, ordering the terminals in a way such that these edges do not cross.

The resulting graph contains three faces that are all connected through $m+4$ terminal pairs; the first two faces are connected by an $m \times m$ grid. All vertices incident on terminal faces are terminals: the only non-terminals appear in the grid. See Figure 3.11 for an illustration.

This instance permits exactly one solution, which uses all vertices. For the terminals not incident on the grid, their respective paths consist of a single edge as these terminals are adjacent to their counterparts. For the terminals on the grid, it is obvious that each path must be a full row of the grid, meaning all vertices of the grid are used.

Because the graph contains an $m \times m$ grid, its treewidth is at least $\lfloor \frac{9}{2}m \rfloor \geq w$ (by the excluded grid theorem, Theorem 2.2). At the same time, it permits a covering of the terminals with three faces. An instance for $k > 3$ can be created by replacing F_3 by a chain of faces, each subsequent pair connected by two terminal pairs. This is not strictly necessary for the proof, though.

Thus it is proven that for any function g, there exists a DPP instance with at most k terminal faces and treewidth at least $g(k)$, that contains no irrelevant vertices. □

Chapter 4

Directed Steiner Tree

The Steiner Tree problem is a classic problem in algorithmic graph theory, which asks to find for a given set of terminal vertices, a tree of minimum weight that connects all the terminals. The problem is famously NP-complete and allows parameterized algorithms by the number of terminals and the graph's treewidth [11]. Formally, the problem is stated as follows:

Steiner Tree
Input: undirected graph G, edge weights $w : E(G) \to \mathbb{R}^+$, terminals $K \subseteq V(G)$
Asked: a minimum Steiner tree in G connecting all terminals

Some less studied variants of the Steiner tree problem work on directed graphs. One specific version, simply called Directed Steiner Tree, takes an extra root terminal, and asks that the output tree is rooted in that vertex. This means the output graph should have a path from the root to every terminal. There is not a lot of literature about this specific problem, which poses the question of to what extent it is more difficult than its undirected counterpart. The problem statement is as follows:

Directed Steiner Tree
Input: directed graph G, arc weights $w : E(G) \to \mathbb{R}^+$, terminals $K \subseteq V(G)$, root $r \in V(G)$
Asked: a minimum directed Steiner tree in G connecting all terminals, rooted in r

Moreover, an area of interest in these problems is the parameterization by the face cover number on a planar graph, similar to Disjoint Paths. In this version, the input is a plane graph, and contains a set $\mathcal{K}$ of faces such that every terminal is incident on at least one face in $\mathcal{K}$. A few algorithms parameterized by the face cover number have been published, but a result for directed graphs remains unclear.

This chapter considers two algorithms for undirected Steiner trees. The first algorithm, by Dreyfus and Wagner [13], was originally published in 1971 and remains very important to this day. With some improvements that were discovered later on, it is still the best known Steiner Tree algorithm parameterized by the number of terminals. Adapting it to the face cover variant gives a great improvement in running time as well. The first few sections of this chapter are dedicated to describing the algorithm as well as its improvements, and how to apply it to the few-faces variant; then Section 4.4 shows how to apply the algorithm to the Directed Steiner Tree problem.

The second considered algorithm was created by Kisfaludi-Bak et al. in 2018 [23], much more recently, and specifically targets the variant parameterized by the face cover number, giving a better result than Dreyfus-Wagner. Section 4.5 describes the original algorithm, and Section 4.6 applies it to the directed variant.

4.1 The Dreyfus-Wagner algorithm

Probably the most well-known algorithm for the undirected Steiner Tree problem is that by Dreyfus and Wagner [13]. It is a simple dynamic programming algorithm that recursively builds solutions for subsets of the instance's terminals. Essential to this thesis is that the algorithm works on the directed variant as well (as shown in Section 4.4), and can be adapted well for the planar case with terminals on few faces for an improved running time (as shown in Section 4.3). The algorithm runs in FPT time parameterized by the number of terminals, meaning that if the number of terminals is fixed, it is polynomial.

4.1.1 Anatomy of a Steiner tree

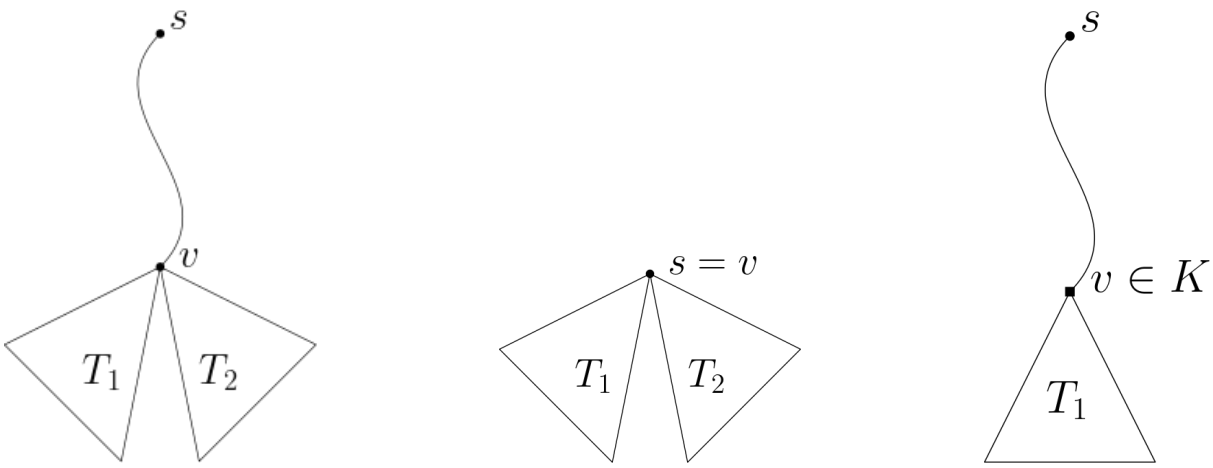
The basic idea of the algorithm is as follows. Suppose S is a minimum Steiner tree on the terminal set K, and s is some vertex that is part of S (terminal or not). If s has degree one, follow the path from s to the first vertex that either has degree greater than two or is a terminal and call this vertex v (if s has degree greater than one, say $v = s$). Let P be this path from s to v, let T_1 be one of the other components connected to v and let T_2 be the rest of the graph connected to v, noting that T_2 consists of only the vertex v and no edges if v is a terminal with degree less than three. Three illustrations of this structure are shown in Figure 4.1.

Observation 4.1. *The following two statements hold with regards to the described Steiner tree structure:*

1. *P is a shortest path from s to v.*

2. *Let $K_1 = K \cap V(T_1)$ and $K_2 = K \cap V(T_2)$. Now T_1 is a minimum Steiner tree connecting the terminals $K_1 \cup \{v\}$ and T_2 is a minimum Steiner tree connecting the terminals $K_2 \cup \{v\}$.*

The proof of these observations follows simply from the fact that S is a minimum Steiner tree comprised of only the components P, T_1 and T_2. If there existed any lower-weight alternatives for these components that still had the same endpoints or terminals, S would no longer be minimum.



(a) $s \neq v$ (b) $s = v$ (c) v is a terminal

Figure 4.1: Three examples of the Steiner tree structure as described. The large triangles denote subtrees containing terminals, and the curved segment a shortest path. In 4.1c, T_2 consists of only v.

4.1.2 Recurrence

From this, a dynamic programming formulation of minimum Steiner trees follows. Suppose that one wants to compute a minimum Steiner tree on the terminal set $D \cup \{s\}$ with $D \subseteq K$ and $s \in V(G)$. Furthermore, suppose one knows the weight of minimum Steiner trees with terminals $D' \cup \{s'\}$ for $D' \subset D$ and $s' \in V(G)$. Now it suffices to guess an intermediate vertex v and a splitting of the terminals between the two subtrees to compute the minimum Steiner tree for $D \cup \{s\}$. The recurrence ends up as follows:

$$B[D, s] = \min_{\substack{v \in V(G) \\ \emptyset \subset D' \subset D}} \left\{ \mathrm{dist}(s, v) + B[D', v] + B[D \setminus D', v] \right\} \tag{4.1}$$

where $B[D, s]$ is the weight of a minimum Steiner tree on terminals $D \cup \{s\}$, and $\mathrm{dist}(s, v)$ is the distance between vertices s and v as computed by a shortest path algorithm. Since the values $B[D', v]$ for any $D' \subset D$ and any vertex $v \in V(G)$ were computed in advance, this recurrence is able to compute a minimum Steiner tree properly by iteratively applying it to larger subsets of the terminal set.

Lemma 4.1. *The recurrence from Equation 4.1 computes minimum Steiner trees correctly.*

Proof. Recall the Steiner tree structure described in Section 4.1.1. Supposing one is computing the value $B[D, s]$, let S be a minimum Steiner tree on terminals $D \cup \{s\}$. Using s as the "starting" vertex, define v, P, T_1 and T_2 as described. The aim is to prove that $B[D, s] = w(S)$.

To prove that $B[D, s] \geq w(S)$, note that every selection of D' and v corresponds to some Steiner tree in the graph that represents all terminals (although the computed weight may actually be greater than that of this tree, since edges can be counted doubly). This is shown easily by the fact that s is connected to the path represented by $\mathrm{dist}(s, v)$, and every terminal is present in one of the subtrees represented by $B[D', v]$ or $B[D \setminus D', v]$. Furthermore, each of these components is connected to v, meaning together they are all connected. This means that every considered structure is at least as large as some Steiner tree on the terminals, thus the final answer must be at least as large as the weight of a minimum Steiner tree.

To prove that $B[D, s] \leq w(S)$, consider that the recurrence at one point selects v as the vertex v in the structure of S, and D' as $D \cap V(T_1)$, which represents the total structure of S. Now it must be that $w(P) = \mathrm{dist}(s, v)$ since P is a shortest path, and that $B[D', v] \leq w(T_1)$ and $B[D \setminus D', v] \leq w(T_2)$, since otherwise it would be possible to substitute one of the computed subtrees with its corresponding part in S, which would yield a smaller minimum Steiner tree. It follows that the recurrence must compute a Steiner tree that is no larger than S. □

4.1.3 Algorithm

With the recurrence from Equation 4.1 proven, it is possible to formulate the dynamic programming algorithm. First, we split the recurrence up into two parts:

$$A[D, v] = \min_{\emptyset \subset D' \subset D} \left\{ B[D', v] + B[D \setminus D', v] \right\}; \tag{4.2}$$

$$B[D, s] = \min_{v \in V(G)} \left\{ \mathrm{dist}(s, v) + A[D, v] \right\}. \tag{4.3}$$

Note that this formulation is equivalent to the previous one. This change is not strictly necessary, but it does yield a better running time for the algorithm. Also note that the formulation of A only works for sets D with $|D| \geq 2$, as it is otherwise impossible to split it into two non-empty subsets. This means the algorithm needs a base case for $|D| = 1$, which amounts to just taking the shortest path between the two given terminals:

$$B[\{u\}, v] = \text{dist}(v, u). \tag{4.4}$$

With this, a simple dynamic programming algorithm is defined. First, it computes the values of B for singleton sets D. Then, for increasingly larger sets, it computes all values of A and B as they only require the answers of smaller sets. Lastly, it suffices to return the value $B[K, t]$ for some terminal $t \in K$, which corresponds to the weight of a minimum Steiner tree on all vertices. A pseudocode for this algorithm is given in Algorithm 4.1.

Algorithm 4.1 The Dreyfus-Wagner algorithm for computing minimum Steiner trees

Input: undirected graph G, edge weights $w : E(G) \to \mathbb{R}^+$ and terminals $K \subseteq V(G)$
Output: the weight of a minimum Steiner tree in G on terminals K

```
Compute the distances between all pairs of vertices using a shortest paths algorithm
for u, v ∈ V(G) do
    B[{u}, v] ← dist(v, u)
end for
for i ∈ {2, ..., |K|} do
    for D ⊆ K with |D| = i do
        for s ∈ V(G) do
            A[D, s] ← min_{∅⊂D'⊂D} {B[D', s] + B[D \ D', s]}
            B[D, s] ← min_{v∈V(G)} {dist(s, v) + A[D, v]}
        end for
    end for
end for
return B[K, t] for any t ∈ K
```

The correctness of this algorithm follows from Lemma 4.1, and Equation 4.4 giving the answer for the base case trivially.

The algorithm returns the weight of a minimum Steiner tree, but it is not difficult to modify it to return the tree itself. For every value of A and B, pointers are stored towards the values of A, B and shortest paths that were used to obtain their result. In the end, when the whole dynamic programming table has been completed, these pointers can be followed again and the shortest paths that make up the Steiner tree can be put together to form the tree itself.

4.1.4 Analysis

The running time of Dreyfus-Wagner is evaluated in three steps. Firstly, computing all-pairs shortest paths can be done in $O(n^2 \log n + nm)$ time using Johnson's algorithm [19]. Next, note that there are $2^k n$ values to be computed for A and B each. Computing a value $A[D, s]$ takes $O(2^{|D|})$ time and computing a value for B takes $O(n)$ time. This means computing all values for A takes $O(3^k n)$ time and all values for B takes $O(2^k n^2)$ time. Adding these times together gives $O(n3^k + n^2(2^k + \log n) + nm)$ time. Furthermore, the algorithm takes $O(2^k n + n^2)$ space to store the intermediate answers and the shortest paths matrix. Note that the algorithm is polynomial if the value of k is fixed.

4.2 Improvements to Dreyfus-Wagner

While Dreyfus-Wagner remains the "gold standard" of computing Steiner trees, a number of improvements have been made to the algorithm since it was first published. The most significant improvement is through fast subset convolution, described in Section 4.2.2, which does not change the dynamic programming recurrence in any way but is a way to perform the computation more efficiently. An improvement by Erickson et al., described in Section 4.2.1, seeks to mitigate

the computation of all-pairs shortest paths by integrating it with the recurrence; classic Dreyfus-Wagner using these two improvements appears to be the most efficient Steiner Tree algorithm to date. Lastly, work by Mölle et al. summarized in Section 4.2.3 abandons the original algorithm, and uses its ideas to create a more efficient variant that splits on multiple vertices rather than a single one.

4.2.1 Erickson's batching technique

The improvement by Erickson et al. [5,14] works as follows. The pre-processing step of computing all-pairs shortest paths is skipped, to be performed "on the fly" while computing the dynamic programming entries. Now, for some terminal set D, instead of computing all values $B[D, s]$ for individual values of s separately, these values are all computed together as a single-source shortest path computation. Take a copy of the input graph, and make it directed by replacing each edge with a pair of parallel arcs in opposite directions. Add a (virtual) vertex x to the graph, and from x to every other vertex v, add an arc of weight $A[D, v]$ and run a shortest paths algorithm with x as its source. Now for every vertex s in the graph, the distance from x to s equals $\min_{v \in V(G)} \{A[D, v] + \text{dist}(v, s)\}$, which equals the formula for B in Equation 4.3. This means that instead of taking $O(n^2)$ time to compute $B[D, s]$ for every s separately, the values are computed using a single run of a shortest paths algorithm. For the base case, it suffices to run a shortest paths algorithm using the single terminal as source, in order to compute all values of B corresponding to it.

By computing shortest paths using Dijkstra's algorithm with Fibonacci heaps [17], this improves the $O(n^2)$ dependency to $O(n \log n + m)$ and eliminates the pre-processing step. The total running time then becomes $O(n3^k + (n \log n + m)2^k)$ and the algorithm takes only $O(n2^k)$ space as there is no longer need to store a shortest paths matrix.

4.2.2 Fast subset convolution

Fast subset convolution (FSC) is a relatively recent technique by Björklund et al. [7] that aims to improve the complexity of hard problems that involve convolution on subsets of a large set as an integral step. Specifically, problems that contain a computation similar to

$$h(S) = \sum_{T \subseteq S} f(T) g(S \setminus T) \tag{4.5}$$

for each $S \subseteq N$, can generally find improvement using fast subset convolution. Normally, to compute all values of h would take $\Omega(3^n)$ time (with n being the size of N), as there are 2^n choices of S and $2^{|S|}$ choices of T. Using FSC however, this running time can be shaved down to 2^n times a polynomial factor dependent on the structure of the values that are worked with. Conveniently, the original paper introducing FSC applies the technique to the Dreyfus-Wagner algorithm; this section will summarize its findings.

The above example works with the integer sum-product ring, which means that the results of the functions f, g and h are simple integers subject to addition and multiplication. It is shown that FSC can be performed in $O(2^n n^2)$ operations on any arbitrary ring, and in roughly $O(2^n \log M)$ time for the integer sum-product ring with integers ranging from $-M$ to M.

In the case of Dreyfus-Wagner it is more complicated, as the central computation is not based on products and sums but rather on sums and minimum values. Note that substituting a min-operation for the summation and addition for multiplication in Equation 4.5 makes it closely resemble the recurrence for A used in Dreyfus-Wagner (Equation 4.2). The problem is that this formulation no longer works with a ring, but with a semiring instead (referred to as the integer min-sum semiring). The difference between rings and semirings is that semirings do not require an additive inverse. In the case of integer min-sum, an additive inverse would mean that there must be, for every value a, some value b such that $\min\{a, b\} = 0$. Obviously, such a

value b does not exist for any negative integer, so an additive inverse is impossible to define for the integer min-sum semiring. Fortunately, the researchers show a way to make FSC work on this specific semiring, in roughly $O(2^n M)$ operations if the integers are bounded between $-M$ and M.

Throughout the section, the notation for sum-product rings is used. At the end, the adaptation to min-sum semirings is briefly summarized along with how to apply this technique to Dreyfus-Wagner.

Möbius transform and inversion

For a function $f : 2^N \to \mathbb{Z}$, the Möbius transform $\hat{f}$ is defined as follows:

$$\hat{f}(X) = \sum_{S \subseteq X} f(S). \tag{4.6}$$

When the result of the Möbius transform is known for all subsets of the domain, it can be inverted just as easily using the inclusion-exclusion principle:

$$f(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} \hat{f}(X). \tag{4.7}$$

Naïvely computing the Möbius transform and its inverse for all subsets of N will take $\Omega(3^n)$ time, which is obviously undesirable. However, there exists a simple algorithm (called *fast Möbius transform*) to compute both in $O(n2^n)$ time, by iterating over all individual elements and adding together values for subsets that do and do not contain said element.

Convolution operations

For the purposes of FSC, it is possible to formulate several similar variants of the convolution operation. The "regular" and most common variant is the one shown before, which splits a set into two disjoint subsets:

$$(f * c)(S) = \sum_{T \subseteq S} f(T) g(S \setminus T). \tag{4.8}$$

Another variant is the "covering product", which allows the two subsets of S to overlap:

$$(f *_c g)(S) = \sum_{\substack{U,V \subseteq S \\ U \cup V = S}} f(U) g(V). \tag{4.9}$$

The goal of FSC is to define these convolution operators on the Möbius transforms of the input functions, in such a way that applying inverse Möbius transform preserves the output. That is, one wants to formulate a function $\hat{f} \circledast \hat{g}$, that equates to $f * g$ when the inverse Möbius transform is applied to it. Then the FSC algorithm consists of simply computing the functions $\hat{f}$ and $\hat{g}$ using fast Möbius transform, computing $\hat{f} \circledast \hat{g}$ and then applying inverse fast Möbius transform to it to obtain the function $f * g$.

Covering product

Computing the covering product $\hat{f} \circledast_c \hat{g}$ on Möbius transformed functions is done by simply taking the product of the two transformed input functions:

$$(\hat{f} \circledast_c \hat{g})(X) = \hat{f}(X) \hat{g}(X). \tag{4.10}$$

The correctness of this can be proven by simply equating it to the Möbius transform of the naïve covering product:

$$\begin{aligned}
(\hat{f} \circledast_c \hat{g})(X) &= \hat{f}(X)\hat{g}(X) \\
&= \left(\sum_{U \subseteq X} f(U)\right)\left(\sum_{V \subseteq X} g(V)\right) \\
&= \sum_{U,V \subseteq X} f(U)g(V) \\
&= \sum_{S \subseteq X} \sum_{\substack{U,V \subseteq S \\ U \cup V = S}} f(U)g(V) \\
&= \sum_{S \subseteq X} (f *_c g)(S).
\end{aligned}$$

Since the formulation of $\hat{f} \circledast_c \hat{g}$ equals the Möbius transform of the covering product $f *_c g$, it is possible to apply the inverse Möbius transform to it to obtain $f *_c g$.

Subset convolution

Applying the same principle to regular convolution is a lot harder. Define the *ranked* Möbius transform on a function f as the following:

$$\hat{f}(k, X) = \sum_{\substack{S \subseteq X \\ |S| = k}} f(S) \tag{4.11}$$

that is, it only counts the subsets of a specific cardinality. The inversion of ranked Möbius transform is a lot simpler, as it is just $f(S) = \hat{f}(|S|, S)$. For computing the ranked transformation, there also exists a fast algorithm similar to the non-ranked version.

Convolution on transformed functions is defined as follows:

$$(\hat{f} \circledast \hat{g})(k, X) = \sum_{j=0}^{k} \hat{f}(j, X)\hat{g}(k - j, X). \tag{4.12}$$

Proving the correctness of this formulation is done in a similar fashion to the covering product, but turns out more difficult[1]. An intuition for understanding the formulation, is that fixing the ranks of the observed subsets ensures that only the given set X "contains" the desired results in its Möbius transform. For instance, suppose that $|X| = 7$ and one is looking at subsets U and V of X with $|U| = 4$ and $|V| = 3$. Now if $U \cap V \neq \emptyset$, then their union is not only a subset of X but also the subset of some subset(s) of X. Whereas if $U \cap V = \emptyset$, it must be that $U \cup V = X$ due to the fixed cardinalities and it is not a subset of any of X's subsets.

From sum-product to min-sum

The adaptation to the integer min-sum semiring (and equivalently max-sum) is done as follows. Assume without loss of generality that the functions operate on the integer range $\{0, 1, ..., M\}$. Define $\beta = 2^{n+1}$, and for every function on the domain f define the function f' as $f'(x) = \beta^{f(x)}$. Now observe that convolution on these functions works as follows:

[1] I included the proof for the covering product as an easy substitute for the purposes of this review. The proof for regular convolution can be found in Section 2.4 of [7].

$$(f' * g')(S) = \sum_{T \subseteq S} f'(T)g'(S \setminus T) = \sum_{T \subseteq S} \beta^{f(T)+g(S \setminus T)} = \sum_{i=0}^{2M} \alpha_i(S)\beta^i \tag{4.13}$$

which is a polynomial on β with coefficients $\alpha_i(S)$. Note now that, due to the choice of β, the coefficient $\alpha_i(S)$ denotes the number of times the value i has appeared as a result in the summation. So picking the minimum or maximum value is done by simply taking the least or greatest value i such that $\alpha_i(S) > 0$ in the polynomial. The rest of the evaluation is unaffected by fast subset convolution.

Applying to Dreyfus-Wagner

Applying FSC to Dreyfus-Wagner should be relatively straightforward at this point. The sole difficulty lies in the fact that it is based on a recurrence (the functions used in the convolution are self-referential), but this has an easy fix. Remember that the algorithm computes values level-wise, with the i-th level computing results for terminal sets of size i. Define the function

$$f_s^i(D) = \begin{cases} B[D, s] & \text{if } |D| < i \\ \infty & \text{otherwise.} \end{cases} \tag{4.14}$$

Now, when using convolution on the integer min-sum semiring, it is obvious that for terminal sets D with $|D| = i$, the following holds:

$$A[D, v] = (f_v^{i-1} * f_v^{i-1})(D). \tag{4.15}$$

From this, it suffices to perform fast subset convolution at each level of the algorithm using the above equation. If M is the greatest edge weight in the graph, then nM is enough as an integer bound as no Steiner tree would exceed this weight. Performing FSC on each of the k levels for each vertex v to compute the values of A takes about $O(2^k k^3 n)$ time. Using the covering product can reduce the factor k^3 to k^2: rather ironically, it would now consider a lot of redundant cases where both subtrees share some terminals in common while doing it all in less time.

Incorporating this in the original analysis gives a running time of $O(n^2k^2(2^k + \log n) + nm)$ time. Erickson's batching technique is compatible with this, as it only changes the computation of B. So applying FSC with the covering product, as well as the batching technique, gives an algorithm that runs in $O(2^k k^2 n^2)$ time.

4.2.3 Splitting in multiple vertices

An algorithm by Mölle et al. expands on the ideas of Dreyfus-Wagner to give a $O((2+\epsilon)^k \cdot n^{f(\epsilon)})$ algorithm for some constant ϵ [33]. It does this, by considering multiple splitting vertices rather than the one intermediate vertex chosen by Dreyfus-Wagner. For a (rooted) tree with $k \geq 1/\epsilon^2$ terminals, the authors show that there exists a set X of at most $\lceil 1/\epsilon \rceil$ vertices, such that removing X from the tree gives subtrees containing at most ϵk terminals each. This leads to a formulation similar to Dreyfus-Wagner albeit much more complicated, using a similar dynamic programming technique to compute minimum Steiner trees. The difficulty lies in the fact that while Dreyfus-Wagner need only treat one non-terminal vertex as a terminal in each subproblem, this algorithm has to store multiple vertices in a similar way. Furthermore, there is not an exact optimal value for ϵ; lower values will decrease the exponential base of the running time, but greatly increase the exponent of the polynomial part. Nevertheless, the algorithm does give an improvement over basic Dreyfus-Wagner without fast subset convolution.

4.3 Dreyfus-Wagner on planar graphs with terminals on few faces

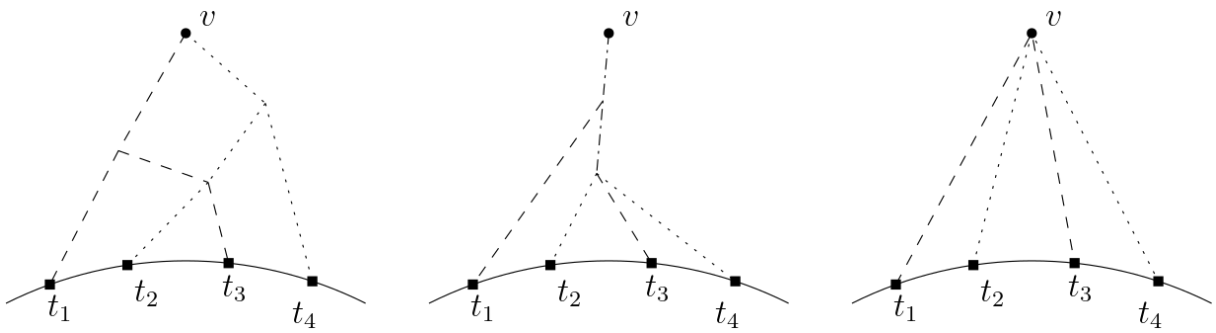
When looking at planar graphs with a low face cover number, significant improvements can be made to Dreyfus-Wagner. The improvement lies in the fact that most choices for D and D' become redundant if many terminals in D lie on the same face [14].

Lemma 4.2. *Suppose the value $A[D, v]$ (Equation 4.2) is computed on a planar graph for some set of terminals $D = \{t_1, t_2, t_3, t_4\}$ which appear in cyclic order along the boundary of a face F. Moreover, suppose in the min-operation of the equation D' is considered to be $\{t_1, t_3\}$ (so $D \setminus D' = \{t_2, t_4\}$). Now the considered value $(B[D', v] + B[D \setminus D', v])$ is either inherently suboptimal, or equal to that of a different choice of D'.*

Proof. Suppose T_1 is a minimum Steiner tree connecting $D' \cup \{v\} = \{t_1, t_3, v\}$ and T_2 is a minimum Steiner tree connecting $(D \setminus D') \cup \{v\} = \{t_2, t_4, v\}$. Note that either t_2 or t_4 is enclosed between T_1 and F: a path between t_2 and t_4 (which should be part of T_2) would thus have to cross T_1. One of the following three situations has to be the case:

1. the union of T_1 and T_2 is not a tree, and therefore also not minimum;
2. T_1 and T_2 have some edges in common, meaning their added weights are greater than the weight of the actual tree;
3. T_1 and T_2 only have the "root" vertex v in common.

In case 1, the solution is trivially suboptimal due to the result not being a tree. In case 2, the exact same tree is considered with another choice of D' for a different vertex v, but without any overlapping edges. In case 3, the same result is obtained by having $D' = \{t_1, t_2\}$ and $D \setminus D' = \{t_3, t_4\}$. Therefore, the case with $D' = \{t_1, t_3\}$ is not needed to find the optimal result. □



(a) case 1 (b) case 2 (c) case 3

Figure 4.2: Three cases of why the terminals cannot be "intersecting". The dashed edges correspond to the tree T_1 and the dotted edges to T_2.

The consequence of this is that one can limit oneself to cases where on each face with terminals, D''s (and $D \setminus D'$'s) terminals appear in an interval not containing any other terminals. That is, one can walk along the boundary of a face from one vertex to another, only hitting those terminals from D' and not hitting any other terminals. Since the equation now only uses earlier values on terminal-intervals, the same principle applies for the choices of D.

The complexity improvement is significant, especially if the face cover number is low. Let f be the face cover number, and let $k_1, ..., k_f$ be the number of terminals incident on each of

these faces (so $\sum_{i=1}^{f} k_i \geq k$). Now with respect to some face i, there are only $O(k_i^2)$ choices for terminals in D. Hence there are at most $O(k^{2f})$ choices for D in total. The choices for D' are similar, but less if D does not contain all terminals incident on some face.

Adding the computation of all-pairs shortest paths, this puts the total running time at $n^2(k^{O(f)} + \log n) + nm$ (or $n^{O(f)}$ assuming $k = O(n)$). If the value f is fixed, this running time is polynomial; this complexity is in the class XP. The space usage also improves to $k^{O(f)}n + n^2$ or $n^{O(f)}$, as only values for intervals D need to be stored rather than all subsets.

4.3.1 Improvements

For this variant of the problem, it is also of interest to look for extra improvements. The first two improvements described in Section 4.2 are considered. Erickson's batching technique appears to be no different on this problem variant, but there is no apparent way to apply fast subset convolution.

Erickson's batching technique

Applying the batching technique is still possible in the few-faces variant, and requires no changes. Since the only difference in the algorithm is in how terminal sets are selected, and the batching technique changes the computation of the values B, which does not involve selecting terminals, the exact same principle can be applied here. The running time then becomes $(n \log n + m)k^{O(f)}$ and the n^2 part is eliminated from the space usage.

Fast subset convolution

Applying fast subset convolution to the few-faces variant proves more difficult, and potentially impossible. The problem lies in the fact that regular convolution operates on subsets of some universe, whereas in the case of the few-faces Dreyfus-Wagner algorithm, we are working with intervals of sequences. An attempt to apply FSC to this case would involve formalizing the notion of intervals in the form of a "sub-interval" lattice, and applying the same principles as FSC to this lattice.

Let $N = \{1, ..., n\}$ be the universe for intervals, and let the interval space I consist of pairs $(i, j) \in N \times N$ with $i < j$ as well as the empty interval $\emptyset$. A non-empty interval (i, j) consists of the values k with $i \leq k < j$. It is straightforward to define a subinterval relation as well as an intersection operator on such intervals, but it is impossible to sensibly define the union of two intervals. Consider the intervals $a = (1, 4)$ and $b = (7, 9)$. Any "union" of a and b would also contain values outside the two intervals, whereas a proper union should only include the values 1, 2, 3, 7 and 8 which is obviously not an interval. This rules out a reformulation of the covering product operation, as it takes two subsets whose union covers the whole set.

In the regular convolution definition, unions are not used. Recall the definition $(f * c)(S) = \sum_{T \subseteq S} f(T)g(S \setminus T)$, which only contains the subset relation and the set subtraction operation, both of which can be translated to intervals. It is therefore actually possible to apply this form of FSC to intervals, but it does not appear to give an improvement in running time.

Consider the problem variant with all terminals on a single face, which is the easiest for illustration purposes as it works with intervals of just one sequence, rather than multiple stitched-together sequences. The running time using the algorithm described previously would be $O(k^3n^2)$, with FSC giving the chance of reducing the k^3 factor. Suppose the ranked Möbius transforms $\hat{f}(k, X)$ on all X can be computed in $o(k^3)$ time, as well as their inversions. To apply FSC, all that remains is to compute the convolutions of the Möbius-transformed function. The formula for this would be the exact same as for regular sets:

$$(\hat{f} \circledast \hat{g})(m, X) = \sum_{j=0}^{m} \hat{f}(j, X)\hat{g}(m - j, X).$$

The problem is that there does not appear to be a way to compute these values in $o(k^3)$ time, as there are already $\Theta(k^3)$ values to compute on their own due to the formulation working with "ranked" Möbius transforms. For regular sets, this is not an issue because computing the $\Theta(k2^k)$ values is still an improvement over the naïve running time of $\Omega(3^k)$, but for intervals the linear factor of the ranks is unaffordable.

While this is not conclusive proof of FSC being entirely inapplicable to intervals, it seems to be an indication that either it is impossible or it would require a significantly different approach from the original version.

4.4 Dreyfus-Wagner on directed graphs

The Dreyfus-Wagner algorithm can easily be adapted to the directed Steiner Tree problem. The main difference is now that the vertex s in $B[D, s]$ indicates the root vertex of the represented Steiner tree, rather than some extra terminal appearing in the tree. The recurrences actually stay the same, provided that the shortest paths are always taken starting in v. The final answer for K and the given root vertex r is now $B[K, r]$.

In the undirected Steiner Tree problem, it is assumed that G is connected (or that all terminals are in the same connected component of G, in which case all other components can be ignored). While it can be similarly assumed in the directed variant that all vertices are reachable from the root vertex, it is still possible that there are pairs of vertices with no path between them in one or both directions. In this case, the distance between the two vertices should be counted as ∞, meaning there is no path. When $B[D, s] = \infty$, it then means there is no Steiner tree for terminals D with v as its root.

4.4.1 Directed planar Dreyfus-Wagner with terminals on few faces

Adding arc directions to the graph in the few-faces variant of Dreyfus-Wagner does not substantially change anything to the algorithm. The structural arguments for terminals appearing in intervals still hold just the same when arcs have directions, and the complexity of enumerating intervals rather than subsets does not change.

4.4.2 Improvements

Again, we can look at improvements to the algorithm in the directed case. For the batching technique, only a minor change is needed in the computation of shortest paths. For fast subset convolution, there is no difference.

Erickson's batching technique. The difference in applying the batching technique, is that all the arc directions need to be flipped during the shortest path computation. Recall that the computation of B involves a shortest path starting in s and ending in v, whereas the shortest path computation in the batching technique starts in the virtual vertex x, goes to v and then ends in s. So in order to properly take the $s - v$ shortest path, all arc directions need to be flipped for the batching technique.

Fast subset convolution. Since the recurrence relations of Dreyfus-Wagner do not change at all in the directed case, fast subset convolution does not require any change and works all the same unless, of course, it is applied to the few-faces variant.

4.5 Algorithm by Kisfaludi-Bak et al.

A recent algorithm by Kisfaludi-Bak et al. aims to improve Dreyfus-Wagner's performance on planar graphs with low face cover [23]. It is a recursive algorithm that guesses a small set of "separating" vertices in the Steiner tree and recursively builds the rest of the tree around it. Using properties of the resulting Steiner tree, it is possible to prove that a sufficient separator exists of limited size such that the terminal faces can be distributed reasonably evenly over the two subproblems.

This section contains only a high-level overview of the algorithm, to sketch the motivations behind it. Section 4.6 applies the same algorithm to directed graphs and goes into detail proving its correctness and running time.

4.5.1 Motivation

An observation leading to the algorithm, briefly noted in the original paper, is that removing an edge e from a Steiner tree S splits the tree into two smaller Steiner trees. If one knows for all terminals in which subtree they are connected, one can build up the original Steiner tree by solving the two sub-instances by treating each of e's endpoints as a terminal in a subtree. This leads to a recursive algorithm, branching $|E| \cdot 2^{|K|}$ times in each step (choosing one edge and one way to split the terminals into two subsets).

This approach works, but is not very efficient due to its high recursion depth. Ideally, the separation of the terminals would be reasonably balanced, but when splitting along a single edge, it is possible that one terminal has to be separated from all the others, leading to a recursion depth of k (Figure 4.3 shows an example of this). A way to mitigate this is to split along a single vertex rather than an edge: now the Steiner tree separates into several subtrees, but by Theorem 2.1 it can be shown that these subtrees can be split into two groups where each group has at most 2/3 of all terminals. When we only have to look at 2/3-balanced separations of the terminals, the recursion depth becomes $O(\log k)$ which is a lot more feasible. Incidentally, this approach is similar to Dreyfus-Wagner, which also guesses a single vertex to unite two subtrees.

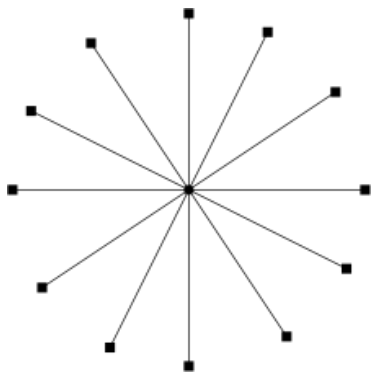


Figure 4.3: An edge-branching algorithm would be inefficient on this star graph, because it would retain all but one terminals in one of the subproblems at each step.

To apply this approach to the problem variant parameterized by the face cover is not trivial. It would be ideal if the terminal faces could be separated in a way similar to the approach above, which would mean each face could have arbitrarily many terminals on its boundary while only the face cover number matters for the complexity. However, there are instances in which it is impossible to split the Steiner tree along a single vertex such that every terminal face has terminals in exactly one subtree, as illustrated by the example in Figure 4.4. In the example, the only way to separate the terminal faces entirely and in a balanced way, is to use two vertices (v and w) as separator. If the Steiner tree is separated along u only, both subtrees contain terminals from every face. In other examples, it is even possible that the terminal faces themselves are

part of the separator, meaning some of the terminals on them end up in one subproblem and others in the other. As a further consequence, the Steiner tree is no longer separated into two smaller Steiner trees: one of the halves consists of two disjoint trees.

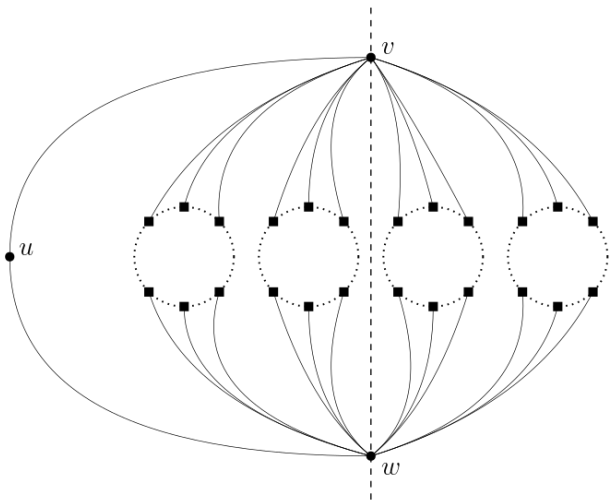


Figure 4.4: Separating this graph along a single vertex is not enough for each subproblem to end up with two full terminal faces. The dotted circles denote the terminal faces. The dashed line visualizes the separation along v and w.

The authors manage this by proving that a $O(\sqrt{|\mathcal{K}|})$ bound on the size of the separator suffices (where $|\mathcal{K}|$ is the face cover number) and reformulating the problem to work with forests rather than trees. This complicates the algorithm by a lot, but still improves on the result of Dreyfus-Wagner in the few-faces variant.

4.5.2 Separating the terminal faces

To show that a separator of size $O(\sqrt{|\mathcal{K}|})$ exists in the tree, the researchers use a bidimensionality proof combined with Theorem 2.1 which connects treewidth with balanced separations.

The existence of this separation means, basically, that there exists a set of $O(\sqrt{|\mathcal{K}|})$ vertices in the Steiner tree, that can separate the tree into two parts such that each part contains at most 2/3 of the terminal faces; in other words, it is possible to split the terminal faces in a reasonably balanced way by removing only $O(\sqrt{|\mathcal{K}|})$ vertices. Note that a terminal face can intersect the separator (so one of the chosen vertices is on a terminal face border) meaning it is represented in both sides of the separation. Since the size of the separator is $o(|\mathcal{K}|)$, this does not mean the split is no longer necessarily balanced, but it does complicate both the proof and the algorithm so as to account for faces that are only partially represented in each subproblem.

4.5.3 Steiner Forests

As briefly mentioned earlier, the adaptation to use separators of more than one vertex does mean that the trees are no longer neatly split into subtrees, and that the algorithm should instead work with forests. This poses extra challenges, because the structure of forests needs to be encoded in a way that ensures the algorithm returns forests that, when joined together again, form a suitable tree.

The structure used in this algorithm, called a "block Steiner forest", consists of multiple connected components (called "blocks") which each must contain a specific set of "boundary vertices" and any amount of terminals. Specifically, for a terminal set K and a family of sets of boundary vertices $\{B_1, ..., B_p\}$, a suitable block Steiner forest is a forest consisting of p

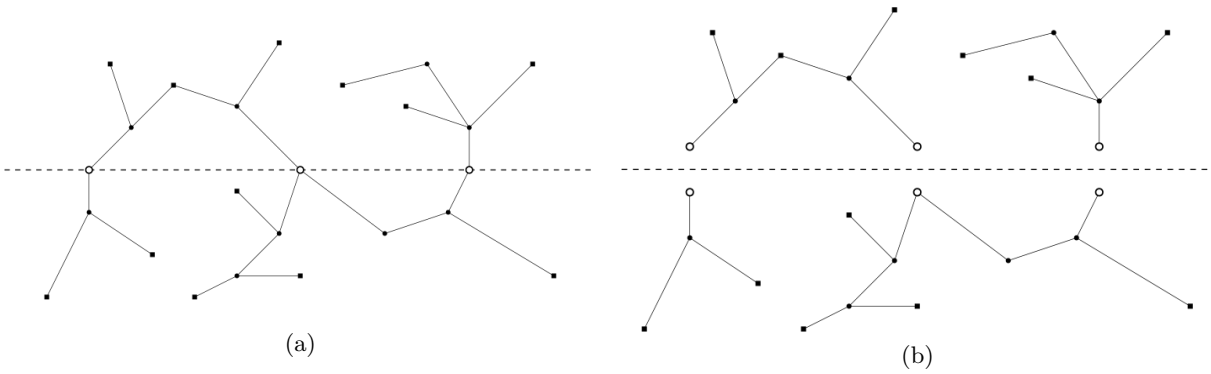


Figure 4.5: Cutting down a tree yields two forests. The circled vertices form the separator, and are boundary vertices in the subproblems.

connected components C_1 through C_p, such that $B_i \subseteq V(C_i)$ for every i, and every terminal in K is connected in one of these components.

The idea behind this, is that encoding the connectivity of certain vertices can ensure that the result, when putting together forests, is also a suitable forest or Steiner tree for the desired outcome. The given boundary vertices are all part of the selected separator(s) in the graph (as described in Section 4.5.2), because these separator vertices have to be part of the outcome while not necessarily being terminals themselves.

As an example of how this connectivity matters, consider the case where the chosen separator consists of the vertices $\{b_1, b_2\}$. The problem is then split into two subproblems, each of which will return a block Steiner forest. Both subproblems will have b_1 and b_2 as boundary vertices, but the connection between them has to exist in exactly one of the subproblems or the result will not be a forest itself. So between the two subproblems, one has to have blocks $\{\{b_1, b_2\}\}$ and the other $\{\{b_1\}, \{b_2\}\}$. This example is illustrated in Figure 4.6.

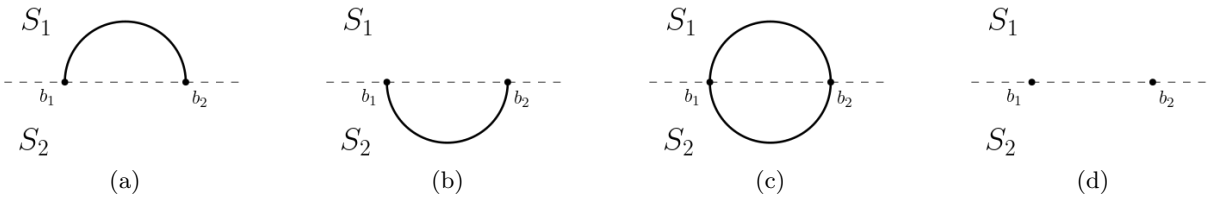


Figure 4.6: To form a tree correctly, b_1 and b_2 need to be connected in either S_1 or S_2 (a and b) but not both (c) or neither (d).

4.5.4 Algorithm

The problem statement and the algorithm are as follows. Most of the algorithm consists of "guessing" parameters for subproblems; here it simply iterates over all possibilities of the given parameter.

Planar Block Steiner Forest on Few Faces
Input: plane graph G, arc weights w, terminals K, face cover $\mathcal{K}$ of K, boundary vertices B and partition π of B
Asked: a minimum (G, w, K, B, π)-block Steiner forest

1. If $|\mathcal{K}| + |B|$ is less than a given constant, solve the problem in polynomial time using a fixed-parameter algorithm (see Section 4.6.2 for a directed adaptation of the base case);

2. Guess the following parameters by enumerating all options:

   (a) a separator $X \subseteq V(G)$ with $|X| \leq 15\sqrt{|\mathcal{K}|} + 2$;

   (b) a partition of B into B_1 and B_2 and of $\mathcal{K}$ into $\mathcal{K}_1$ and $\mathcal{K}_2$, such that $|B_1| + |\mathcal{K}_1| \leq \frac{2}{3}(|B|+|\mathcal{K}|)$ and $|B_2|+|\mathcal{K}_2| \leq \frac{2}{3}(|B|+|\mathcal{K}|)$. For terminal faces intersecting X, separate segments of the face, bounded by separating vertices, instead. Let K_1 and K_2 thus be the terminals represented in each part of the partition of $\mathcal{K}$.

   (c) partitions π_1 of $X \cup B_1$ and π_2 of $X \cup B_2$, such that when combined, these partitions represent the connectivity encoded by π.

3. Recurse on the inputs $(G, w, K_1, \mathcal{K}_1, B_1, \pi_1)$ and $(G, w, K_2, \mathcal{K}_2, B_2, \pi_2)$ and unite the two resulting forests;

4. Return the forest with minimum weight out of all guessed parameters.

The bulk of the algorithm's complexity is in guessing the parameters. For the separator X, there are $n^{O(\sqrt{|\mathcal{K}|})}$ options. For separating B and the terminals, there are $2^{O(|B|+|\mathcal{K}|)}$ options. For the two partitions, there are $(|B|+|X|)^{O(|B|+|X|)}$ options, as there are $p^{O(p)}$ ways to partition a set of p elements. The total running time of this algorithm is $2^{O(k)}n^{O(\sqrt{k})}$ with $k = |\mathcal{K}|$; for a more detailed analysis on the directed version, see Section 4.6.6.

Aside from the algorithm, the authors also prove a lower bound for the problem. Assuming that the Exponential Time Hypothesis[2] is true, there does not exist an $f(k)n^{o(\sqrt{k})}$ algorithm for Planar Steiner Tree parameterized by the face cover number of the terminals.

4.6 Algorithm by Kisfaludi-Bak et al. on directed graphs

This section describes a $2^{O(k)}n^{O(\sqrt{k})}$ algorithm for Directed Planar Steiner Tree with terminals on few faces, where k is the face cover number. This is an adaptation of the algorithm by Kisfaludi-Bak et al. described in Section 4.5. The algorithm follows closely along the same lines as its undirected counterpart; like the adaptation of Dreyfus-Wagner (Section 4.4), it needs only few adjustments to work on directed graphs. Where the section on the undirected algorithm did not go into details, this section does.

The main difference in the directed algorithm is in reformulating the definition of block Steiner forests, and how their connectivity is handled. Since the asked result of Directed Steiner Tree is a directed tree, with a pre-defined root, a block Steiner forest in this variant should consist of directed trees with pre-defined roots itself. The adapted formalization of block Steiner forests is defined in Section 4.6.1, and their connectivity in Section 4.6.3. The base case of the algorithm is discussed in Section 4.6.2.

A secondary difference with the undirected algorithm is in proving a bound on the size of the selected separator in the algorithm. This posed itself as a challenge in designing the algorithm, as the original proof relies on properties of tree decompositions, whereas tree decompositions or treewidth are not well defined for directed graphs. Fortunately, it turns out that the proof needs very little change from the original as explained in Section 4.6.4.

The theorem this section sets out to prove is the following, and is proven in Section 4.6.6.

Theorem 4.1. Directed Planar Steiner Tree *can be solved in* $2^{O(k)}n^{O(\sqrt{k})}$ *time and polynomial space when* k *is the face cover number of the input graph.*

[2]The Exponential Time Hypothesis states that there is no subexponential-time algorithm for 3-SAT.

4.6.1 Directed block Steiner forests

A directed block Steiner forest is defined similarly to its undirected variant. Instead of by a regular partitioning of the boundary vertices, the connectivity is encoded by a "rooted" partition, which defines for each block a root vertex.

Definition 4.1 (Rooted partition)**.** *A* rooted partition π *on a set* B *is a set* $\{(r_1, B_1), ..., (r_p, B_p)\}$ *such that*

1. $B_1 \cup ... \cup B_k = B$;
2. *for every* $i \neq j$, $B_i \cap B_j = \emptyset$;
3. *for every* i, $r_i \in B_i$.

The algorithm works on directed forests that have a structure represented by a rooted partition on the boundary vertices. For every block in the rooted partition, there must be one connected component with all its boundary vertices in it, and the block's root as its root, while every terminal vertex must be connected to one of these components.

Definition 4.2 (Directed Block Steiner Forest)**.** *On a directed graph* G *with arc weights* $w : A(G) \to \mathbb{R}_{\geq 0}$, *terminals* $K \subseteq V(G)$, *block vertices* $B \subseteq V(G)$ *and a rooted partition* π *on* B, *a* (G, w, K, B, π)-directed block Steiner forest *is a subgraph* S *of* G *such that*

1. *for every block* (r_i, B_i) *of* π, S *contains a path from* r_i *to every vertex in* B_i;
2. *for every terminal* $t \in K$, *there is some block* (r_i, B_i) *in* π *such that* S *contains a path from* r_i *to* t.

A minimum directed block Steiner forest *is a directed block Steiner forest of minimum weight.*

Note now, that a (G, w, K, r)-(minimum) directed Steiner tree is equivalent to a $(G, w, K, \{r\}, \{(r, \{r\})\})$-(minimum) directed block Steiner forest. Since this forest has only one block in its partition, it will only have one connected component meaning it is a tree, and this tree has r as its root and connects all terminals. This means that using the algorithm to compute a minimum directed Steiner tree simply involves computing the right minimum directed block Steiner forest.

The problem statement follows simply:

Planar Directed Block Steiner Forest on Few Faces
Input: directed plane graph G, arc weights w, terminals K, face cover $\mathcal{K}$ of K, boundary vertices B and rooted partition π of B
Asked: a minimum (G, w, K, B, π)-directed block Steiner forest

4.6.2 Base case

For the recursive algorithm to work, there must be a base case for when the input has a low number of terminal faces and boundary vertices. At this point, the input size has been sufficiently reduced that it should be solved with a different algorithm. This section describes a simple algorithm that can solve the problem in polynomial time if the number of terminal faces and boundary vertices is bounded by a constant c_0.

Lemma 4.3. *For any constant* $c_0 \in \mathbb{N}$, *there exists a polynomial-time algorithm to solve* Planar Directed Block Steiner Forest on Few Faces *provided that* $|\mathcal{K}| + |B| \leq c_0$.

The constant c_0 is quite arbitrary. Its precise value does not make a difference in terms of asymptotic complexity but can make a difference to the running time in practice. For theoretical purposes, it does not need defining but if the algorithm were to be applied in practice, an experimental setup could be used to determine an optimal value for specific types of instances.

The base case algorithm works by assigning intervals of terminals to each block of the input, and then computing a tree for every block using a polynomial-time algorithm like the variant of Dreyfus-Wagner, described in Section 4.3. Since these trees should not intersect each other, the number of possible assignments of terminal intervals to blocks is small. The details of this are described below.

Definition 4.3 (Non-crossing sequence)**.** *A* non-crossing sequence *is an n-element sequence $x \in \{1, ..., \ell\}^n$ with the property that for any $a, b \in \{1, ..., \ell\}$ with $a \neq b$, there exist no i, j, k, l with $1 \leq i < j < k < l \leq n$ such that $x_i = x_k = a$ and $x_j = x_l = b$. A non-crossing sequence is* minimal *if there is no i such that $x_i = x_{i+1} = x_{i+2}$, ie. there are no three elements in a row with the same value.*

Note the similarity between non-crossing sequences and the "crossing" constraints described in Section 4.3. Non-crossing sequences are used to encode to which block each terminal is assigned.

Lemma 4.4 (Lemma 3.6 from [23])**.** *A minimal non-crossing sequence on ℓ values has length at most 4ℓ.*

Proof. The proof is by induction on ℓ. For $\ell = 1$, a minimal non-crossing sequence has at most two elements so the statement holds.

Let x be a minimal non-crossing sequence on $\ell \geq 2$ elements. For each $v \in \{1, ..., \ell\}$, define $l(v)$ and $r(v)$ to be the leftmost and rightmost indices of v in x. Obviously, for separate values a and b, it cannot be that $l(a) < l(b) < r(a) < r(b)$. It follows from this that there must be a value v with $r(v) \leq l(v) + 1$, ie. its elements do not enclose the elements of another value. Since the sequence is minimal, there are at most two elements with this value.

If the elements $l(v)$ and $r(v)$ are removed from the sequence, one ends up with a non-crossing sequence on $\ell - 1$ elements. This sequence can be made minimal by removing at most two elements: the element that was to the left of $l(v)$ and the one that was to the right of $r(v)$. This could be the case if the same value appears on both sides of v's interval, but since the original sequence is minimal, there can be only at most four of this value in a row in the new sequence. Reducing this new interval to at most two elements makes the new sequence minimal as well.

Thus it is shown that a minimal non-crossing sequence on ℓ values can be reduced to a minimal non-crossing sequence on $\ell - 1$ values by removing at most four elements. By the induction hypothesis, the reduced sequence has at most $4(\ell-1)$ elements, so the original sequence has at most 4ℓ elements. □

The algorithm works as follows.

Proof of Lemma 4.3. Let $\ell = |B|$, and suppose some face in $\mathcal{K}$ has terminals $t_1, ..., t_p$. These p terminals can be assigned to the ℓ blocks using a non-crossing sequence. By making such a non-crossing sequence minimal, and storing for each element to which terminal it points (meaning the removed elements can be deduced from this), there are at most $\ell^{4\ell} p^{4\ell} < \ell^{4\ell} n^{4\ell}$ possible sequences for these terminals. This value is polynomial in n because $\ell = |B| \leq c_0$.

The algorithm works by, for each terminal face separately, iterating over all possible assignments, solving each individual block (with its assigned terminals) using Dreyfus-Wagner, and taking the assignment with minimum total weight. Combining the optimal solutions for all terminal faces gives an optimal solution overall.

There is a constant number of terminal faces and a polynomial number of assignments per face. The Dreyfus-Wagner algorithm works in polynomial time, as it has at most $|B| - 1$ loose terminals with all the other terminals being on a single face. Multiplying these values together gives a running time that is also polynomial. □

4.6.3 Connecting directed forests

In the original undirected algorithm, the solution is separated into smaller and smaller forests. Provided that the different parts of the forests are connected properly, this is not a complicated process as it is not hard to see that a separation of a tree yields a forest. In the directed variant of the algorithm it does get slightly more complicated, because the tree has a distinct root, and the arc directions matter. This section investigates the complexity of applying the same process to directed forests, and shows that it is computationally not more difficult than in the undirected version.

For designing a directed variant of the algorithm, it is important to understand the structure of a graph after separating an undirected tree. As it turns out, this process is very similar to the undirected version, as a separation of a directed tree is simply a directed forest, consisting of multiple directed trees.

Lemma 4.5. *Let T be a directed tree or forest and let $X \subseteq V(T)$ be a subset of T's vertices. The graph $T[X]$, obtained by removing all vertices not in X, is a directed forest.*

This can be easily proven by the following observation:

Observation 4.2. *A directed graph is a directed forest if and only if it is acyclic and every vertex has an in-degree of at most 1.*

This observation follows simply from the fact that every vertex must have exactly one "root" if it has at most one parent; this root can be determined by simply following the arcs backwards until a vertex without any in-arcs is reached.

The lemma follows immediately from this fact.

Proof of Lemma 4.5. Both properties of directed forests are closed under removing vertices from the graph. If a vertex is removed from the graph, it cannot spontaneously become cyclic, nor can the degree of a vertex increase. Thus, if T is a directed forest, any graph obtained by removing vertices from it is too. □

From this, it follows that every step of the algorithm should return a directed forest. It is still important to encode how these forests should connect to each other to form a proper tree in the end. In the original algorithm, this was done by treating the separating vertices as "boundary vertices" for the forests in subproblems, and encoding which boundary vertices should appear in the same connected component. In the directed variant, it is also important to encode the roots of these connected components, as to ensure that paths retain their proper direction.

As mentioned earlier, the forests used in the algorithm can be encoded using a rooted partition, which is a partition of vertices which assigns a root for each block. A block (r_i, B_i) now corresponds to a connected component (directed tree) C, in that r_i should be C's root and every vertex from B_i should appear in C.

The following concepts can be defined for rooted partitions:

Definition 4.4 (Blocks and roots)**.** *For a rooted partition π on B and a vertex $b \in B$:*

- *$\pi(b)$ is the block containing b, ie. B_i such that $b \in B_i$;*
- *$root_\pi(b)$ is the root of the block containing b, ie. r_i such that $b \in B_i$;*
- *$roots(\pi)$ is the set of roots $\{r_1, ..., r_p\}$ in π.*

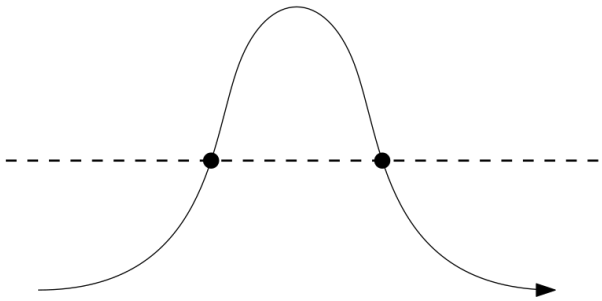


Figure 4.7: A path is intersected twice by the separator, splitting it into three parts delimited by boundary vertices. The direction needs to be maintained in each of the three parts properly.

Definition 4.5 (Consistency)**.** *Two rooted partitions π_1 and π_2 on B_1 and B_2 with $B_1 \cap B_2 \neq \emptyset$ are called* consistent *if:*

1. *for every $(r_i, B_i) \in \pi_1$ and $(r_j, B_j) \in \pi_2$, $B_i \cap B_j = \emptyset$, or $r_i \in B_j$ or $r_j \in B_i$;*
2. *there is no "loop" in the roots of the partitions. If any root of either partition is not a root in the other, it should be possible to walk up the line of roots through both partitions without encountering a cycle.*

Definition 4.6 (Joint rooted partition)**.** *For two consistent rooted partitions π_1 and π_2 on B_1 and B_2, the $\pi_1 \sqcup \pi_2$ is their* joint rooted partition*, on $B_1 \cup B_2$, obtained by the following process:*

1. *start with the set $\pi_1 \cup \pi_2 = \{(r_1, B_1), ..., (r_p, B_p)\}$;*
2. *as long as there are two distinct blocks (r_i, B_i) and (r_j, B_j) with $r_j \in B_i$, combine the two blocks into one block $(r_i, B_i \cup B_j)$.*

The result of this fits the definition of a rooted partition, provided that π_1 and π_2 are consistent.

Definition 4.7 (Restriction)**.** *For a rooted partition π on B and a set $B' \subseteq B \setminus roots(\pi)$, $\pi_{|B'}$ is the* restriction *of π on B', defined as $\pi_{|B'} = \{(r_i, B_i \cap B') : (r_i, B_i) \in \pi\}$.*

Definition 4.8 (Encoded forest)**.** *A rooted partition π on B* encodes *a directed forest T, if every block from π corresponds to a component of T with the same root and containing the same vertices.*

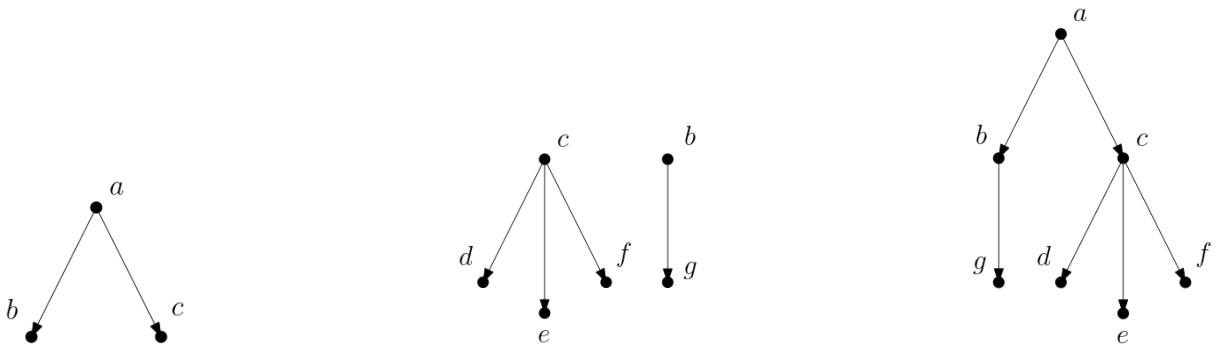
The consistency property and the join operator can be visualized by defining for each partition a rudimentary directed graph (the simplest directed forest encoded by the partition), and taking the union of these graphs. If this union is still a directed forest, the two partitions are consistent. A simple example of this is shown in Figure 4.8.

The central point of these definitions is to prove that consistent rooted partitions can be used to encode the forests required by subproblems. When we an instance has the rooted partition π in its input, it needs to produce a forest encoded by π. In turn, the subproblems of this instance require rooted partitions π_1 and π_2, but these must be chosen in a way to ensure that the join of their results fits the requirement of π.

Lemma 4.6. *Let π be a rooted partition on B, and let π_1 and π_2 be consistent rooted partitions on B_1 and B_2 with $B_1 \cup B_2 \supseteq B$ such that $(\pi_1 \sqcup \pi_2)_{|B} = \pi$. Let T_1 be a directed forest encoded by π_1 and T_2 be one encoded by π_2. Then the union of T_1 and T_2 is a directed forest encoded by π.*

Proof. Let T be the union of the graphs T_1 and T_2. The aim is to prove that T is a directed forest, and that for every vertex $b \in B$, the root of b in T is $\text{root}_\pi(b)$.

Since π_1 and π_2 are consistent, the following two conditions must hold:



(a) $\pi_1 = \{(a, \{a, b, c\})\}$

(b) $\pi_2 = \{(c, \{c, d, e, f\}), (b, \{b, g\})\}$

(c) $\pi_1 \sqcup \pi_2 = \{(a, \{a, b, c, d, e, f, g\})\}$

Figure 4.8: Representations of two rooted partitions and their union. The union is a directed forest itself, so the partitions are consistent.

1. for every vertex $b \in B$, either $\text{root}_{\pi_1}(b) = b$ or $\text{root}_{\pi_2}(b) = b$ or $\text{root}_{\pi_1}(b) = \text{root}_{\pi_2}(b)$;
2. between the two partitions, there is no cycle in roots.

Note the correspondence with the conditions of directed forests described in Observation 4.2. The first condition ensures that, between the two partitions, no vertex has more than one parent. The second ensures that there is no cycle. Since the parent-vertex relation in π_1 and π_2 is represented by paths in T_1 and T_2, this proves that T_1 and T_2 together fit the definition of a directed forest.

For the second part of the proof, I will prove that the root of every vertex from B is the same in the graph T as in the rooted partition π. Take any vertex b from B in T_1 (resp. T_2), and follow its parent chain to reach its root $r = \text{root}_{\pi_1}(b)$. If $r = \text{root}_{\pi}(b)$, then we are done. Otherwise, it must be that $\text{root}_{\pi_2}(r) \neq r$ meaning the parent chain continues in T_2 (resp. T_1). This follows from the fact that b must have the same root in both π and $\pi_1 \sqcup \pi_2$, which would not be the case if the parent chain ended in r. Following this parent chain up, switching between T_1 and T_2 when needed, will end up in the proper root, $\text{root}_{\pi}(b)$. □

Lemma 4.6 shows that the only condition for picking rooted partitions in the algorithm, is that the two chosen rooted partitions are consistent and that their join restricted to the original boundary vertices, equals the original partition. To actually determine these rooted partitions, it proves to be enough to simply enumerate all possibilities and checking if they satisfy the requirements. A loose upper bound for the number of rooted partitions on a set B is $|B|^{|B|}$. This number is obtained by treating the rooted partition as a function in the space $B \to B$, which assigns to every element a root. It follows from this that enumerating the pairs of partitions on $B_1 \cup B_2$ takes $O(|B_1 \cup B_2|^{|B_1 \cup B_2|})$ time.

4.6.4 Choosing a separator

The core of the algorithm lies in choosing a separator between the two subproblems. The goal here is to prove there always exists a set of vertices in the graph of limited size, such that it forms a balanced separation between the terminal faces and boundary vertices in the resulting block Steiner forest.

This is done by proving an upper bound of the treewidth of a auxiliary graph that contains the solution and the terminal faces. From the treewidth, an upper bound on the size of a minimal balanced separator can be inferred. Interestingly, this part of the algorithm is virtually identical to the undirected algorithm, despite treewidth not having a proper definition for directed graphs.

Let T be a minimum directed block Steiner forest for a given instance of the problem, and let $\mathcal{K}^\flat$ be the graph containing only the terminal faces (with their incident vertices and arcs) of the instance. Now define $H' = T \cup \mathcal{K}^\flat$ to be the union of these two graphs. Let H be an undirected graph that is obtained by removing arc directions from H' (in the case of two parallel, opposite-direction arcs, they can be replaced by a single edge; the weight is unimportant).

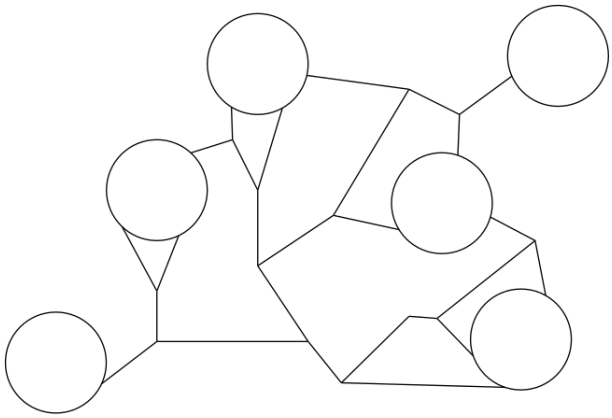


Figure 4.9: An example of the graph H. The circles represent terminal faces and the straight segments form the solution tree/forest.

Lemma 4.7. *$tw(H) \leq 15\sqrt{|\mathcal{K}|} + 1$, where $|\mathcal{K}|$ is the number of terminal faces in the problem instance.*

Proving this requires the following two theorems:

Theorem 4.2. *If a planar graph G admits a dominating set of size k, then $tw(G) \leq 15\sqrt{k}$ [11, 12].*

Theorem 4.3. *Let G be a planar graph and let G^* be its planar dual. Then $|tw(G) - tw(G^*)| \leq 1$ [9, 27].*

The proof of Theorem 4.2 is by bidimensionality; the full details can be found in the cited sources but a summary is given here. It can easily be shown that a dominating set on a grid graph has a size that is at least linear in the grid's size. By the excluded grid theorem (Theorem 2.2), every planar graph contains a grid minor whose size is linear with regard to the graph's treewidth; with this, one can either obtain a "triangulated grid" as a topological minor (Theorem 7.25 from [11]) or apply only the edge contractions from the minor model (and not the edge/vertex deletions) to obtain a grid that contains pieces of the original graph in some of its faces [12]. Using either of these models, combined with the bound on a dominating set in a grid, it can be shown that a planar graph with treewidth w has a dominating set of size at least $O(w^2)$ and vice versa. The factor 15 follows from the proof's technicalities and is unimportant to the algorithm's theoretical complexity.

The proof of Theorem 4.3 is much more complicated. The statement was first conjectured early on by Robertson and Seymour [37] but not proven until later. The first proof [27] involves looking at the equivalents of tree decompositions and planar duals in hypergraphs, which are graphs that permit "edges" with more than two endpoints. A later proof [9] of the theorem is slightly simpler and involves separators on maximal planar graphs.

Proof of Lemma 4.7. Let H^* be the planar dual of H, which is obtained by creating a vertex for every face in H and connecting two vertices if their corresponding faces share an edge in H. I will prove that H^* has a dominating set of size $|\mathcal{K}|$.

In H, apply the following coloring to the edges. Color an edge *red* if it replaces an arc from $\mathcal{K}^\flat$, ie. it is incident to a terminal face in the original graph. If an edge is not colored red, this

means it represents an arc from T, meaning it is part of the solution forest. In this case, color it *blue*. If an edge represents arcs from both $\mathcal{K}^\flat$ and T, color it red.

Applying this coloring to the edges has two consequences. Firstly, every terminal face in H has a border consisting entirely of red edges. Secondly, every other face has at least one red edge on its border. To see this, consider the graph obtained by removing all the red edges from H. This ends up as a subgraph of T (ignoring arc directions); but since T cannot contain any cycles due to it being a forest, there cannot be a cycle in the blue edges of H either. Therefore it is impossible to surround a face entirely by blue edges, meaning every face must contain at least one red edge.

Since every red edges borders a terminal face, and every face has a red edge on its border, it must follow that every face in H either is a terminal face, or is adjacent to a terminal face. Going back to the dual graph H^*, this translates to there being a dominating set consisting of the vertices corresponding to terminal faces; this dominating set has size $|\mathcal{K}|$.

From Theorem 4.2 it then follows that $\mathrm{tw}(H^*) \leq 15\sqrt{|\mathcal{K}|}$. From Theorem 4.3, $|\mathrm{tw}(H) - \mathrm{tw}(H^*)| \leq 1$, which proves that $\mathrm{tw}(H) \leq 15\sqrt{|\mathcal{K}|} + 1$. □

From Lemma 4.7 and Theorem 2.1, it follows that H must contain a $\frac{2}{3}$-balanced separator (Y, Z) with $|Y \cap Z| \leq 15\sqrt{|\mathcal{K}|} + 2$. It is not hard to see that (Y, Z) is also a separator on the directed graph H', as the underlying structure of the graph is not changed and every directed path in H' is still a path in H.

Define the following weight function $\omega : V(H') \to \mathbb{N}$:

- for every vertex $v \in V(H') \setminus B$, set $\omega(v) \leftarrow 0$;
- for every boundary vertex $v \in B$, set $\omega(v) \leftarrow 1$;
- for every terminal face $F \in \mathcal{K}$, pick an arbitrary vertex v from $V(F)$ and set $\omega(v) \leftarrow \omega(v) + 1$.

The total weight of this function is $|\mathcal{K}| + |B|$, and as shown, (Y, Z) is a $\frac{2}{3}$-balanced separation on this weight function. This is not enough to prove a balanced separation on the terminal faces though, as it is possible that the "point" for a face in ω is counted for one part of the separation while the separator does cross the face still, and nothing is counted for the other half. Accounting for separated faces (which are represented in both subproblems), it is still possible to prove a constant-factor balance provided that the input is large enough. It should also be noted that every vertex in the separator appears as a boundary vertex in both subproblems.

Lemma 4.8. *Let $|\mathcal{K}|$ and $|B|$ be the number of terminal faces and boundary vertices for a given directed block Steiner forest instance, and let $|\mathcal{K}'|$ and $|B'|$ be the respective parameters of one of its subproblems. Provided that $|\mathcal{K}| + |B| \geq c_0$ for some constant c_0, it is possible to choose a separator X, with $|X| \leq 15\sqrt{|\mathcal{K}|} + 2$ such that $|\mathcal{K}'| + |B'| \leq \frac{3}{4}(|\mathcal{K}| + |B|)$.*

First, the following simple lemma is proven in order to bound the separated faces:

Lemma 4.9. *Let G be a plane graph; let F be a subset of G's faces and let $X \subseteq V(G)$. There are at most $3|X| - 6$ faces in F that have at least two vertices from X on their border.*

Proof. Let F' be the set of faces in F with at least two vertices from X on their border. Construct a graph R with $V(R) = X$ and for every face f in F', add one edge to R connecting an arbitrary pair of vertices on f's border.

Obviously, R is planar, $|V(R)| = |X|$ and $|E(R)| = |F'|$. Because R is planar, $|E(R)| \leq 3|V(R)| - 6$, meaning $|F'| \leq 3|X| - 6$, completing the proof. □

Proof of Lemma 4.8. As proven earlier using Lemma 4.7 and Theorem 2.1, there exists a $\frac{2}{3}$-balanced separation (Y, Z) in the graph H' of the right size. Let $X = Y \cap Z$ be the separator. It is possible to draw a closed simple curve through all vertices of X that does not intersect any other vertices or arcs in H'. Every terminal face that is separated by this curve has to be represented in both subproblems. Unfortunately, this curve is not known at the time of the algorithm so it is impossible to know exactly which terminal faces are fully separated and which ones are only touched in some vertex or vertices. However, for the curve to properly separate a terminal face, it must go through at least two vertices on the face's border, so it suffices to take all terminal faces that have at least two vertices from X on their border.[3]

By Lemma 4.9, there are at most $3|X| - 6$ such faces. It follows then, that the input parameters of the subproblems are

$$|\mathcal{K}'| + |B'| \leq \frac{2}{3}\left(|\mathcal{K}| + |B|\right) + 4|X| - 6$$

accounting for the balanced separation on non-separated terminal faces and boundary vertices, the $|X|$ separator vertices that appear as boundary vertices in both subproblems, and the at most $3|X|-6$ terminal faces that are separated between the subproblems. Since $|X| = O(\sqrt{|\mathcal{K}|})$, it follows that this value is less than $\frac{3}{4}(|\mathcal{K}| + |B|)$ if $|\mathcal{K}| + |B|$ is greater than some constant. □

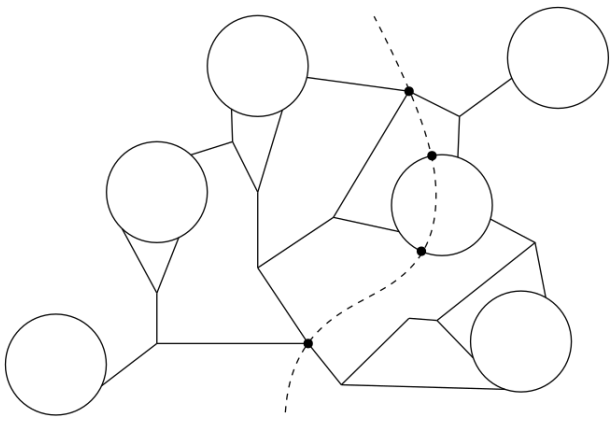


Figure 4.10: A four-vertex separator in the graph of Figure 4.9, with a curve drawn through it. One of the terminal faces is separated.

4.6.5 Algorithm

The algorithm follows. It is essentially the same as the undirected algorithm summarized in Section 4.5.4; only in its implementation would it be important to account for arc directions properly. Pseudocode for the algorithm is listed in Algorithm 4.2.

In line 2, the base case algorithm is called as the input size is small enough. In line 5, a separator is guessed based on the findings of Section 4.6.4. Line 6 determines the terminal faces that were separated by the separator, ie. the ones that have at least two vertices from X on their border. In line 7, the remaining terminal faces and boundary vertices are divided between the two subproblems. Line 10 divides the segments of separated terminal faces; here $\mathrm{cc}(\mathcal{K}_\bullet, X)$ is the family of sets of terminals that lie on contiguous segments along terminal face borders, delimited by separator vertices from X. If X contains any terminals, these are not counted as they would appear as boundary vertices in the subproblems anyway. Based on this information,

[3]The original paper [23] uses a different method to obtain an upper bound of $3|X|$ on the intersected faces, but it requires that the input graph is converted to an equivalent subcubic graph which is not necessary in my proof.

Algorithm 4.2 Algorithm `steiner` for Directed Block Steiner Forest with Terminals on Few Faces

Input: directed graph G, arc weights $w : A(G) \to \mathbb{R}^+$, terminals $K \subseteq V(G)$, terminal faces $\mathcal{K} \subseteq 2^{A(G)}$, boundary vertices $B \subseteq V(G)$, rooted partition π of B

Output: the weight of a (G, w, K, B, π)-minimum directed block Steiner forest

1: **if** $|\mathcal{K}| + |B| \leq c_0$ **then**
2: **return** steinerBase$(G, w, K, \mathcal{K}, B, \pi)$ ▷ Defer to the base case algorithm
3: **end if**
4: best $\leftarrow \infty$
5: **for** $X \subseteq V(G)$ with $|X| \leq 15\sqrt{|\mathcal{K}|} + 2$ **do** ▷ Pick a separator
6: $\mathcal{K}_\bullet \leftarrow \{F \in \mathcal{K} : |V(F) \cap X| \geq 2\}$ ▷ Determine the terminal faces intersected by X
7: **for** $\mathcal{K}_1 \subseteq \mathcal{K} \setminus \mathcal{K}_\bullet$ and $B_1 \subseteq B \setminus X$ such that $\frac{1}{3} \leq \frac{|\mathcal{K}_1|+|B_1|}{|\mathcal{K}|+|B|} \leq \frac{2}{3}$ **do**
8: $\mathcal{K}_2 \leftarrow \mathcal{K} \setminus (\mathcal{K}_\bullet \cup \mathcal{K}_1)$ ▷ Divide the untouched terminal faces and boundary vertices
9: $B_2 \leftarrow B \setminus (X \cup B_1)$
10: **for** $S_1 \subseteq \mathrm{cc}(\mathcal{K}_\bullet, X)$ **do** ▷ Divide the segments of split terminal faces
11: $K_1 \leftarrow ((K \cap V(\mathcal{K}_1)) \cup S_1) \setminus X$ ▷ Determine the terminals for each subproblem
12: $K_2 \leftarrow K \setminus (X \cup K_1)$
13: **for** partitions π_1 on $B_1 \cup X$ and π_2 on $B_2 \cup X$ **do** ▷ Enumerate partitions
14: $\pi' \leftarrow \pi_1 \sqcup \pi_2$
15: **if** $\pi'_{|B} = \pi$ and for every vertex $x \in X$, $\pi'(x) \cap B \neq \emptyset$ **then**
▷ Ensure that the partitions are proper
16: best$_1 \leftarrow$ steiner$(G, w, K_1, \mathcal{K}_1, B_1, \pi_1)$
17: best$_2 \leftarrow$ steiner$(G, w, K_2, \mathcal{K}_2, B_2, \pi_2)$
18: best $\leftarrow \min\{\text{best}, \text{best}_1 + \text{best}_2\}$
19: **end if**
20: **end for**
21: **end for**
22: **end for**
23: **end for**
24: **return** best

the full sets of terminals for each subproblem are computed. Line 13 enumerates all potential rooted partitions for the subproblems, and line 15 checks whether or not their join matches the original partition. It is important that every vertex from X is connected to some original boundary vertex, as otherwise these might appear in loose connected components in the end. Lastly, the recursive calls are performed based on the guessed information, and the best result is returned.

The algorithm simply returns the weight of a minimum directed Steiner forest. To actually determine the forest itself is quite straightforward. After a call of `steinerBase`, the resulting forest is saved and passed up to last recursive call of `steiner`. At line 18, where the weights of the two Steiner forests are compared to the previous best, the two determined forests are also combined. If this is a better result than the previous best, this combined forest replaces the best forest in memory. At the end of the algorithm, the best forest is returned along with its weight.

4.6.6 Analysis

With the algorithm written, it is possible to prove its correctness and complexity.

Correctness

The correctness is proven by showing that the algorithm considers an optimal result and does not consider any infeasible results; here a feasible result is one that fits the input parameters but is not necessarily minimum.

Lemma 4.10. *Algorithm 4.2 gives a correct result.*

Proof. The correctness of the base case of the algorithm follows from Lemma 4.3. The rest of the proof is in two parts: first, I prove that the returned value must be a feasible directed block Steiner forest fulfilling the input parameters, by showing that every forest considered for the minimum is so. Then, I show that a forest of minimum weight is one of the considered forests.

For the first part, consider the recursive calls in lines 16-17. It can be assumed, by the principle of induction, that the returned values of the recursive calls are correct; best_1 is the weight of a (G, w, K_1, B_1, π_1)-directed block Steiner forest and best_2 is the weight of a (G, w, K_2, B_2, π_2)-directed block Steiner forest. Let T_1 be the block Steiner forest corresponding to best_1 and T_2 that corresponding to best_2. Since $(\pi_1 \sqcup \pi_2)_{|B} = \pi$, the union $T = T_1 \cup T_2$ is a directed block Steiner forest encoded by π (Lemma 4.6). Furthermore, every terminal in K is either in K_1, K_2 or X, which are all connected in T. It follows from these facts that both the terminals and boundary vertices from the input are properly connected in T, meaning that T is a (G, w, K, B, π)-directed block Steiner forest and thus all considered results are feasible.

For the second part of the proof, consider T' to be a minimum directed block Steiner forest for the input. By the enumerations in line 5, a set of vertices that separates T' must be considered at some point. Let T'_1 and T'_2 be the two subgraphs induced by this separation. This separator splits all the terminals, terminal faces and boundary vertices in T' between T'_1 and T'_2. The splits of non-touched terminal faces and boundary vertices are considered at some point in the enumerations of line 7; the terminals on touched terminal faces can only be split along the lines of intervals delimited by separating vertices, and this split is considered in line 6. Taking the boundary vertices and the separating vertices and partitioning them along the connected components they appear in together in either part of T''s separation, yields two rooted partitions. Combining these partitions should give the original partition of T' (by Lemma 4.6), meaning the two partitions are considered in the enumeration of line 13. It follows from all this that the algorithm considers at some point a set of parameters K_1, K_2, B_1, B_2, π_1 and π_2 such that T'_1 is a (G, w, K_1, B_1, π_1)-directed block Steiner forest and T'_2 is a (G, w, K_2, B_2, π_2)-directed block Steiner forest. The recursive call in line 16 returns a forest equivalent to T'_1 and the one in line 17 returns a forest equivalent to T'_2; the weights of these added together must then equal the weight of T', which is considered for the minimum.

Since all considered forests are feasible, and at least one minimum forest is considered, the output of the algorithm must be a minimum (G, w, K, B, π)-directed block Steiner forest. □

Complexity

The complexity follows from the number of iterations in each "guessing" step, and the fact that the recursion is proven to reduce the input size by a constant factor. Interestingly, it is identical to the complexity of the undirected algorithm as described in Section 4.5.4.

Proof of Theorem 4.1. Let $k = |\mathcal{K}|$ from the original problem input throughout. From Lemma 4.8 it follows that the algorithm has a recursion depth of $O(\log k)$. Furthermore, since the set of boundary vertices increases by the separator vertices X (and usually decreases some), with $|X| = O(\sqrt{k})$, it can be held as an invariant that $|B| = O(\sqrt{k} \log k)$ at any recursive call of the algorithm.

If $|\mathcal{K}| + |B| \leq c_0$, the base case algorithm is called which runs in $n^{O(1)}$ time. Otherwise, the following parameters are guessed by enumeration. Here $p = |\mathcal{K}| + |B|$.

- X (line 5), which has $n^{O(\sqrt{p})}$ options;
- $\mathcal{K}_1$ and $\mathcal{K}_2$ (line 7) with $2^{O(p)}$ options;
- B_1 and B_2 (line 7) with $2^{O(p)}$ options;
- S_1 (line 10) with $2^{O(\sqrt{p})}$ options;
- π_1 and π_2 (line 13) with $O((\sqrt{k}\log k)^{\sqrt{k}\log k})$ options.

Other than these enumerations and the recursive calls, every operation in the algorithm takes polynomial time. This gives the following recurrence:

$$T(n,p) = \begin{cases} n^{O(1)} & \text{if } p \leq c_0 \\ n^{O(\sqrt{p})} 2^{O(p)} T(n, \frac{3}{4}p) & \text{otherwise.} \end{cases}$$

This recurrence solves to $2^{O(p)} n^{O(\sqrt{p})}$. Filling in k for p proves this part of the theorem.

The space complexity follows from the fact that the algorithm only needs to store the above parameters which are of polynomial size, as well as maintaining the best seen Steiner forest and weight. □

Chapter 5

Conclusions and discussion

This thesis has discussed two very well-known problems in graph theory. While these problems have much in common, given the fact that they both ask for some connectivity in a given set of vertices allowing for a corresponding parameterized formulation, they require quite different ideas to solve.

The algorithms for Disjoint Paths and Steiner Tree function very well on planar graphs, especially when many terminals are incident on the same few faces, due to topological properties of the graph. For instance, in Disjoint Paths, knowledge about one path may form an "obstruction" to the vertices of another path, which is utilized in the one-face algorithm of Section 3.5.1 for example.

A notion that has appeared multiple times throughout the investigation of both problems, is that of "non-crossing" sets of terminals. When four terminals s_i, s_j, t_i, t_j appear in this order cyclically around one face's border, then it can immediately be rejected that the pairs (s_i, t_i) and (s_j, t_j) could be connected by two disjoint components: due to their ordering, these two components would necessarily intersect. In Disjoint Paths, this idea can be used to immediately reject instances as it implies disjoint paths are not possible. In the Dreyfus-Wagner algorithm for Steiner Tree, it is used to significantly reduce the number of subproblems, as two intersecting trees will not lead to a minimum solution.

Both problems allow an XP algorithm parameterized by the face cover number of the terminals. Face cover number parameterizations appear powerful, as they (virtually) eliminate the dependence on the exact number of terminals, and map well to certain real-life applications of the problems. Moreover, it was seen for Disjoint Paths, that a face cover number of 1 or 2 allows for a linear-time algorithm. This is especially significant considering the extremely slow algorithms in more general cases.

Below is, for both problems, a summary of the discussed ideas and results in the thesis, as well as some open questions and directions for future research.

5.1 Disjoint Paths

Several algorithms for Disjoint Paths have been discussed. The first algorithm, famously by Robertson and Seymour, shows that the problem is FPT parameterized by the number of terminals, which was an important discovery in the overarching work of the authors. Unfortunately, the parameter dependence of the algorithm's running time is enormous, making it completely impractical for real-world usage. The algorithm introduced the *irrelevant vertices technique*, which has become a staple in further efforts to create algorithms for the problem. A requirement for the irrelevant vertices technique is an FPT algorithm parameterized by the graph's treewidth, which was developed by Scheffler.

Applying the irrelevant vertices technique is shown by Adler et al. to be fruitful when the input is restricted to planar graphs: here, the parameter dependence is "only" $2^{2^{O(k)}}$. A

later algorithm by Lokshtanov et al. combined the technique with other methods, to obtain a parameter dependence of $2^{O(k^2)}$ on planar graphs, which is the best known result yet.

Schrijver used an algebraic approach, leading to an XP algorithm on planar graphs. He used a similar algorithm to show that the problem is also XP when parameterized by the face cover number of the terminals. It is also shown that the problem can be solved in linear time if the terminals are covered by one or two faces.

Better parameterized algorithms. A leading question in research about Disjoint Paths remains to find better parameterized algorithms than the ones currently known. While Robertson and Seymour's algorithm proves definitively that Disjoint Paths is in FPT, there is absolutely no practical use for it. A naïve algorithm could simply enumerate all $2^{|E|}$ subsets of edges in the input graph, and check for each subset whether or not it is a valid solution to the problem. This algorithm would be preferable to the one by Robertson and Seymour in every realistic application, despite not being fixed-parameter tractable.

The only significant improvements that have been made based on Robertson and Seymour's algorithm, have been on planar graphs specifically. To break the existing lower bound of $2^{2^{\Omega(k)}}$ in Planar Disjoint Paths, it was necessary to abandon the "pure" irrelevant vertices technique and combine it with other approaches. This begs the question of whether or not a similar method is necessary on general graphs. It is a question well beyond the scope of this thesis, considering the enormous effort required to achieve Robertson and Seymour's running time in the first place let alone its improvements, but a question worth considering nonetheless.

FPT algorithm on few faces. The algorithm for few-faces Planar Disjoint Paths by Schrijver, discussed in Section 3.5.3, runs in some $n^{f(k)}$ time, meaning that it is in XP. While this means the algorithm runs in polynomial time for fixed k, it is less desirable than an FPT algorithm, so an FPT algorithm could be a direction of future research.

In Section 3.5.4, it is shown that the irrelevant vertices technique cannot be applied to this variant of the problem. Of course, this is not a conclusive proof that Planar Disjoint Paths parameterized by the face cover number is not FPT; other techniques may still prove useful.

5.2 Steiner Tree

For the Steiner Tree problem, two algorithms were discussed and both were lifted to directed graphs. The first algorithm, by Dreyfus and Wagner, can be considered the de facto standard exact algorithm for solving the problem. As it turns out, it can perform better on planar graphs with terminals on few faces than on general graphs. In general, the algorithm is in FPT parameterized by the number of terminals. On planar graphs, it is also XP parameterized by the face cover number. Both of these variants are also applicable to the directed variant of the problem, for identical running time.

A few improvements can be made to Dreyfus-Wagner. The most notable improvement, through fast subset convolution, reduces the exponential component of the algorithm's running time. Unfortunately, the technique cannot be as simply applied to the version with terminals on few faces.

The second discussed algorithm, by Kisfaludi-Bak et al., is made specifically for planar graphs with terminals on few faces. Its running time is $2^{O(k)} \cdot n^{O(\sqrt{k})}$, making it faster than the Dreyfus-Wagner variant. It is also shown that this running time is (almost) tight with a hypothetical lower bound. With small adjustments to the algorithm and its underlying proofs, it is easily applicable to the directed problem.

Hardness of Directed Steiner Tree. A noteworthy observation of the discussed Steiner Tree algorithms, is that both run in (asymptotically) equal time on undirected and directed

graphs. In fact, the formulation of directed Dreyfus-Wagner is virtually identical to the undirected version. In the algorithm by Kisfaludi-Bak et al., the only significant difference in the algorithm is in the partitions encoding the connectivity of block Steiner forests; this does not lead to an increased asymptotic running time however.

It would be the expectation that the directed variant is more difficult to solve than the undirected variant, because an algorithm on directed graphs can be used to solve an undirected instance but not the other way around. This does not appear to be the case for the two discussed algorithms. As a matter of fact, the vast majority of literature about Directed Steiner Tree appears to be on approximation algorithms and linear programming relaxations, meaning that there is likely not much new to say about exact algorithms.

Faster subset convolution on intervals. A question arising naturally from 4.3.1 is whether or not it is possible to apply the ideas of fast subset convolution to intervals, in order to shave down the running time of planar Dreyfus-Wagner on few faces. As shown in the section, set operators (specifically the union operator) cannot be properly translated to intervals, which immediately rules out a translation of fast convolution with the covering product. A direct translation of regular fast subset convolution is possible (as it does not require the union operator), but does not lead to a better running time than naïvely computing the convolutions.

Nevertheless, the discussion should not end with that. There might still be other ways to incorporate ideas from fast subset convolution. Considering the large running time improvement caused by fast subset convolution, a translation to intervals could have significant implications, especially on problems like Steiner Tree which already perform better when the terminals are incident on few faces of a planar embedding.

Faster algorithms for terminals on one face. In the chapter about Disjoint Paths, it was seen that besides the existence of an XP algorithm parameterized by the face cover number, there are specific algorithms for one and two faces that run in linear time, which are much better than simply using the XP algorithm. For Steiner Tree, no such algorithms were discussed however, and only two XP algorithms for an arbitrary face cover number were given.

Kisfaludi-Bak et al. showed, in addition to their algorithm, that Steiner Tree parameterized by the face cover number has a lower bound of $f(k) \cdot n^{\Omega(\sqrt{k})}$ on the running time assuming the Exponential Time Hypothesis is true. Considering this, it seems unnecessary to ask for an FPT algorithm. Still, it may be worth investigating algorithms for one face specifically (and possibly other low values).

Applying the technique of few-faces Dreyfus-Wagner to a single face would lead to a running time of $O(n^5)$ (assuming the number of terminals is $O(n)$). It seems like there must be a better algorithm for this, also considering that the result for Disjoint Paths on one or two faces is so much better than that. One potential way to achieve a better running time was fast subset convolution, but as shown this does not immediately seem possible.

Bibliography

[1] Isolde Adler, Stavros G Kolliopoulos, Philipp Klaus Krause, Daniel Lokshtanov, Saket Saurabh, and Dimitrios Thilikos. Tight bounds for linkages in planar graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 110–121. Springer, 2011.

[2] Isolde Adler, Stavros G Kolliopoulos, Philipp Klaus Krause, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M Thilikos. Irrelevant vertices for the planar disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 122:815–843, 2017.

[3] Isolde Adler, Stavros G Kolliopoulos, and Dimitrios M Thilikos. Planar disjoint-paths completion. *Algorithmica*, 76(2):401–425, 2016.

[4] Julien Baste and Ignasi Sau. The role of planarity in connectivity problems parameterized by treewidth. *Theoretical Computer Science*, 570:1–14, 2015.

[5] Marshall Bern. Faster exact algorithms for steiner trees in planar networks. *Networks*, 20(1):109–120, 1990.

[6] Daniel Bienstock and Clyde Monma. On the complexity of covering vertices by faces in a planar graph. *SIAM J. Comput.*, 17:53–76, 02 1988.

[7] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets möbius: fast subset convolution. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 67–74, 2007.

[8] Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

[9] Vincent Bouchitté, Frédéric Mazoit, and Ioan Todinca. Treewidth of planar graphs: connections with duality. In *Euroconference on Combinatorics, Graph Theory and Applications*, volume 10, pages 34–38, 2001.

[10] Marcus Brazil, Ronald L Graham, Doreen A Thomas, and Martin Zachariasen. On the history of the euclidean steiner tree problem. *Archive for history of exact sciences*, 68(3):327–354, 2014.

[11] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 5th edition, 2015.

[12] Erik D Demaine, Fedor V Fomin, MohammadTaghi Hajiaghayi, and Dimitrios M Thilikos. Fixed-parameter algorithms for (k, r)-center in planar graphs and map graphs. *ACM Transactions on Algorithms (TALG)*, 1(1):33–47, 2005.

[13] Stuart E Dreyfus and Robert A Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[14] Ranel E Erickson, Clyde L Monma, and Arthur F Veinott Jr. Send-and-split method for minimum-concave-cost network flows. *Mathematics of operations research*, 12(4):634–664, 1987.

[15] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[16] András Frank. Packing paths, cuts, and circuits-a survey. *Paths, Flows and VLSI-Layout*, 49:100, 1990.

[17] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[18] Joseph L Ganley. Computing optimal rectilinear steiner trees: A survey and experimental evaluation. *Discrete Applied Mathematics*, 90(1-3):161–171, 1999.

[19] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[20] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[21] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012.

[22] Ken-ichi Kawarabayashi and Paul Wollan. A shorter proof of the graph minor algorithm: The unique linkage theorem. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, page 687–694, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Sándor Kisfaludi-Bak, Jesper Nederlof, and Erik Jan van Leeuwen. Nearly eth-tight algorithms for planar steiner tree with terminals on few faces. *ACM Transactions on Algorithms (TALG)*, 16(3):1–30, 2020.

[24] Yusuke Kobayashi and Christian Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4):234–245, 2010.

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

[26] Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283, 1930.

[27] Denis Lapoire. Treewidth and duality for planar hypergraphs. 1996.

[28] Richard J Lipton and Kenneth W Regan. David johnson: Galactic algorithms. In *People, Problems, and Proofs*, pages 109–112. Springer, 2013.

[29] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 760–776. SIAM, 2011.

[30] Daniel Lokshtanov, Pranabendu Misra, Michał Pilipczuk, Saket Saurabh, and Meirav Zehavi. An exponential time parameterized algorithm for planar disjoint paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1307–1316, 2020.

[31] Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. Efficient graph minors theory and parameterized algorithms for (planar) disjoint paths. In *Treewidth, Kernels, and Algorithms*, pages 112–128. Springer, 2020.

[32] Karl Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–1159, 1927.

[33] Daniel Mölle, Stefan Richter, and Peter Rossmanith. A faster algorithm for the steiner tree problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 561–570. Springer, 2006.

[34] Bruce A Reed, Neil Robertson, Alexander Schrijver, and Paul D Seymour. Finding disjoint trees in planar graphs in linear time. *Contemporary Mathematics*, 147:295–295, 1993.

[35] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. Linear-time algorithms for disjoint two-face paths problems in planar graphs. *International Journal of Foundations of Computer Science*, 7(02):95–110, 1996.

[36] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. The vertex-disjoint menger problem in planar graphs. *SIAM Journal on Computing*, 26(2):331–349, 1997.

[37] Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[38] Neil Robertson and Paul D. Seymour. Graph minors. vi. disjoint paths across a disc. *Journal of Combinatorial Theory, Series B*, 41(1):115–138, 1986.

[39] Neil Robertson and Paul D. Seymour. Graph minors .xiii. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[40] Neil Robertson and Paul D Seymour. Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

[41] Petra Scheffler. A practical linear time algorithm for disjoint paths in graphs with bounded tree-width. 1994.

[42] Alexander Schrijver. Disjoint homotopic paths and trees in a planar graph. *Discrete & Computational Geometry*, 6(4):527–574, 1991.

[43] Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal on Computing*, 23(4):780–788, 1994.

[44] Alexander Schrijver. A course in combinatorial optimization. 2017.

[45] Anand Srinivas and Eytan Modiano. Finding minimum energy disjoint paths in wireless ad-hoc networks. *Wireless Networks*, 11(4):401–417, 2005.

[46] Jens Vygen. Np-completeness of some edge-disjoint paths problems. *Discrete Applied Mathematics*, 61(1):83–90, 1995.

[47] Klaus Wagner. Über eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114(1):570–590, 1937.