Degree-Based Spanning Tree Optimization

PhD Thesis

Gábor Salamon

Department of Computer Science and Information Theory
Budapest University of Technology and Economics

Supervisor: András Recski

2010

Alulírott Salamon Gábor kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Alulírott Salamon Gábor hozzájárulok a doktori értekezésem interneten történő korlátozás nélküli nyilvánosságra hozatalához.

.....
Salamon Gábor

Contents

Chapter 1

Introduction

1.1 Motivation and Research Goals

Design process, operation and maintenance of telecommunication networks are dynamically developing research areas. Graphs are often used as mathematical models of these networks giving a special importance to the research of effective graph algorithms. In most cases, the obtained mathematical models are too complex to be solved optimally, thus the aim is to find a "good enough" solution in an acceptable time limit. Unfortunately, "good enough" can be interpreted in many ways. Computer engineers build heuristic algorithms and examine them experimentally, in many cases without any mathematical foundation. On the other hand, mathematicians use a different approach, they try to prove theoretical limits on the performance of the algorithms. However, these limits may not necessarily guarantee that the algorithms work efficiently in practice.

The main goal of our work is to combine these approaches in order to obtain algorithms that perform well enough both in theory and in practice. We consider a design problem from the world of optical telecommunication networks, and we model it with the help of graph theory (or more precisely with spanning tree optimization problems). Finally, we present a collection of algorithms and analyze them from both the theoretical and empirical points of view.

Our mathematical analysis has direct connection to the hamiltonicity theory, therefore, beside algorithmic aspects, our results have their own theoretical importance as well.

In the next subsections, we introduce the network design problem under investigation, then we describe the considered mathematical model and the corresponding combinatorial optimization problems.

1.1.1 Connection routing in DWDM networks

In this subsection we present the telecommunication network design problem which inspired us to deal with degree-based spanning tree optimization problems.

Dense Wavelength Division Multiplexing (DWDM) is a technology widely used in optical networks in order to increase the available bandwidth. The basic idea is to use different light wavelengths within a single optical fiber enabling multiple data connections at the same time. Switch devices in a DWDM network must be able to deal with wavelength multiplexing in order to correctly route data connections. To explain how this capability can be implemented, we first give a brief and simplified insight to the *layers* of a DWDM network. For a comprehensive monograph about the DWDM networks the reader is referred to [63].

Let us have two applications which want to communicate to each other over our DWDM network. Their co-operation must be independent of the details of the underlying network protocols and technologies. Therefore, they are using an *application layer* to communicate. This application layer is built on the top of and is served by a *logical network layer* which is responsible for building up and managing the connection and for accessing the layers of physical transport. Logical network layer sends the data to be transferred to the *electronic (physical) layer* which converts it to an electronic signal. This signal now can be transported without dealing with its logical meaning. Up to this point, we have the layers of a classical network. However, optical networks transfer optical signals in optical fibers, thus they need an additional layer under the above mentioned ones: the *optical (physical) layer*. When the sender application generates a new connection demand, the electronic physical layer creates an electronic signal to be sent through the network. This electronic signal must be converted to an optical signal at the entry terminal of the DWDM network. Thus the physical transfer itself happens in the optical layer. Similarly, when the optical signal arrives to its destination, the end terminal converts it back to an electronic signal and passes it to the electronic layer which forwards it to the application. Figure 1.1 shows this layering concept whose biggest advantage is its transparency. Each layer has its own responsibility. Its functionality can be implemented without any knowledge on the details of other layers [18, 20]. (Obviously, we still need well-defined interfaces between layers.)

In *transparent DWDM* networks, the whole connection forwarding and routing is handled by full optical devices: *optical repeaters* and *optical cross connects (OXC's)*. Repeaters only forward the data stream without changing it. OXC's, in contrast, can route the connections and can execute wavelength multiplexing, demultiplexing, and conversions. However, the elevated price of OXC's makes their mass use inefficient. Therefore, network designers tend to use cheaper devices for the same task. These devices, called *electronic cross connects (EXC's)*, execute the routing and wavelength manipulation functionalities in the electronic layer. They convert the incoming optical signals to electronic ones, route connections, and then convert back the electronic signals to optical ones. Though their cost is significantly lower,

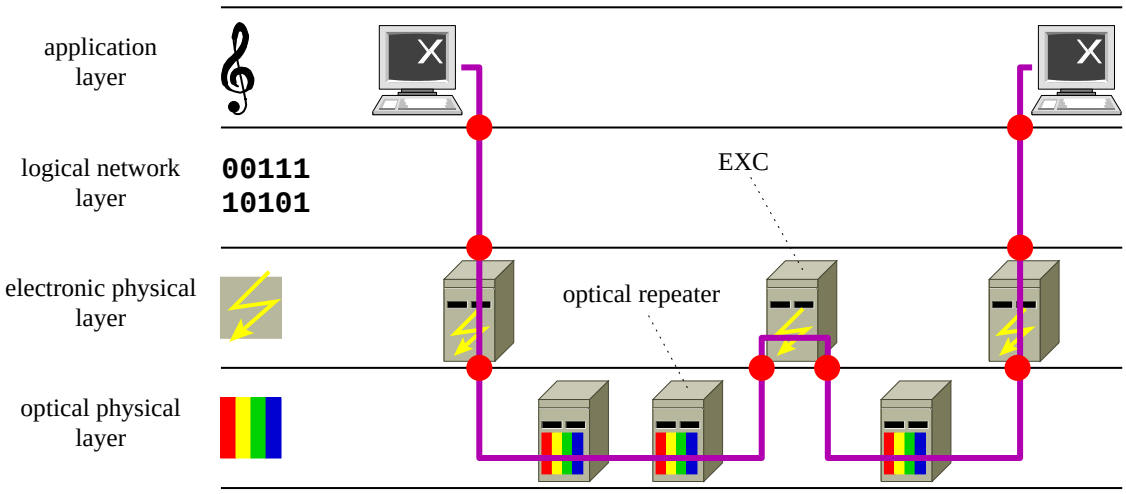


Figure 1.1: Layers of an opaque DWDM network

they slow down the connection routing process and lose the transparency of the optical layer.

DWDM networks using EXC's for routing are called *opaque DWDM* networks. In these networks, the data is transferred in the form of optical signals using optical repeaters. However, all routing functionality is implemented in the electronic layer with the help of EXC's. In the example of Figure 1.1, the first two and the forth internal nodes of the connection path contain optical repeaters forwarding connection only in the optical layer. In contrast, the third one uses an EXC to implement routing functionality. This device converts the signal to the electronic layer and then back to the optical one.

We have to mention here another advantage of EXC's. They are indeed able to use traffic grooming. This is basically a Time Division Multiplexing (TDM) technique which uses the same wavelength of the same fiber to transfer multiple connections. This can be done by defining time-slots and assign one of them to each connection. Though the bandwidth can be highly increased by the use of grooming, this technology is currently available only in the electronic layer, no OXC's are grooming capable.

To summarize, OXC's are faster and they preserve transparency, but their price is much higher than that of EXC's. In contrast, EXC's are cheaper, but they slow down connection routing and lose transparency. As a compromise, we want to build opaque networks where the routing is fully implemented by EXC's, but at the same time we want to use as few of these devices as possible.

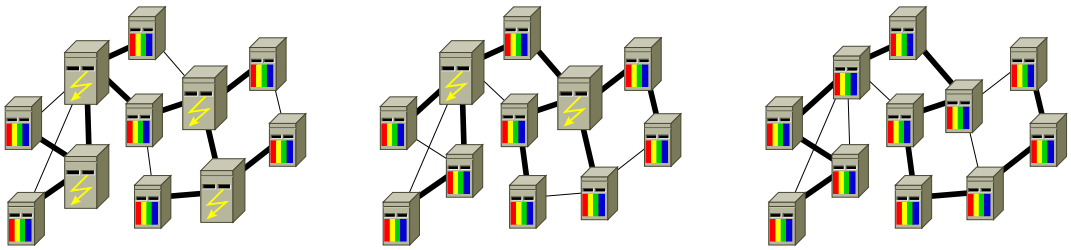
The design problem considered is the following. We have an existing infrastructure composed of network nodes connected by optical fibers. Some of these nodes have a special role: they are connection terminals, that is, they input and output user requests to and from the network. We also have a traffic matrix which shows the

estimated amount of data to be transferred between each pair of terminals. We have to place EXC's to the nodes and route all connections such that the estimated traffic can be sent through the network without major congestion. One can come up with different cost functions including the cost of devices and network links used, routing delays, loss of transparency, etc. If a combination of these cost functions is present, exact mathematical discussion becomes hardly possible though soft-computing techniques can still perform well. In [1] we present a genetic algorithm based approach which deals with many of these cost functions at a time.

In this work, we use a single cost function, that is, we want to minimize the number of EXC's used. Several simplifications will be applied to our model. For example, we suppose that EXC's can handle an infinite number of connections on their ports. The case when the traffic hits the limits of their throughput capacities would need a more sophisticated model and is out of the scope of this work. As per our model we want to build a network where every node can communicate to every other node and there is no need to build protection paths to handle network failures. This means that we must ensure the existence of a single path between each terminal pair.

Our mathematical model is based on graphs. Network nodes are represented by the vertices of a graph G. The existing optical fiber links between nodes give the edges of G. The requirement that each terminal pair must be connected is satisfied by looking for a spanning tree T of G. We can suppose that we need routing functionality and wavelength manipulation only in the network nodes which communicate to at least three adjacent nodes. Therefore we have to put costly switch devices only to these nodes. For our model, this means that our aim is to minimize the number of at-least-3-degree vertices of T [2, 4]. Figure 1.2 shows an example how to decrease the number of EXC's needed in a communication network. According to our assumption, EXC's must be placed exactly to those nodes of whom at least three links are used for communication. To all other nodes, optical repeaters can be installed to forward connections only in the optical layer.

As a result, we are faced to the following combinatorial optimization problem: given a graph G, find a spanning tree of G with a minimum number of branchings (vertices of degree at least 3). This is the Minimum Branching Spanning Tree problem which will be discussed in details in Chapter 3. We mention here, that this problem is a degree-based spanning tree optimization problem, as the measure function is based on the degree distribution of the resulting spanning tree. We will give a formal definition of such problems in Section 1.2. Beside Minimum Branching Spanning Tree , this work discusses a set of closely related degree-based spanning tree optimization problems, too, as they help us to build efficient heuristics for the Minimum Branching Spanning Tree problem. Moreover, as shown in the next subsection, these problems, being the generalizations of the Hamiltonian Path problem, are interesting also from the theoretical point of view.



(a) (b) (c)

Figure 1.2: Eliminating EXC's by connection redesign

1.1.2 Generalizations of the Hamiltonian Path problem

Does a given graph have a simple path containing each vertex exactly once? Though this question is easy to formulate, the answer is hard to give. Indeed, HAMILTONIAN PATH problem, the corresponding decision problem turned out to be NP-complete.

Following a theoretical approach, many necessary or sufficient conditions of hamiltonicity were set up. Most of them are based on structural properties of graphs. We mention a few of these conditions in Chapter 4, where we also present new sufficient conditions of hamiltonicity as part of our theoretical investigation.

Another way to deal with hamiltonicity is to transform the original decision problem to a (necessarily NP-hard) optimization problem. For such a generalization of the HAMILTONIAN PATH problem one can construct approximation algorithms and use them in real-world applications. In this work, we use this approach and consider several spanning tree optimization problems which are all generalizations of the HAMILTONIAN PATH problem. Moreover, we restrict ourselves to so called *degree-based spanning tree optimization problems*, that is, we have an input graph G and we aim to construct a spanning tree T of G such that T optimizes (minimizes or maximizes) a measure function which depends only on the degree distribution of T.

As a first example, we mention the MINIMUM DEGREE SPANNING TREE problem, where the cost of a solution T equals to the highest vertex degree of T. This problem clearly generalizes the HAMILTONIAN PATH problem, as Hamiltonian paths (if exist) are the only spanning trees with a maximum degree of 2. The MINIMUM DEGREE SPANNING TREE problem is easy to approximate. Indeed, Fürer and Raghavachari [29] gave a local improvement based approximation algorithm which finds a spanning tree whose maximum degree is at most one higher than the optimum solution. This result gave us the idea to use local improvement techniques for degree-based spanning tree optimization (see Section 5.5).

The MINIMUM BRANCHING SPANNING TREE problem was first considered by Gargano et al. [33]. In this case, the aim is to find a spanning tree T with a minimum number of branchings (vertices of degree at least 3). Clearly, this is a generalization of the HAMILTONIAN PATH problem, as Hamiltonian paths (if exist) are the only

spanning trees with no branchings. In Chapter 3, we deal with this problem.

Another way of generalizing the HAMILTONIAN PATH problem is to look for a spanning tree with a minimum number of leaves (1-degree vertices). Clearly, every spanning tree must have at least 2 leaves, and Hamiltonian paths (if exist) are the only ones with exactly 2 leaves. This problem, called MINIMUM LEAF SPANNING TREE problem, turned out to be hard to approximate: Lu and Ravi [54] showed that it has no constant factor approximation algorithm, unless P=NP. However, if we complement the measure function and we count the non-leaves (internal vertices) instead of the leaves then the situation is better. The obtained MAXIMUM INTERNAL SPANNING TREE problem is trivially equivalent to the MINIMUM LEAF SPANNING TREE problem as long as we focus on the set of optimal solutions. From an approximation point of view, however, the two problems behave differently. Indeed, the MAXIMUM INTERNAL SPANNING TREE problem can be constant approximated. In Chapter 5, we present a linear-time 2-approximation and a $\mathcal{O}(n^4)$-time 7/4-approximation algorithm for this problem. This latter factor is guaranteed only in graphs with no 1-degree vertices. These algorithms result in a small number of leaves and therefore they can be used as heuristics for the MINIMUM LEAF SPANNING TREE problem.

Clearly, not all degree-based spanning tree optimization problems are generalizations of the HAMILTONIAN PATH problem. As an example, the MAXIMUM LEAF SPANNING TREE problem aims to maximize the number of leaves (so it has the same measure function as the MINIMUM LEAF SPANNING TREE problem, but to be maximized instead of minimized). For this problem Lu and Ravi [54, 55] gave the first constant factor approximation followed by a 2-approximation algorithm of Solis-Oba [64].

1.1.3 Research goals

As mentioned in the above subsections, our research is motivated both by telecommunication network design applications and by theoretical combinatorics. Our goal is to obtain results which can be used in both areas: to build algorithms based on simple steps and then to prove theoretical bounds on their goodness. Beside a mathematical analysis, we also aim to implement some of our algorithms to investigate their behavior on randomly generated inputs.

A part of our research is devoted to graph vulnerability parameters and their connection to spanning tree leaves. This direction relates our work to vulnerability and hamiltonicity theories and becomes a useful tool for proving approximation ratios of our algorithms.

Firstly, we deal with the MINIMUM BRANCHING SPANNING TREE problem (Chapter 3). As we will see (Section 3.2), there is no efficient approximation algorithm for this problem in general graphs. Thus we aim to give an approximation for a subclass of input graphs, namely for evenly dense graphs (Section 3.3), and also to find good heuristics for general graphs. These heuristics are based on the idea of

looking for a spanning tree with only a few 1-degree vertices. Though the number of 1-degree vertices does not determine the number of branchings, decreasing their number helps, in most cases, decreasing the number of branchings, too.

Therefore, our second goal is to deal with degree based spanning tree optimization problems in a bit more general way. We consider the MINIMUM LEAF SPANNING TREE problem (Chapter 5), where the task is to find a spanning tree with a minimum number of leaves. We remark that, behind the scenes, our algorithms (Sections 5.2 and 5.5) will perform the best when they can find a spanning tree with many 2-degree vertices.

Our third goal is to obtain results in the field of vulnerability theory (Chapter 4). Vulnerability parameters measure how much damage can be caused to a graph by removing some of its "important" parts. These parameters have strong connection to hamiltonicity theory and, as we will show, to the number of leaves of spanning trees. Some of our results on the MAXIMUM INTERNAL SPANNING TREE problem in Chapter 5 are based on the theory we present in Chapter 4.

1.2 Problem Statement

In this section, we give the formal definition of the general degree-based spanning tree optimization problem and its special cases which are investigated in this work. First we define what we consider to be an optimization problem.

Definition 1.2.1 *An* optimization problem $\mathcal{P}$ *is characterized by the following quadruple of objects* $(I_{\mathcal{P}}, \mathrm{SOL}_{\mathcal{P}}, m_{\mathcal{P}}, \mathrm{goal}_{\mathcal{P}})$*, where:*

1. $I_{\mathcal{P}}$ *is the set of* instances *of* $\mathcal{P}$*;*
2. $\mathrm{SOL}_{\mathcal{P}}$ *is a function that associates to any input instance* $x \in I_{\mathcal{P}}$ *the set of* feasible solutions *of* x*;*
3. $m_{\mathcal{P}}$ *is the* measure function*, defined for pairs* (x, y) *such that* $x \in I_{\mathcal{P}}$ *and* $y \in \mathrm{SOL}_{\mathcal{P}}(x)$*. For every such pair* (x, y)*,* $m_{\mathcal{P}}(x, y)$ *provides a value in* $\mathbb{Q}$ *which is the value of the feasible solution* y*. This value is also called* cost *in minimization problems and* utility *in maximization problems;*
4. $\mathrm{goal}_{\mathcal{P}} \in \{\mathrm{MIN}, \mathrm{MAX}\}$ *specifies whether* $\mathcal{P}$ *is a minimization or a maximization problem.*

A *spanning tree optimization problem* is to find a spanning tree T of a given undirected connected graph G which, depending on the problem, minimizes or maximizes a measure function $m(.)$. To be more general we allow weights to be put on the vertices and/or on the edges of G. If such weights are present, they can be taken into consideration when calculating $m(T)$. Examples for the spanning tree optimization problems are the MINIMUM WEIGHT SPANNING TREE problem [17, 50, 59],

or the MINIMUM DIAMETER SPANNING TREE problem [39, 40]. For a good survey on spanning tree optimization problems, the reader is referred to [72].

A spanning tree optimization problem is *degree-based* if $m(T)$ depends only on the vertex-degree distribution of T. If G has weights on its vertices then $m(T)$ can also depend on these weights. More precisely, if $d_T(v)$ is the degree of a vertex v in T and $c(v)$ is the weight of vertex v then $m(T)$ is determined by the pairs $(d_T(v); c(v))$. In this thesis, we restrict ourselves to the special case where $m(T)$ can be written in the form of

$$m(T) = \sum_{i=1}^{n-1} f_i \sum \{c(v) : d_T(v) = i\}, \tag{1.1}$$

for some f_i, with $c(v) \equiv 1$ used for the unweighted case.

The MAXIMUM LEAF SPANNING TREE problem [21, 28, 53, 54, 55, 64] is to find a spanning tree with a maximum number of 1-degree vertices, that is, $m(T) = |\{v : d_T(v) = 1\}|$. The MINIMUM DEGREE SPANNING TREE problem [29] is to find a spanning tree whose maximum degree is as small as possible, that is, $m(T) = \min_{v \in T} d_T(v)$. We will see further examples in this section but first let us formalize our above definitions.

Definition 1.2.2 *A* spanning tree optimization problem $\mathcal{P}$ *is an optimization problem* $(I_{\mathcal{P}}, \mathrm{SOL}_{\mathcal{P}}, m_{\mathcal{P}}, \mathrm{goal}_{\mathcal{P}})$*, where:*

1. $I_{\mathcal{P}}$ *is the set of undirected connected (possibly vertex-/edge-weighted) graphs;*
2. $\mathrm{SOL}_{\mathcal{P}}$ *maps any input graph* G *to the set of its spanning trees;*
3. $m_{\mathcal{P}}$ *is the* measure function *that maps a rational number to each solution spanning tree;*
4. $\mathrm{goal}_{\mathcal{P}} \in \{\mathrm{MIN}, \mathrm{MAX}\}$ *specifies whether* $m_{\mathcal{P}}$ *is to be minimized (representing cost) or to be maximized (representing utility).*

The spanning tree optimization problem $\mathcal{P}$ *is* degree-based *if for any solution* T*,* $m_{\mathcal{P}}(T)$ *is fully determined by* $\{(d_T(v); c(v)) : v \in V(T)\}$*, where* $d_T(v)$ *is the degree of a vertex* v *in* T *and* $c(v)$ *is the weight of* v*.*

In what follows we define the degree-based spanning tree optimization problems to be investigated. Their measure function is in the form of (1.1).

Problem 1: MINIMUM LEAF SPANNING TREE
Input: An undirected connected graph G.
Goal: Find a spanning tree T of G with a minimum number of 1-degree vertices (leaves), that is, minimize $m(T) = \|\{v : d_T(v) = 1\}\|$.

Throughout this thesis, we denote by $\mathrm{ml}(G)$ the value of the optimum solution of the MINIMUM LEAF SPANNING TREE problem.

Problem 2: Maximum Forwarding Spanning Tree
Input: An undirected connected graph G.
Goal: Find a spanning tree T of G with a maximum number of 2-degree vertices (forwarding vertices), that is, maximize $m(T) = \|\{v : d_T(v) = 2\}\|$.

Problem 3: Minimum Branching Spanning Tree
Input: An undirected connected graph G.
Goal: Find a spanning tree T of G with a minimum number of (≥ 3)-degree vertices (branchings), that is, minimize $m(T) = \|\{v : d_T(v) \geq 3\}\|$.

Problem 4: Maximum Internal Spanning Tree
Input: An undirected connected graph G.
Goal: Find a spanning tree T of G with a maximum number of (≥ 2)-degree vertices (internal vertices), that is, maximize $m(T) = \|\{v : d_T(v) \geq 2\}\|$.

Problem 5: Maximum Weighted Internal Spanning Tree
Input: An undirected connected graph G with a measure function $c : V(G) \to \mathbb{Q}$ on its vertices.
Goal: Find a spanning tree T of G with a maximum total weight of (≥ 2)-degree vertices (internal vertices), that is, maximize $m(T) = \sum \{c(v) : d_T(v) \geq 2\}$.

As we have already mentioned in Subsection 1.1.2, all of these problems can be viewed as a generalization of the Hamiltonian Path problem, and therefore are NP-hard.

1.3 Overview of Results

The rest of the thesis is organized as follows.

Chapter 2 introduces our notation, gives some basic definitions and summarizes the mathematical background we use throughout the thesis.

Chapter 3 deals with the Minimum Branching Spanning Tree problem. We obtain both positive and negative approximability results. Namely, on one hand we present Algorithm MinBST which yields a spanning tree having at most $3\left\lceil \log_{\frac{1}{1-c}} n \right\rceil + 1$ branchings for evenly dense graphs (of which every vertex has a degree of at least cn), see Theorem 3.3.1. On the other hand, we show that this approximation ratio is very likely the best possible. We give an approximation ratio preserving reduction from the Minimum Set Cover problem to the Minimum Branching Spanning Tree problem thus proving that any ratio better than $\Omega(\log n)$ implies P=NP, see Theorem 3.2.3. Our results discussed in Chapter 3 have originally been published in [2].

Chapter 4 is focusing on our work on the connection of spanning tree leaves and two graph vulnerability parameters: scattering number [44, 74], and cut-asymmetry (Definition 4.3.1). Some of these results are then used in Chapter 5 to build our approximation algorithms for the Maximum Internal Spanning Tree problem. In Section 4.1 we generalize the well-known necessary condition of traceability by proving that every spanning tree of a graph G has at least one more leaf than its scattering number $\mathrm{sc}(G)$ (Theorem 4.1.5). If the graph itself is a tree, its scattering number can be used to upper bound the number of leaves (Theorem 4.1.7). In Section 4.3 we first provide some basic properties of cut-asymmetry $\mathrm{ca}(G)$ of a graph G. Namely, we show that $\mathrm{ca}(G) = 0$ if and only if G is either a complete graph or a cycle (Theorem 4.3.2). We also prove that $\mathrm{ca}(G) \leq 1$ is a sufficient condition for the existence of a Hamiltonian path in G (Theorem 4.3.6). Unfortunately, even a graph with a Hamiltonian path can have a big cut-asymmetry, as shown in Theorem 4.3.7. Later in Section 4.3, we define leaf-independence $\mathrm{li}(G)$ as the maximum number of independent leaves in a spanning tree of G. It turns out that this measure can be considered as an equivalent definition of cut-asymmetry, since $\mathrm{li}(G) = \mathrm{ca}(G) + 1$ always holds (Theorem 4.3.10). Corollary 4.3.13 shows that leaf-independence and the cardinality of a minimum connected vertex cover sums up to n and thus proves that both cut-asymmetry and leaf-independence are NP-hard to compute (Theorem 4.3.14). Our results presented in Chapter 4 have originally been published in [6], in [7], and in [8].

Chapter 5 is about the Maximum Internal Spanning Tree problem. First we present Algorithm ILST that yields a spanning tree with independent leaves, and then we prove that such a spanning tree always forms a 2-approximation for the Maximum Internal Spanning Tree problem (Theorem 5.2.2). In Section 5.2 we provide three different proofs for this fact, one direct proof, one based on the results of Chapter 4, and one based on primal-dual linear programming techniques. Algorithm RDFS, a refined version of Algorithm ILST, has even better approximation properties when applied on special graph classes. It is a 3/2-approximation for claw-free graphs (Theorem 5.4.2) and a 6/5-approximation for cubic graphs (Theorem 5.4.4). In Section 5.5, we develop Algorithm LOST, and using an improved version of the above mentioned linear programming approach, we prove that it is a 7/4-approximation for the Maximum Internal Spanning Tree problem for graphs with no pendant vertices (Theorem 5.5.2). Section 5.6 deals with the case when vertices are weighted and we have to minimize the weighted sum of the branchings of the obtained spanning tree (Maximum Weighted Internal Spanning Tree problem). To solve this problem (for graphs with no pendant vertices), we present Algorithm WLOST which yields a $(2\Delta - 3)$-approximation (Theorem 5.6.2). We then further improve this algorithm obtaining Algorithm RWLOST which is a 2-approximation if the input graph is claw-free (Theorem 5.6.6). In Section 5.7 we consider the Maximum Internal Spanning Tree problem from another point of view: instead of looking for a spanning tree with few leaves, we try to cover as

many vertices as possible with a $\leq q$-leaf subtree. This approach is a generalization of searching for a long path in graphs, see Theorem 5.7.4. In Section 5.8, we show that if any $(q+1)$-element independent set of a claw-free graph (on n vertices) has a degree-sum of at least $n-q$, then the graph has a spanning tree with at most q leaves (Theorem 5.8.1). Finally, in Section 5.9, we present the results of our experimental analysis on our algorithms paying a particular attention to compare the traversals mentioned in Section 2.2. Our results discussed in Chapter 5 have originally been published in [2], in [3], in [5], in [6], and in [8].

Chapter 2

Preliminaries

In this chapter we first introduce our notation and provide definitions for the notions used throughout the thesis. A summary of our notation can be found in Appendix B on page 100. In Section 2.2 we mention a few algorithms (such as Depth First Search) for traversing graphs. These traversals form the basis of our approximation algorithms. Finally, in Sections 2.3 and 2.4 we summarize a few elementary results from the field of approximation theory and linear programming, respectively.

2.1 Notation, Definitions, Properties of Trees

If X is a set and x is an element then $|X|$ stands for the number of elements in X, and $X + x$ and $X - x$ abbreviates $X \cup \{x\}$ and $X \setminus \{x\}$, respectively.

By a *graph* $G = (V, E)$ we mean an undirected simple graph on vertex set $V(G) = V$ and edge set $E(G) = E$. We use $n = |V(G)|$ to denote the number of vertices and $m = |E(G)|$ to denote the number of edges of G. Every graph in this thesis is supposed to be connected unless explicitly stated otherwise.

A *spanning tree* T of a graph G is an acyclic connected subgraph of G containing all of its vertices. Edges of G are called G-edges, edges of T are called T-edges or *tree-edges*, elements of $E(G) \setminus E(T)$ are called *non-tree edges*. Vertices u and v are *G-neighbors* if they are adjacent in G, that is, $(u, v) \in E(G)$, and *T-neighbors* if they are adjacent in T, that is, $(u, v) \in E(T)$. We denote by $N_G(v)$ and $N_T(v)$ the G-neighbors and the T-neighbors of v, respectively. The *G-degree* (*T-degree*) of a vertex v is the number of its G-neighbors (T-neighbors) and is denoted by $d_G(v)$ ($d_T(v)$). When it causes no confusion, we might leave the G out from these notions to abbreviate G-neighbors of v as neighbors of v, or $N(v)$, and G-degree of v as degree of v, or $d(v)$. For some $X \subseteq V$ the notation $d_G(X)$ stands for $\sum_{v \in X} d_G(v)$. A vertex v is called *pendant* if $d_G(v) = 1$. The degree of the highest G-degree vertex is denoted by $\Delta(G)$, or simply by Δ. For a vertex v, $\delta_G(v)$, or simply $\delta(v)$ stands for the set of G-edges incident to v. The graph G is *d-regular* if all of its vertices have G-degree d. A 3-regular graph is also called *cubic*.

This thesis deals with spanning tree optimization problems where the measure to be optimized depends on the degree-distribution of the solution spanning tree. Therefore, the sets of vertices with some special T-degrees play an important role throughout this work. Vertices whose T-degree is exactly i (at least i) form the set $V_i(T)$ ($V_{\geq i}(T)$). A vertex v is a *leaf*, a *forwarding vertex*, or a *branching* of T, if its T-degree is 1, 2, or at least 3, respectively. Equivalently, elements of $V_1(T)$ are the leaves, elements of $V_2(T)$ are the forwarding vertices, and elements of $V_{\geq 3}(T)$ are the branchings of T. Forwarding vertices and branchings are called *internal vertices*, that is, internal vertices are the elements of $V_{\geq 2}(T)$. We also use $L(T)$ and $I(T)$ to denote the set of leaves and internal vertices of T, respectively. As T has no isolated vertices, $V(T) = L(T) \mathbin{\mathring{\cup}} I(T)$, where $\mathring{\cup}$ denotes disjoint union.

If u and v are vertices of a spanning tree T then $P_T(u, v)$ is the unique (u, v) path in T. We denote by $u^{\rightarrow v}$ the vertex succeeding u on the path $P_T(u, v)$.

For the definitions in this paragraph we assume that T is not a Hamiltonian path and that l is a leaf of T. Then the *branching of* l, denoted by $b(l)$, is the branching of T being closest to l. In other words, $b(l)$ is the branching for which all but the end vertices of $P_T(l, b(l))$ are internal vertices of T. The path $P_T(l, b(l))$ itself is called the *branch of* l, and the set of its vertices is denoted by $br(l)$. The vertex preceding $b(l)$ along path $P_T(l, b(l))$ is denoted by $b^-(l)$. This is a simplified notation for $b(l)^{\rightarrow l}$. Notice that the branch of some leaf l might have no internal vertex, in which case $br(l) = \{l, b(l)\}$, and $b^-(l) = l$. The vertices which are neither leaves nor internal vertices of a branch are called *trunk vertices* of T. The tree-edges they span are the *trunk-edges*. These trunk-edges form the *trunk* of T.

A vertex set is said to be *G-independent* (or independent) if it spans no G-edges. Similarly, a vertex set X is *T-independent* if it spans no T-edges. Note that in this latter case X is still allowed to span non-tree edges. The size of the highest cardinality G-independent set is denoted by $\alpha(G)$. The spanning tree T of G is called an *independence tree* if its leaves are G-independent. The notion of independence tree is crucial as it is used throughout the thesis to establish approximation algorithms.

Let X and Y be subsets of $V(G)$. Then $G[X]$ is the subgraph of G spanned by X, $\mathrm{comp}_G(X)$ or simply $\mathrm{comp}(X)$ is the number of components of $G[X]$, $e_G(X)$ is the number of G-edges of $G[X]$, and $e_G(X, Y)$ is the number of G-edges between X and Y. As a special case, $e_G(v, X)$ stands for the number of G-edges between X and a vertex $v \in V(G)$.

A *Hamiltonian path* of a graph is a simple path containing all vertices of the graph and a *Hamiltonian cycle* is a cycle with the same property. If G has a Hamiltonian path, it is called *traceable*. K_n is a complete graph and C_n is a cycle on n vertices, while K_{n_1,n_2} is the complete bipartite graph with color classes of size n_1 and n_2. Particularly, $K_{1,3}$ is called a *claw*. A graph is *claw-free* if it does not contain a $K_{1,3}$ as an induced subgraph.

At last, we remind the reader to some basic properties of trees.

Claim 2.1.1 *For any tree T we have $|V_1(T)| \geq |V_{\geq 3}(T)| + 2$.*

PROOF: On one hand, as T has $|V(T)| - 1$ edges, the total degree of vertices is $D = 2|V(T)| - 2 = 2|V_1(T)| + 2|V_2(T)| + 2|V_{\geq 3}(T)| - 2$. On the other hand, $D \geq |V_1(T)| + 2|V_2(T)| + 3|V_{\geq 3}(T)|$. Putting these together, the claim is immediate. □

Claim 2.1.2 *For any tree T with a maximum degree Δ it holds that $|V_1(T)| - 2 \leq (\Delta - 2)|V_{\geq 3}(T)|$.*

PROOF: For the total degree of vertices, we have

$$D = 2|V(T)| - 2 = 2|V_1(T)| + 2|V_2(T)| + 2|V_{\geq 3}(T)| - 2,$$

and also

$$D \leq |V_1(T)| + 2|V_2(T)| + \Delta|V_{\geq 3}(T)|.$$

Putting these together immediately yields the claim. □

Claim 2.1.3 *For any tree T we have $|V_{\geq 3}(T)| \leq \frac{|V(T)|-2}{2}$.*

PROOF: We have $|V_{\geq 3}(T)| \leq |V(T)| - |V_1(T)| \leq |V(T)| - |V_{\geq 3}(T)| - 2$, where the second inequality comes from Claim 2.1.1 thus finishing the proof. □

2.2 Traversal Algorithms

Traversal algorithms visit the vertices of a given graph one by one and yield a rooted spanning tree whose edges are the ones used for the traversal itself. The two best known such traversal algorithms are Depth First Search (DFS) (see for example [67, p. 538]) and Breadth First Search [67, p. 539]. They both build a spanning tree of the input graph in linear time in the number of edges. In Chapter 5 we use both a generalized and a specialized version of the DFS algorithm in order to build up our approximations for the MAXIMUM INTERNAL SPANNING TREE problem. In this section we summarize the traversal algorithms used throughout the thesis.

First, we introduce the most general one: the greedy traversal. In order to build up a spanning tree this algorithm starts from a root vertex and visits vertices one by one. The only rule is that if the current vertex has some unvisited neighbors then the next vertex to be visited must be one of these, no backtracking is allowed in such situations. Greedy traversal has a non-deterministic behavior. Indeed, there is no rule to specify which unvisited neighbor of the current vertex must be visited next (see Step $(*)$ of Algorithm Greedy Traversal). Also there is no rule to decide where to step back from the current vertex when it has no more unvisited neighbors. Indeed, Algorithm Greedy Traversal uses an edge repository to store some edges that are candidates for becoming the next spanning tree edge. However, the way how it chooses the next edge from the candidates is not specified. Both Algorithm DFS

and Algorithm FIFO-DFS overcome this non-determinism by using their own data structure to implement this edge repository. Algorithm DFS uses a stack (a LIFO store), that is, the elements are fetched from the repository in the reverse order of their insertion. On the contrary, Algorithm FIFO-DFS implements the edge repository with the use of a queue (a FIFO store), that is, elements are fetched in the order of their insertion. Notice that FIFO-DFS differs from the well-known BFS traversal algorithm which is also implemented using a queue. Though these algorithms are more specific than the general Greedy Traversal, they still contain some non-determinism as they do not specify which unvisited neighbor of the current vertex should be chosen next. Algorithm RDFS in Section 5.4 partially solves this problem by giving some rules to support this decision.

More specifically, Algorithm Greedy Traversal works as follows. The input is a simple connected graph G. We begin with an empty graph T on $V(G)$. As we traverse G, we add more and more G-edges to T, until T becomes a spanning tree. For each vertex v, we maintain its rank in the order of visiting (arriving first time to the vertex) in the variable $\mathsf{VisitingRank}[v]$. This rank is initialized to and remains 0 until v gets visited. Variable EdgeRepository is used for storing the edges being candidates to be processed next. The traversal starts by visiting an arbitrarily chosen root vertex r. When a vertex v is visited (function `VisitVertex`), we check whether v has unvisited neighbors. If this is the case, we choose one of them as the next vertex to visit and add the corresponding edge to T. Otherwise, we cannot continue the traversal from v, we step back, that is, we pick an element from EdgeRepository and add it to T if no circle arises. Note that in the general Greedy Traversal, this "pick" operation gives an arbitrary element of EdgeRepository, while in Algorithm DFS and Algorithm FIFO-DFS, the operation is deterministic. Before finishing the visiting of v, we add to EdgeRepository all those G-edges incident to v which lead to an unvisited vertex.

Let us give some basic traversal related definitions. As we mentioned earlier, the output of the traversal algorithms is a spanning tree T rooted at vertex r. Each vertex $v \neq r$ has a unique *parent* which is the vertex preceding v along path $P_T(r, v)$. If w is the parent of v then v is called a child of w. Based on these one-level descendant relations, we straightforwardly define the *descendents* and *ancestors* of a vertex. More precisely, descendents of a vertex w are all the vertices (excluding w itself) in the subtree of w. Ancestors of $v \neq r$ are all the vertices (excluding v itself) of the path $P_T(r, v)$. A vertex with no child is called a *d-leaf* of T. It is important to see the difference between the d-leaves of a spanning tree T rooted in vertex r, and the leaves of its non-rooted version T'. All d-leaves of the rooted tree T are leaves of the non-rooted tree T' as their T'-degree is 1. However, if $d_T(r) = 1$, that is, the root has a single child in T, then r is a leaf of T' without being a d-leaf of T. We will have to pay attention to this difference in Section 5.2 when using Algorithm DFS to build up an algorithm which creates an independence tree (see Algorithm ILST).

Algorithm Greedy Traversal

Input: A simple connected graph G
Output: A rooted spanning tree T of G (called greedy traversal tree)
begin
  $T \leftarrow (V, \emptyset)$
  foreach $v \in V(G)$ **do**
    $\mathsf{VisitingRank}[v] \leftarrow 0$
  $\mathsf{NextVisitingRank} \leftarrow 1$
  $\mathsf{EdgeRepository} \leftarrow \emptyset$
  $r \leftarrow$ an arbitrary vertex of G
  `VisitVertex(`r`)`
  return T
end

`// Traversal from a vertex` v
function `VisitVertex(`v`)`
begin
  $\mathsf{VisitingRank}[v] \leftarrow \mathsf{NextVisitingRank}$
  $\mathsf{NextVisitingRank} \leftarrow \mathsf{NextVisitingRank} + 1$
* **if** *v has a neighbor w such that* $\mathsf{VisitingRank}[w] = 0$ **then**
    $x \leftarrow v$; $y \leftarrow w$
  else
    if EdgeRepository *is not empty* **then**
      $(x, y) \leftarrow$ the "first" element of EdgeRepository such that $\mathsf{NextVisitingRank}[y] = 0$
    else
      return
  Add (x, y) to T
  foreach *neighbor w of x such that $w \neq y$ and* $\mathsf{NextVisitingRank}[w] = 0$ **do**
    Put (x, w) to EdgeRepository
  `VisitVertex(`y`)`
end

Algorithm DFS (Depth First Search)

Input: A simple connected graph G
Output: A rooted spanning tree T of G (called DFS-tree)
begin
  Run Algorithm Greedy Traversal using a stack (LIFO store) as EdgeRepository.
end

Algorithm FIFO-DFS (First In First Out DFS)
Input: A simple connected graph G
Output: A rooted spanning tree T of G (called FIFO-DFS-tree)
begin
  Run Algorithm Greedy Traversal using a queue (FIFO store) as EdgeRepository.
end

Claim 2.2.1 *Let T be a greedy traversal tree of graph G. Then the d-leaves of T form a G-independent set.*

PROOF: Suppose for a contradiction that there are two d-leaves l_1 and l_2 of T which are G-neighbors. Let l_1 be the one first visited by the traversal. As l_1 is a d-leaf we had to step back after visiting it. We step back from a vertex only if it has no unvisited neighbors (see function `VisitVertex` of Algorithm Greedy Traversal). However, at the moment when we step back from l_1 its neighbor l_2 is still unvisited. This forms a contradiction. □

Now let us recall another basic property of Depth First Search. Note that this is not true for all greedy traversal trees in general.

Claim 2.2.2 *Let T be a spanning tree of an undirected graph G output by Algorithm DFS. Then each G-edge connects two vertices of which one is an ancestor of the other in T.*

PROOF: Suppose for a contradiction that G has an edge (u, v) such that neither u nor v is an ancestor of the other. Without loss of generality we can say that u was visited before v. Among the common ancestors of u and v there is a unique one which is a descendant of all others. Let us denote this vertex by x. As EdgeRepository is a stack, we have to finish the traversal of u before stepping back to x and continue our traversal towards v. This means that v is still unvisited when we step back from u. This, however, forms a contradiction as u cannot have more unvisited neighbors at that moment. □

Observe that Algorithm Greedy Traversal uses a general EdgeRepository which does not guarantee that the traversal of u is finished before stepping back to v. It might happen that u still has unvisited neighbors (v among others) when we step back to x and reach v from that alternative direction.

The traversal algorithms mentioned in this section can all be used in Algorithm LOST (see Section 5.5) as a building block for finding an initial spanning tree. As part of our research, we have done an experimental analysis to demonstrate how the choice of the traversal influences the number of leaves of the output spanning tree. The results of this analysis is discussed in Section 5.9.

2.3 Optimization and Approximation

In this section, we summarize the concepts of approximation algorithms, approximation ratios and approximation ratio preserving reductions. For further details on approximation theory, the reader is referred to monographs [10] and [68]. We follow the notation of the former one.

It is worth mentioning here that our negative approximability results are based on the assumption that P $\neq$ NP.

Now recall Definition 1.2.1. An optimization problem $\mathcal{P}$ is characterized by the set of input instances, the corresponding set of solutions, and the measure function to be minimized or maximized. According to our notation, $\mathcal{P}$ is a quadruple of objects $(I_{\mathcal{P}}, \mathrm{SOL}_{\mathcal{P}}, m_{\mathcal{P}}, \mathrm{goal}_{\mathcal{P}})$.

Given an input instance x, we denote by $\mathrm{SOL}^*_{\mathcal{P}}(x)$ the set of *optimal solutions* of x, that is, the set of solutions whose value is optimal (minimum or maximum depending on whether $\mathrm{goal}_{\mathcal{P}} = \mathrm{MIN}$ or $\mathrm{goal}_{\mathcal{P}} = \mathrm{MAX}$). More formally, for every $y^*(x)$ such that $y^*(x) \in \mathrm{SOL}^*_{\mathcal{P}}(x)$:

$$m_{\mathcal{P}}\left(x, y^*(x)\right) = \mathrm{goal}_{\mathcal{P}}\left\{v : v = m_{\mathcal{P}}(x, z) \wedge z \in \mathrm{SOL}_{\mathcal{P}}(x)\right\}.$$

The value of an optimal solution $y^*(x)$ of x is denoted by $m^*_{\mathcal{P}}(x)$.

Once we have an optimal solution we can compare any other solution y against it in order to set up a performance ratio for y. This ratio shows how good solution y is.

Definition 2.3.1 *Given an optimization problem $\mathcal{P}$, for any instance x of $\mathcal{P}$ and for any feasible solution y of x, the* performance ratio *of y with respect to x is defined as*

$$R(x, y) = \frac{m(x, y)}{m^*(x)}$$

for minimization problems, and as

$$R(x, y) = \frac{m^*(x)}{m(x, y)}$$

for maximization problems.

Using this, we can now define the approximation ratio of an algorithm.

Definition 2.3.2 *Given an optimization problem $\mathcal{P} = (I_{\mathcal{P}}, \mathrm{SOL}_{\mathcal{P}}, m_{\mathcal{P}}, \mathrm{goal}_{\mathcal{P}})$, an algorithm $\mathcal{A}_{\mathcal{P}}$ for $\mathcal{P}$ returns a feasible solution $\mathcal{A}_{\mathcal{P}}(x) \in \mathrm{SOL}_{\mathcal{P}}(x)$ for any given instance $x \in I_{\mathcal{P}}$. We say that $\mathcal{A}_{\mathcal{P}}$ is an $f(x)$-*approximation algorithm *for $\mathcal{P}$ if, given any input instance x of $\mathcal{P}$*

1. *$R\left(x, \mathcal{A}_{\mathcal{P}}(x)\right) \leq f(x)$, and*
2. *$\mathcal{A}_{\mathcal{P}}$ runs in polynomial time (in the size of x).*

Such an $f(x)$ is also called a performance ratio*, or an* approximation ratio *of $\mathcal{A}_\mathcal{P}$. If $f(x)$ is constant it is called an* approximation factor *of $\mathcal{A}_\mathcal{P}$.*

A whole hierarchy of approximability classes is built up similarly to those of complexity classes [10, 68]. These classes can be used to obtain limits for approximability of optimization problems. In this thesis, we use only one of these classes: APX, the class of constant factor approximable problems.

Definition 2.3.3 APX *is the class of all NP optimization problems $\mathcal{P}$ such that for some constant $r \geq 1$, there exists an r-approximation algorithm for $\mathcal{P}$.*

In order to give negative approximability results, we need a reduction, analogous to Karp-reduction [30], which preserves the approximability properties.

Definition 2.3.4 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two NP optimization problems. $\mathcal{P}_1$ is said to be* AP-reducible *to $\mathcal{P}_2$ if there exist two functions f, g and a constant $\kappa \geq 1$ such that:*

1. *For any instance $x \in I_{\mathcal{P}_1}$ and for any rational number $r > 1$, $f(x, r) \in I_{\mathcal{P}_2}$;*
2. *For any instance $x \in I_{\mathcal{P}_1}$ and for any rational number $r > 1$, if $\mathrm{SOL}_{\mathcal{P}_1}(x) \neq \emptyset$ then $\mathrm{SOL}_{\mathcal{P}_2}(f(x, r)) \neq \emptyset$;*
3. *For any instance $x \in I_{\mathcal{P}_1}$, for any rational number $r > 1$ and for any $y \in \mathrm{SOL}_{\mathcal{P}_2}(f(x, r))$ it holds that $g(x, y, r) \in \mathrm{SOL}_{\mathcal{P}_1}(x)$;*
4. *f and g are computable by algorithms whose running time is polynomial for any fixed r;*
5. *For any instance $x \in I_{\mathcal{P}_1}$ and for any rational number $r > 1$, and for any $y \in \mathrm{SOL}_{\mathcal{P}_2}(f(x, r))$,*

$$R_{\mathcal{P}_2}(f(x, r), y) \leq r \text{ implies } R_{\mathcal{P}_1}(x, g(x, y, r)) \leq 1 + \kappa(r - 1).$$

The triple (f, g, κ) is said to be an AP-reduction *from $\mathcal{P}_1$ to $\mathcal{P}_2$.*

The acronym stands for *approximation preserving* and is justified by the following result showing how to use AP-reducibility to prove that an approximability problem is in APX.

Theorem 2.3.5 [10] *If $\mathcal{P}_1$ is AP-reducible to $\mathcal{P}_2$ and $\mathcal{P}_2 \in APX$ then $\mathcal{P}_1 \in APX$.*

We use this result in Section 3.2 to prove a negative approximability result, namely that the Minimum Branching Spanning Tree problem is not in APX. For this purpose, we give an AP-reduction to it from the Minimum Set Cover problem.

2.4 Linear Programming

In this section we remind the reader a basic result of linear programming which we use in Chapter 5 in order to prove approximation ratios.

Let $x_0 \in \mathbb{R}^{k_0}$, $x_1 \in \mathbb{R}^{k_1}$, $y_0 \in \mathbb{R}^{l_0}$, and $y_1 \in \mathbb{R}^{l_1}$ be variable vectors, $b_0 \in \mathbb{R}^{l_0}$, $b_1 \in \mathbb{R}^{l_1}$, $c_0 \in \mathbb{R}^{k_0}$, and $c_1 \in \mathbb{R}^{k_1}$ constant vectors. Furthermore let us have matrices $\mathrm{A} \in \mathbb{R}^{l_0 \times k_0}$, $\mathrm{B} \in \mathbb{R}^{l_0 \times k_1}$, $\mathrm{C} \in \mathbb{R}^{l_1 \times k_0}$, and $\mathrm{D} \in \mathbb{R}^{l_1 \times k_1}$. We consider the set of solutions of two linear inequality systems:

$$\mathcal{P} = \{(x_0, x_1) : \mathrm{A}x_0 + \mathrm{B}x_1 = b_0, \mathrm{C}x_0 + \mathrm{D}x_1 \leq b_1, x_1 \geq 0\},$$

and

$$\mathcal{D} = \{(y_0, y_1) : y_0\mathrm{A} + y_1\mathrm{C} = c_0, y_0\mathrm{B} + y_1\mathrm{D} \geq c_1, y_1 \geq 0\}.$$

The solution sets $\mathcal{P}$ and $\mathcal{D}$ are called *polyhedra*. Based on them we define two linear programming problems. The so-called primal problem is in the form of

$$\mathrm{val_p} = \max\{(c_0x_0 + c_1x_1) : (x_0, x_1) \in \mathcal{P}\}. \tag{2.1}$$

The dual problem is

$$\mathrm{val_d} = \min\{(b_0y_0 + b_1y_1) : (y_0, y_1) \in \mathcal{D}\}. \tag{2.2}$$

It might happen that either of these problems have no solution, that is, either polyhedron $\mathcal{P}$ or polyhedron $\mathcal{D}$ is empty. The following theorem (the Duality Theorem of linear programming) claims that if neither of the two polyhedra is empty then the optimum solutions of the primal and the dual programs are exactly the same.

Theorem 2.4.1 (Duality Theorem of linear programming [61, p. 90]) *Let us have the above defined non-empty polyhedra $\mathcal{P}$ and $\mathcal{D}$. Then the maximum defined in* (2.1) *and the minimum defined in* (2.2) *are equal, that is* $\mathrm{val_p} = \mathrm{val_d}$.

Subsection 5.2.3 shows how to apply this theorem to build proofs for approximation ratios. In fact, we only use that every primal solution is upper bounded by any dual solution. See [36, 42, 65] for proofs of approximation ratios using a similar technique. In Section 5.5 we build a 7/4-approximation algorithm for Maximum Internal Spanning Tree problem for graphs with no pendant vertices and prove its approximation ratio using this primal-dual method.

Chapter 3

Minimizing the Number of Branchings

In this chapter we investigate the MINIMUM BRANCHING SPANNING TREE problem which aims to find a spanning tree T of a given input graph G such that T has a minimum number of branchings among all spanning trees of G. As spanning trees with no branchings are exactly the Hamiltonian paths of G, we cannot expect exact polynomial-time solution for this problem. Instead, we deal with approximation algorithms. Unfortunately, as we show in Section 3.2, the MINIMUM BRANCHING SPANNING TREE problem is hard even to approximate. An approximation ratio preserving reduction from the MINIMUM SET COVER problem proves that any approximation ratio better than $\Omega(\log n)$ would imply P=NP. In Section 3.3 we achieve the approximation ratio of $\mathcal{O}(\log n)$ for a special class of input graphs, the evenly dense non-traceable graphs. A greedy strategy yields a spanning tree with at most $\mathcal{O}(\log n)$ branchings whenever each vertex has a degree of $\Omega(n)$.

The chapter is organized as follows: in Section 3.1 we summarize the known results on the MINIMUM BRANCHING SPANNING TREE problem; in Section 3.2 we present our negative approximability result; finally in Section 3.3 we give an approximation algorithm for evenly dense non-traceable graphs.

3.1 Related Results

The MINIMUM BRANCHING SPANNING TREE problem was first considered by Gargano et al. [33] as a degree-based generalization of the HAMILTONIAN PATH problem. They have proved that it is NP-complete to decide whether a spanning tree with at most k branchings exists (for any fixed k). They have also given an algorithm [32] that finds a single-branching spanning tree (a so called spanning spider) if each 3-element independent set of the input graph G has a degree sum of at least $|V(G)| - 1$. The case when the input graph is bipartite is discussed in [31]. Flandrin et al. [27] considered the problem of finding a spanning spider having its branching fixed in advance. They proved that a graph G has a spanning spider whenever the sum of its minimum and maximum vertex degree is at least $|V(G)|$.

We have to mention here the MINIMUM CONNECTED DOMINATING SET problem where, given an input graph $G = (V, E)$, the aim is to find a minimum cardinality subset S of V such that each vertex is either in S or it has a neighbor in S, and that S spans a connected subgraph of G [30]. From a connected dominating set S, we can always build a spanning tree with at most $|S|$ branchings. Indeed, we start with a spanning tree of $G[S]$ and join all vertices of $V \setminus S$ to it. All these vertices will be leaves of our spanning tree, as implied by the fact that S is a connected dominating set. However, if we have a spanning tree with $|V_{\geq 3}|$ branchings, we cannot always bound the cardinality of a minimum connected dominating set by the means of $|V_{\geq 3}|$. For the MINIMUM CONNECTED DOMINATING SET problem Guha and Khuller gave an approximation algorithm with an approximation ratio of $\mathcal{O}(\log n)$ [37, 38]. Our algorithm shows some similarity to, and is inspired by, their result. However, there are many differences to be emphasized including the problem formulation and the objective function itself. While Guha and Khuller use their algorithm to prove a multiplicative approximation ratio on the cardinality of a connected dominating set, we always find a spanning tree with at most $\mathcal{O}(\log n)$ branchings in evenly dense graphs. Both our algorithm and that of Guha and Khuller build the solution iteratively, by adding new vertices to the spanning tree / dominating set under construction. However, our proof of the approximation ratio is new, as that of Guha and Khuller for the MINIMUM CONNECTED DOMINATING SET problem cannot be adapted to the MINIMUM BRANCHING SPANNING TREE problem.

3.2 A Negative Approximability Result

In this section we present our main negative approximability result on the MINIMUM BRANCHING SPANNING TREE problem by giving an approximation ratio preserving reduction from the MINIMUM SET COVER problem. Recall the following definition [43]:

Problem 6: MINIMUM SET COVER
Input: A ground set $\mathcal{S}$ and a set $\Sigma = \{S_j\}_{j=1}^{s}$ of its subsets.
Goal: Find a minimum number of subsets of Σ whose union contains each element of $\mathcal{S}$.

Regarding this problem, Alon et al. [9] proved the following:

Theorem 3.2.1 *The* MINIMUM SET COVER *problem is not approximable better than a multiplicative ratio of* $\Omega(\log |\mathcal{S}|)$*, unless P=NP, that is, the* MINIMUM SET COVER *problem is not in APX.*

In what follows we construct the AP-reduction (f, g, κ) from the MINIMUM SET COVER problem to the MINIMUM BRANCHING SPANNING TREE problem. For this purpose we define f, g, and κ, and prove that they satisfy the five criteria of AP-reducibility, see p. 19 for the notation.

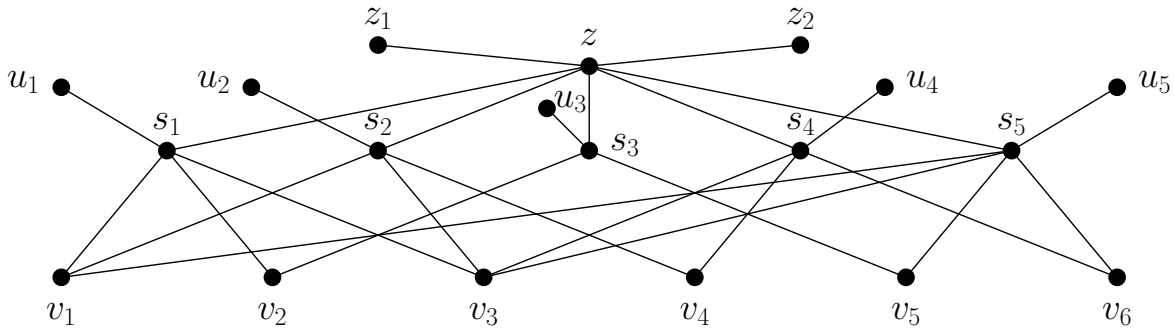


Figure 3.1: Reduction of the MINIMUM SET COVER problem to the MINIMUM BRANCHING SPANNING TREE problem. The graph derived from set $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$, and subsets $\Sigma = \{\{1, 2, 3\}, \{1, 3, 4\}, \{2, 5\}, \{3, 4, 6\}, \{1, 3, 5, 6\}\}$

Given an instance $x = (\mathcal{S}, \Sigma)$ of the MINIMUM SET COVER problem, we construct the graph $f(x, r) = f(x) = G$ as follows (Fig. 3.1).

For each element e_i of $\mathcal{S}$ we take a vertex v_i. For each subset S_j in Σ we take a vertex s_j. We connect s_j and v_i by an edge if and only if $e_i \in S_j$. We also take additional vertices $\{u_j : j = 1, \ldots, |\Sigma|\}$, z, z_1, and z_2. Then we add an edge between each u_j and its s_j pair. Furthermore, we add an edge between z and each s_j. Finally we add edges (z_1, z) and (z_2, z).

Criterion 1 of the AP-reducibility is satisfied by this construction, as G is an instance of the MINIMUM BRANCHING SPANNING TREE problem. Also, Criterion 2 is held, since G is connected (and so has a spanning tree) if the original instance x had a solution. Moreover, we show now how the value of solutions of the two problems are related.

Let us suppose that a cover M of $\mathcal{S}$ contains k elements $S_{j_1}, S_{j_2}, \ldots S_{j_k}$ of Σ. We construct a spanning tree T of G having $k + 1$ branchings (solution y of the MINIMUM BRANCHING SPANNING TREE instance $f(x)$). For this aim, we add to T every G-edge incident to $s_{j_1}, s_{j_2}, \ldots,$ or s_{j_k}. As M is a cover of $\mathcal{S}$, now we have $d_T(v_i) \geq 1$ for every i. We then drop some of these edges from T, if necessary, in order to set the degree of each vertex v_i to exactly 1. Finally, we add to T each G-edge incident to z, and also edges (s_j, u_j) for every j. T is now a spanning tree of G.

This construction ensures that all vertices v_i, u_j, z_1, and z_2 are leaves of T while z is a branching of T. The most interesting part is the T-degree of vertices s_j. For each vertex s_j we have $d_T(s_j) = 2$ if and only if S_j is not used in the set cover M. Otherwise, s_j is a branching of T. As a result, T has exactly $k + 1$ branchings.

Now to see the other direction, let us have a spanning tree T of G with b branchings. Vertex z must be a branching of T as removing it splits G to at least 3 components. Each vertex s_j has an edge to u_j. This implies that each vertex v_i must have at least one branching neighbor (among s_j's). Otherwise v_i and z would

be in separate components of T, which would give a contradiction as T is a spanning tree. As a result, the set M of sets $S_{j_1}, S_{j_2}, \dots S_{j_p}$ corresponding to branchings $s_{j_1}, s_{j_2}, \dots s_{j_p}$ of T is a set cover of $\mathcal{S}$. Since z is a branching we have a set cover of size $p \leq b-1$. This mapping from the solutions of the MINIMUM BRANCHING SPANNING TREE problem on G to the solutions of the MINIMUM SET COVER problem is the function g of our AP-reduction. Clearly, Criterion 3 is satisfied by the construction.

Observe that applying the two aforementioned mappings on optimum solutions shows that the value of the optimum solution for instance x (of the MINIMUM SET COVER problem) is exactly one less than the value of the optimum solution for instance $G = f(x)$ (of the MINIMUM BRANCHING SPANNING TREE problem).

To check Criterion 4 of AP-reducibility, we have to observe that the construction of G needs only polynomial time (in size of x). The polynomial-time computability of g is trivial.

To prove that Criterion 5 is satisfied, let $m^*(f(x,r))$ denote the value of the optimum solution of instance $f(x,r)$, and $m(f(x,r),y)$ denote the value of solution y. Also let $m^*(x)$ be the value of the optimum solution for instance x, and $m(x, g(x,y,r))$ be the value of solution $g(x,y,r)$.

Then suppose

$$r \geq R_{\text{MINIMUM BRANCHING SPANNING TREE}}(f(x,r),y) = \frac{m(f(x,r),y)}{m^*(f(x,r))} \tag{3.1}$$

for some $r > 1$ and fixed $\kappa = 2$.

We have to show that

$$1 + 2(r-1) \geq R_{\text{MINIMUM SET COVER}}(x, g(x,y,r)) = \frac{m(x, g(x,y,r))}{m^*(x)}.$$

The above mappings between the solutions yield that $m^*(x) = m^*(f(x,r)) - 1$ and that $m(x, g(x,y,r)) = m(f(x,r),y) - 1$. Using this, we are to show that

$$1 + 2(r-1) \geq \frac{m(f(x,r),y) - 1}{m^*(f(x,r)) - 1},$$

or equivalently

$$2rm^*(f(x,r)) - m^*(f(x,r)) - 2(r-1) \geq m(f(x,r),y).$$

Using Inequality (3.1), it is enough to show that

$$rm^*(f(x,r)) - m^*(f(x,r)) - 2(r-1) \geq 0,$$

that is,

$$(r-1)(m^*(f(x,r)) - 2) \geq 0.$$

This inequality is true for all $r > 1$, since every spanning tree of G has at least two branchings (z and at least one of s_i's).

Thus, we conclude that the above defined $(f, g, 2)$ is an AP-reduction from Minimum Set Cover problem to Minimum Branching Spanning Tree problem.

Now Theorem 3.2.1 and Theorem 2.3.5 imply:

Theorem 3.2.2 *The* Minimum Branching Spanning Tree *problem is not in APX.*

Moreover, if we set $r = \Omega(\log |V(G)|)$ in the above calculations then using Theorem 3.2.1 we get

Theorem 3.2.3 *The* Minimum Branching Spanning Tree *problem is not approximable better than a multiplicative ratio of* $\Omega(\log |V(G)|)$*, unless P=NP.*

3.3 Approximation in Evenly Dense Graphs

We have seen in the last section that the Minimum Branching Spanning Tree problem very unlikely has an approximation algorithm with a ratio better than $\Omega(\log n)$. In this section we present the first approximation algorithm which achieves this approximation ratio whenever the input graph is non-traceable and evenly dense, that is, all of its vertices have a degree of $\Omega(n)$. More precisely, in evenly dense graphs our algorithm produces a spanning tree with $\mathcal{O}(\log n)$ branchings.

Given an input graph G, our approximation algorithm starts with an empty graph $H = (V, \emptyset)$. Then it subsequently adds G-edges to H, until H becomes a spanning forest without isolated vertices. In each iteration of this spanning forest building process, we greedily select a vertex v such that there is a maximum number of isolated vertices of H among the G-neighbors of v. Then we add to H all G-edges which connects v to an isolated vertex of H. When H has no more isolated vertices, some additional G-edges are used to connect the components of H forming the output spanning tree.

During the spanning forest building process we maintain three disjoint vertex-sets. C contains the vertices which have already been selected, B contains those G-neighbors of vertices in C that are not in C, that is, $B = N_G(C) \setminus C$, and $A = V \setminus (C \cup B)$ contains all other vertices of G. At the beginning, every vertex is in A. At each iteration one vertex moves to C from A or B and some vertices may move from A to B. The algorithm guarantees that elements of A are exactly the isolated vertices of H. Thus the first phase runs as long as A has some elements.

In the description of Algorithm MinBST, we use a subscript i to denote the value of a variable *before* iteration i. This means that during iteration i the algorithm selects vertex v_i from $A_i \cup B_i$ and moves it to C_{i+1}, thus $C_{i+1} = C_i + v_i$. All neighbors of v_i being in A_i are moved to B_{i+1}, thus $B_{i+1} = B_i - v_i \cup (N(v_i) \cap A_i)$. Finally,

$A_{i+1} = A_i \setminus (v_i + N(v_i))$. Iteration i also adds some edges to the current spanning forest H_i yielding a new forest H_{i+1}.

In iteration i our algorithm selects a vertex v_i which maximizes $e_G(v_i, A_i)$ among all vertices of $A_i \cup B_i$. Particularly, the first vertex v_1 to be selected is a highest degree vertex of G. After selecting v_i, we add to H_i all G-edges connecting v_i to A_i. In the implementation of Algorithm MinBST we use array ADegree to ease the calculation of $e_G(v, A_i)$. Namely, during the i^{th} iteration ADegree$[v]$ is equal to $e_G(v, A_i)$. It is initialized to $e_G(v, A_1) = d_G(v)$, and is updated after each iteration.

Algorithm MinBST (Minimum Branching Spanning Tree)

Input: A simple connected graph G
Output: A spanning tree T of G
Initialization:
begin
  $H_1 \leftarrow (V, \emptyset)$
  $A_1 \leftarrow V$
  $B_1 \leftarrow \emptyset$
  $C_1 \leftarrow \emptyset$
  foreach $v \in V(G)$ **do** ADegree$[v] \leftarrow d_G(v)$
  $i \leftarrow 1$
end
First phase: building a forest
begin
  while $A_i \neq \emptyset$ **do**
    $v_i \leftarrow$ the vertex $v \in A_i \cup B_i$ which maximizes $e_G(v, A_i)$
    $C_{i+1} \leftarrow C_i + v_i$
    $B_{i+1} \leftarrow B_i - v_i \cup (N_G(v_i) \cap A_i)$
    $A_{i+1} \leftarrow V \setminus (B_{i+1} \cup C_{i+1})$
    $E' \leftarrow \delta_{G[A_i + v_i]}(v_i)$
    $H_{i+1} \leftarrow H_i + E'$
    foreach $v \in A_i \setminus A_{i+1}$ **do**
      foreach $w \in N(v) \cap (A_{i+1} \cup B_{i+1})$ **do**
        ADegree$[w] \leftarrow$ ADegree$[w] - 1$
    $i \leftarrow i + 1$
end
Second phase: connecting the components
begin
  `// Add edges from` G `to` H_i `to obtain a spanning tree`
  `JoinTrees(`G, H_i`)`
end

The second phase runs a traversal on G using the already existing components

of the forest H. Whenever a G-edge e connects two different components, we add it to H.

Second phase of Algorithm MinBST: connecting components

```
function JoinTrees(G, H)
Input: A simple connected graph G and its subforest H
Output: A spanning tree T of G
begin
    foreach v ∈ V(G) do Marked[v] ← 0
    S ← ∅
    Choose an arbitrary vertex v_0
    Run a traversal on the component of v_0 in H
    when visiting a vertex x set Marked[x] ← 1 and put x into S
    while S ≠ ∅ do
        Let v be any element of S
        Remove v from S
        if v has a G-neighbor w such that Marked[w] = 0 then
            Add (v, w) to H
            Run a traversal on the component of w in H
            when visiting vertex x set Marked[x] ← 1 and put x into S
end
```

Theorem 3.3.1 is the main result of this chapter. It states that Algorithm MinBST always finds a spanning tree with $\mathcal{O}(\log n)$ branchings in evenly dense graphs. $\mathcal{O}(\log n)$ is a valid approximation ratio only if the input graph is not traceable as otherwise the value of the optimum solution is 0. First we prove the bound on the number of branchings then we analyze the time complexity of the algorithm.

Theorem 3.3.1 *Let G be a connected graph on n vertices and m edges. If the G-degree of each vertex is at least cn (for some number $c \in \mathbb{R}$) then Algorithm MinBST yields a spanning tree with at most $3\left\lceil \log_{\frac{1}{1-c}} n \right\rceil + 1$ branchings in $\mathcal{O}(m + n \log n)$ time.*

Let p denote the number of iterations executed in the first phase. At first, we show a few basic properties of the sets A_i, B_i, and C_i.

Claim 3.3.2 *For the sets A_i, B_i, and C_i (for $2 \leq i \leq p$) of Algorithm MinBST, followings are trivially true:*

1. *A_i is an H_i-independent set;*
2. *B_i is an H_i-independent set;*
3. *there is no G-edge between A_i and C_i;*

4. $|C_i| = i$;

5. $\mathrm{comp}_{H_i}(B_i \cup C_i) \leq i$;

6. *for all* $v \in B_i$ *we have* $d_{H_i}(v) = 1$.

We decompose the proof of Theorem 3.3.1 to several lemmas.

Lemma 3.3.3 *For all* $1 \leq i \leq p$ *we have* $e_G(v_i, A_i) \geq c|A_i|$.

PROOF: Let

$$k_i = \frac{|A_i|nc}{n-i}.$$

Suppose for a contradiction that

$$k_i > e_G(v_i, A_i).$$

Then we obtain

$$k_i|A_i| \stackrel{1}{>} \sum_{w \in A_i} e_G(w, A_i) \stackrel{2}{=} \sum_{w \in A_i} d(w) - \sum_{w \in A_i} e_G(w, B_i) \stackrel{3}{\geq}$$
$$|A_i|nc - \sum_{w \in A_i} e_G(w, B_i) \stackrel{4}{=} |A_i|nc - \sum_{v \in B_i} e_G(v, A_i) \stackrel{5}{>} |A_i|nc - |B_i|k_i.$$

Here we have used the fact that v_i maximizes $e_G(v, A_i)$ for Inequalities 1 and 5; Claim 3.3.2/3 for Equality 2; the fact that every vertex has a G-degree of at least nc for Inequality 3; and a double counting of the edges between A_i and B_i for Equality 4.

Thus

$$|A_i|nc < (|A_i| + |B_i|)\, k_i = (|V| - |C_i|)\, k_i = (n-i)k_i = |A_i|nc$$

gives a contradiction, and so proves the lemma as

$$e_G(v_i, A_i) \geq k_i = \frac{|A_i|nc}{n-i} \geq c|A_i|.$$

□

Lemma 3.3.4 *For all* $2 \leq i \leq p$ *we have* $|A_i| \leq (1-c)^{i-2}(n - \Delta - 1)$.

PROOF: We use induction to prove the lemma. Recall that $A_1 = V$. Clearly, one of the highest G-degree vertices is selected to be v_1 and so

$$|A_2| = |V| - e_G(v_1, A_1) - 1 = n - \Delta - 1.$$

Observe that if $v_i \in B_i$ then $|A_{i+1}| = |A_i| - e_G(v_i, A_i)$, and if $v_i \in A_i$ then $|A_{i+1}| = |A_i| - e_G(v_i, A_i) - 1$. Hence, for $1 \leq i \leq p-1$, we have

$$|A_{i+1}| \leq |A_i| - e_G(v_i, A_i) \leq |A_i|(1-c),$$

by Lemma 3.3.3. This directly proves the lemma. □

Lemma 3.3.5 *The first phase of Algorithm MinBST consists* $p \leq \left\lceil \log_{\frac{1}{1-c}} n \right\rceil + 1$ *iterations.*

PROOF: By Lemma 3.3.4, we have a sufficient condition for H_i being a suitable spanning forest (or equivalently, for C_i being a dominating set of V). Indeed, H_i is a spanning forest with no isolated vertices if A_i is empty, which is always the case whenever

$$1 > (n - \Delta - 1)(1 - c)^{i-2},$$

or equivalently

$$i \geq \left\lceil \log_{\frac{1}{1-c}}(n - \Delta - 1) \right\rceil + 2.$$

Therefore we never need more than

$$\left\lceil \log_{\frac{1}{1-c}}(n - \Delta - 1) \right\rceil + 2$$

iterations, that is, using $\Delta \geq nc$,

$$p \leq \left\lceil \log_{\frac{1}{1-c}}(n - \Delta - 1) \right\rceil + 2 \leq \left\lceil \log_{\frac{1}{1-c}} n(1 - c) \right\rceil + 2 = \left\lceil \log_{\frac{1}{1-c}} n \right\rceil + 1.$$

This concludes the proof of the lemma. □

We now turn to the counting of the branchings in the obtained spanning forest. First observe that, by Claim 3.3.2/6, B_p has no branchings. Now let b_1 denote the number of branchings in C_p. Then $b_1 \leq |C_p| = p$.

If H_p is connected then we set $H = H_p$ and in this case H is a spanning tree with $b = b_1$ branchings. If H_p has more than one components then they must be connected by adding $\mathrm{comp}_{H_p}(B_p \cup C_p) - 1$ pieces of G-edges (say E'') to H_p. These extra edges, added by the second phase of the algorithm, produce

$$b_2 \leq 2|E''| = 2\left[\mathrm{comp}_{H_p}(B_p \cup C_p) - 1\right]$$

new branchings. In this case, we set $H = H_p + E''$, that is, we add edges in E'' to H_p. Then H has $b = b_1 + b_2$ branchings.

In both cases, the number of branchings is

$$b \leq b_1 + b_2 \leq p + 2\,\mathrm{comp}_{H_p}(B_p \cup C_p) - 2 \leq 3p - 2 \leq 3\left\lceil \log_{\frac{1}{1-c}} n \right\rceil + 1,$$

as stated by Theorem 3.3.1.

Now we consider the running time of Algorithm MinBST. According to Lemma 3.3.5, during the first phase, there are $\mathcal{O}(\log n)$ iterations. The i^{th} iteration is composed of the following steps. We search for the vertex $v \in A_i \cup B_i$ maximizing $e_G(v, A_i)$. This requires $\mathcal{O}(n)$ time. Then we move v_i to C and its neighbors being in A_i to B. Finally we update $\mathsf{ADegree}[w]$ for all $w \in (A_{i+1} \cup B_{i+1}) \cap N(A_i \setminus A_{i+1})$. Observe that such updates are done at most twice for each G-edge, namely, when one

of its ends is removed from A. Therefore, the cumulated number of such updates is $\mathcal{O}(m)$. As a result, the first phase needs $\mathcal{O}(n \log n + m)$ time in total. In the second phase, we consider each edge a constant number of times: at most once when its component is traversed, and once for each of its end vertices when they get removed from S. Thus we need $\mathcal{O}(m)$ time to create a spanning tree from H in the second phase. Therefore, the total running time of the algorithm is $\mathcal{O}(m + n \log n)$. This finishes the proof of Theorem 3.3.1.

Chapter 4

Spanning Tree Leaves and Vulnerability

This chapter focuses on graph vulnerability parameters. They are used to measure how much structural damage can be caused in a graph by removing some "important parts" of it [11, 12, 34, 35, 74]. Both hamiltonicity theory and network design applications widely use these parameters to describe the structure of graphs. Our work fits into both of these categories. We first prove some theoretical results on the connection between the number of spanning tree leaves and vulnerability parameters. Then, in Chapter 5, we use these results to build an approximation algorithm for the Maximum Internal Spanning Tree problem. Besides, our framework yields a new proof for the fact that the internal vertices of any independence tree give a 2-approximation for the Minimum Connected Vertex Cover problem [60].

The first connection between hamiltonicity and vulnerability was given in a well-known theorem of basic graph theory:

Theorem 4.0.6 [46, p. 30] *If a graph is traceable then it cannot be split to more than $k+1$ components by removing at most k of its vertices.*

This theorem gives only a necessary condition of traceability. Observe that this condition is founded on a vulnerability property of the graph. Therefore, it is worth investigating how vulnerability parameters can provide sufficient conditions of traceability. A considerable amount of research followed this approach, for a survey on them, the reader is referred to [12].

In this chapter, we use two closely related vulnerability parameters: scattering number and cut-asymmetry. Scattering number shows how many components we can get by removing a few vertices [44, 47, 74]. Cut-asymmetry does the same when a few *connected subgraphs* are removed [6, 8]. We use scattering number to lower bound the number of leaves in a spanning tree, and cut-asymmetry to upper bound the number of G-independent leaves of a spanning tree. By means of scattering number we can restate Theorem 4.0.6 as follows: if a graph is traceable then its scattering number is at most one, that is, the scattering number provides a necessary condition of traceability. We show that cut-asymmetry gives a sufficient

condition, namely if its value is at most one then the graph is traceable. This is our main hamiltonicity related result.

The rest of this chapter is organized as follows. In Section 4.1 we investigate scattering number and the minimum number of spanning tree leaves. In Section 4.2 we give two basic properties of minimum leaf spanning trees. Finally, Section 4.3 introduces cut-asymmetry, gives a set of its basic properties and shows a connection to the maximum number of independent spanning tree leaves.

4.1 Scattering Number and the Minimum Number of Leaves

Jung defined the scattering number as follows:

Definition 4.1.1 [44] *The* scattering number *of a non-complete graph* $G = (V, E)$ *is*

$$\mathrm{sc}(G) = \max_{X \subset V, X \neq \emptyset} \{\mathrm{comp}(G[V \setminus X]) - |X| : \mathrm{comp}(G[V \setminus X]) \geq 2\}.$$

By definition, the scattering number of the complete graph K_n *is* $\mathrm{sc}(K_n) = -\infty$.

Using this notion, Theorem 4.0.6 can be rewritten in the form of

Theorem 4.1.2 *If* G *is a traceable graph then* $\mathrm{sc}(G) \leq 1$.

Recall that $\mathrm{ml}(G)$ is the minimum number of leaves in the spanning trees of G. An observation based on Theorem 4.1.2 is

Theorem 4.1.3 *If* G *is a traceable graph then* $\mathrm{ml}(G) \geq \mathrm{sc}(G) + 1$.

Theorem 4.1.2 gives a connection between scattering number and the minimum number of spanning tree leaves whenever this latter equals to 2. In this section we show that Theorem 4.1.3 holds for any graph, namely that every spanning tree of an arbitrary graph G has at least $\mathrm{sc}(G) + 1$ leaves.

As a first step, we prove this statement for trees.

Lemma 4.1.4 *Let* T *be a tree with* q *leaves. Then* $q \geq \mathrm{sc}(T) + 1$.

Proof: To prove the upper bound let $X = \{x_1, x_2, \ldots, x_k\}$ be the vertex-set providing the maximum in Definition 4.1.1. If several such sets exist, let us choose one of minimum cardinality. Thus, by definition, $\mathrm{sc}(T) = \mathrm{comp}(T[V \setminus X]) - |X|$. Moreover, each vertex $x_j \in X$ is a branching of T, that is, $d(x_j) \geq 3$ (for $j = 1, 2, \ldots, k$). Otherwise, if some $x_j \in X$ had degree less than 3 then for $X' = X - x_j$ we would have $\mathrm{comp}(T[V \setminus X']) \geq \mathrm{comp}(T[V \setminus X]) - 1$, and so we should have chosen X' instead of X.

Now let $X_0 = \emptyset$ and let $X_j = X_{j-1} + x_j$ (for $j = 1, 2, \ldots, k$). Furthermore let $T_j = T[V \setminus X_j]$ (for $j = 0, 1, \ldots, k$). That is, T_j is the forest obtained from T by removing the vertices of X_j. Then $\mathrm{comp}(T_j) \leq \mathrm{comp}(T_{j-1}) + d(x_j) - 1$, and hence

$$\mathrm{comp}(T_k) = \mathrm{comp}(T[V \setminus X]) \leq 1 + \sum_{x_j \in X} d(x_j) - |X|. \tag{4.1}$$

On the other hand, let Y denote the set of branchings not in X, namely $Y = V_{\geq 3}(T) \setminus X$. Then counting the vertices of T according to their degree we obtain:

$$\begin{aligned} |V(T)| &= q + |V_2(T)| + |X| + |Y| \\ &= q + |V_2(T)| + \sum_{i \geq 3} |V_i(T) \cap X| + \sum_{i \geq 3} |V_i(T) \cap Y|. \end{aligned} \tag{4.2}$$

Counting the total degree of vertices in T we have:

$$2|V(T)| - 2 = q + 2|V_2(T)| + \sum_{i \geq 3} i|V_i(T) \cap X| + \sum_{i \geq 3} i|V_i(T) \cap Y|, \tag{4.3}$$

Equations (4.2) and (4.3) yield

$$\begin{aligned} \sum_{x_j \in X} d(x_j) = \sum_{i \geq 3} i|V_i(T) \cap X| = \sum_{i \geq 3} (i-2)|V_i(T) \cap X| + 2|X| = \\ q - 2 - \sum_{i \geq 3} (i-2)|V_i(T) \cap Y| + 2|X| \leq q - 2 + 2|X|. \end{aligned}$$

Using (4.1), this implies

$$\mathrm{sc}(T) = \mathrm{comp}(T[V \setminus X]) - |X| \leq q - 1$$

proving the upper bound on the scattering number. □

Now we generalize this for arbitrary graphs.

Theorem 4.1.5 *For any graph G we have* $\mathrm{ml}(G) \geq \mathrm{sc}(G) + 1$.

PROOF: Let T be a minimum leaf spanning tree of G, that is, $|V_1(T)| = \mathrm{ml}(G)$, and let X be a set maximizing $\mathrm{comp}(G[V \setminus X]) - |X|$. Then as $\mathrm{comp}(G[V \setminus X]) \leq \mathrm{comp}(T[V \setminus X])$, by Lemma 4.1.4 we have:

$$\begin{aligned} \mathrm{sc}(G) = \mathrm{comp}(G[V \setminus X]) - |X| \leq \mathrm{comp}(T[V \setminus X]) - |X| \leq \\ \mathrm{sc}(T) \leq |V_1(T)| - 1 = \mathrm{ml}(G) - 1. \end{aligned}$$

□

Notice that the bounds of Lemma 4.1.4 and Theorem 4.1.5 are both tight. Indeed, let T be a spanning tree with a single branching and q leaves. Then removing the branching we obtain q components and so $\mathrm{sc}(T) \geq q - 1 = |V(T)| - 1$.

Theorem 4.1.5 provides a simple lower bound on the minimum number of spanning tree leaves by means of the scattering number. Unfortunately, the scattering number can be negative even for non-traceable graphs. Therefore, it cannot be used for upper bounding the minimum number of spanning tree leaves. To see this, let us mention here toughness, another vulnerability parameter whose connection to Hamiltonicity is under an intensive research [11, 12, 49].

Definition 4.1.6 [23] *The* toughness *of a non-complete graph $G = (V, E)$ is*

$$\tau(G) = \min_{X \subset V, X \neq \emptyset} \left\{ \frac{|X|}{\mathrm{comp}(G[V \setminus X])} : \mathrm{comp}(G[V \setminus X]) \geq 2 \right\}.$$

By definition, the toughness of the complete graph K_n is $\tau(K_n) = \infty$.

This definition immediately implies that $\tau(G) > 1$ if and only if $\mathrm{sc}(G) < 0$. Chvatal conjectured [23] that there is a minimum value of toughness implying traceability. For a long period, $\tau(G) = 2$ was believed to be this limit. However, Bauer et al. showed non-traceable 2-tough graphs [13]. This means that there are graphs with negative scattering number even among non-traceable graphs.

We are in a better situation if we restrict ourselves for trees. In this case, Theorem 4.1.7 gives an upper bound for the number of leaves in terms of the scattering number.

Theorem 4.1.7 *Let T be a q-leaf tree on at least 3 vertices. Then $q \leq 2\,\mathrm{sc}(T)$.*

Proof: We construct a subset X of internal vertices of T such that each component of $T[V \setminus X]$ contains at most one leaf of T implying $\mathrm{comp}(T[V \setminus X]) \geq q$. Then we prove that $|X| \leq q/2$. Combining these with the definition of scattering number directly yields the theorem.

Recall that a spider is a tree that has at most one branching vertex. We build a sequence $\{T_1 = T, T_2, \ldots, T_k\}$ of trees such that T_k is a spider. If, for some i, T_i is not a spider we construct T_{i+1} as follows: let x_i be a branching of T_i with exactly one trunk-edge (x_i, y_i) incident to it (such an x_i exists if T_i is not a spider). Let T_i' be the component of $T_i[V(T_i) - x_i]$ that contains y_i. If y_i is not a leaf of T_i' then let $T_{i+1} = T_i'$. Otherwise, T_{i+1} is obtained by deleting the branch of y_i from T_i' (Fig. 4.1). Observe that the leaves of T_{i+1} form a subset of leaves of T_i, and that a leaf l is in $V_1(T_i) \setminus V_1(T_{i+1})$ if and only if $b(l) = x_i$.

If T_i is a spider (thus $i = k$) then we define x_k to be the branching of the spider. If T_k is a path then x_k is chosen to be any internal vertex of T_k. Let $X = \{x_1, x_2, \ldots, x_k\}$. Observe that this construction ensures that each leaf of T is in a different component of $T[V \setminus X]$, thus $q \leq \mathrm{comp}(T[V \setminus X])$.

To see that $q \geq 2|X|$ let b_i denote the number of those branches of T_i that end in x_i. Then $b_i = d_{T_i}(x_i) - 1$ and $b_i \geq 2$ for $1 \leq i \leq k-1$. This implies that

$$\sum_{i=1}^{k} b_i \geq 2k - 1.$$

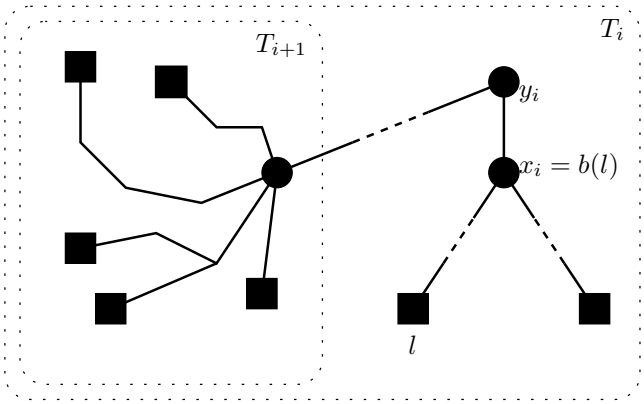


Figure 4.1: Constructing T_{i+1} from T_i by removing the branching x_i

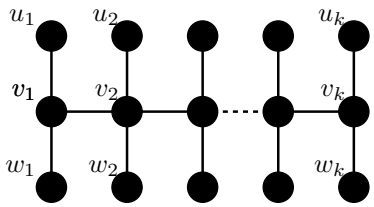


Figure 4.2: A graph proving that the bound of Theorem 4.1.7 is tight.

As x_i cuts down exactly b_i leaves from T_i, and as T_k is a spider with $b_k + 1$ leaves we have $|V_1(T_{i+1})| = |V_1(T_i)| - b_i$, and $|V_1(T_k)| = b_k + 1$. Putting these together gives

$$q = \sum_{i=1}^{k} b_i + 1 \geq 2k = 2|X|.$$

As a result

$$\mathrm{sc}(T) \geq \mathrm{comp}(T[V \setminus X]) - |X| \geq q - \frac{q}{2} = \frac{q}{2},$$

which finishes the proof. □

Again, we have a tight bound in Theorem 4.1.7. Indeed, let T be a tree on vertices $V(T) = \{v_1, v_2, \ldots, v_k, u_1, u_2, \ldots, u_k, w_1, w_2, \ldots, w_k\}$ having edges

$$E(T) = \{(v_i, v_{i+1})\}_{i=1,2,\ldots,k-1} \cup \{(v_i, u_i)\}_{i=1,2,\ldots,k} \cup \{(v_i, w_i)\}_{i=1,2,\ldots,k},$$

(see Fig. 4.2). Then it is easy to see that $\mathrm{sc}(T) = k = \frac{|V_1(T)|}{2}$, and so the bound is tight.

We have seen that a difference of factor 2 can arise between the number of leaves and the scattering number of trees. In Section 4.3 we present cut-asymmetry as a new vulnerability parameter which exactly determines the number of leaves when

applied on a tree. Cut-asymmetry is then used (together with the results of this section) to prove the approximation ratio of our algorithm in Section 5.5.

4.2 Basic Properties of Minimum Leaf Spanning Trees

In this section we give two basic properties of minimum leaf spanning trees. We remind the reader that if l is a leaf of tree T then $b(l)$ denotes the branching closest to l, $b^-(l)$ stands for the neighbor of $b(l)$ in the branch of l, and $x^{\to l}$ is the successor of vertex x on the path $P_T(x,l)$.

Lemma 4.2.1 *If T is a minimum leaf spanning tree of G then T is either a Hamiltonian path or an independence tree.*

PROOF: Suppose for a contradiction that T is not a Hamiltonian path and there exist two leaves l_1, l_2 of T such that $(l_1,l_2) \in E(G)$. Then the spanning tree T' with edge-set $E(T') = E(T) + (l_1,l_2) - (b(l_1), b^-(l_1))$ has less leaves than T contradicting the minimality of T (Fig. 4.3(a)). □

Lemma 4.2.2 *Let G be a non-traceable graph, and T be a minimum leaf spanning tree of G. Let l_1 be an arbitrary leaf of T, and $(l_1,x) \in E(G) \setminus E(T)$ be a non-tree edge. Then $d_T(x^{\to l_1}) = 2$ and if $l_2 \neq l_1$ is a leaf of T then $(x^{\to l_1}, l_2) \notin E(G)$.*

PROOF: Let us consider the spanning tree T' with edge-set $E(T') = E(T) + (l_1,x) - (x^{\to l_1}, x)$. Vertex $x^{\to l_1}$ must be a forwarding vertex of T. Indeed, if $d_T(x^{\to l_1}) > 2$ then $V_1(T') = V_1(T) - l_1$ and so T' has less leaves than T, a contradiction (see Fig. 4.3(b)). If $d_T(x^{\to l_1}) = 2$ then $V_1(T') = V_1(T) - l_1 + x^{\to l_1}$, that is, T and T' has the same number of leaves. However, if $(x^{\to l_1}, l_2)$ was a G-edge, it would connect two leaves of T' and thus, by Lemma 4.2.1, neither T' nor T would be a minimum leaf spanning tree (see Fig. 4.3(c)). □

We have the following trivial corollary of Lemma 4.2.1:

Corollary 4.2.3 *If G is a non-complete connected graph then* $\mathrm{ml}(G) \leq \alpha(G)$.

4.3 Cut-Asymmetry and the Maximum Number of Independent Leaves

In Section 4.1 we have seen that scattering number provides both lower and upper bounds on the number of leaves of a tree. In this section we use another vulnerability measure, namely cut-asymmetry, for obtaining better bounds. Recall that scattering number shows how much structural damage can be caused by removing some individual vertices from the graph. We define cut-asymmetry in a similar way but

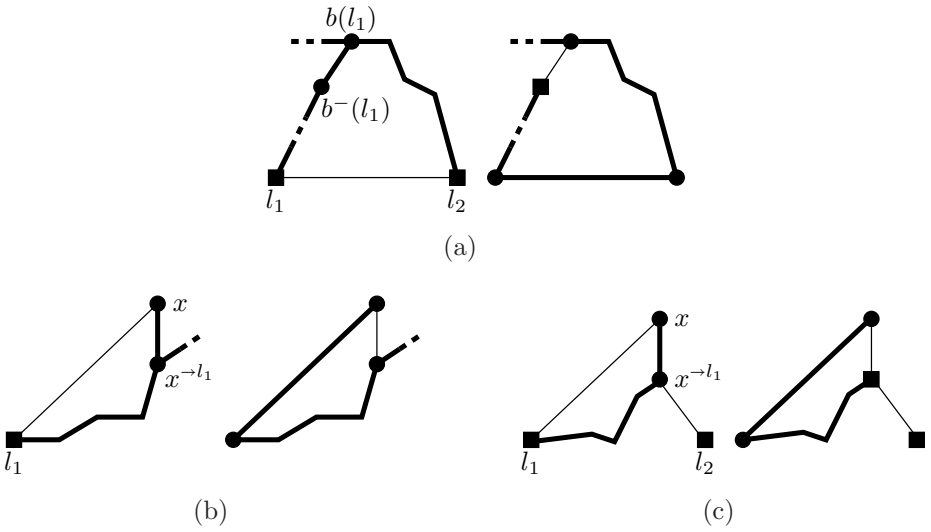


Figure 4.3: Decreasing the number of leaves by local improvements

we count the connected subgraphs (instead of individual vertices) to be removed. As an example, let us consider graph G of Figure 4.4, a K_8 with a triangle joint to each vertex. If we remove k ($1 \leq k \leq 7$) vertices of the complete K_8 subgraph, we obtain $k+1$ components. Thus, $\mathrm{sc}(G) \geq k+1-k = 1$. Contrary, removing the whole K_8 (a single connected subgraph), we get 8 components. Therefore, the cut-asymmetry $\mathrm{ca}(G) \geq 8-1 = 7$. In fact, one can easily check that $\mathrm{sc}(G) = 1$ and $\mathrm{ca}(G) = 7$.

In Subsection 4.3.2 we prove that cut-asymmetry of a tree exactly characterizes the number of its leaves. In Subsection 4.3.3 we use cut-asymmetry to determine the maximum number of independent leaves of a spanning tree of an arbitrary graph. This puts cut-asymmetry in the context of research on independence trees [16, 19]. Besides, in Subsection 4.3.3, we point out a connection between cut-asymmetry and the minimum size of a connected vertex cover.

First, we define cut-asymmetry as follows.

Definition 4.3.1 [8] *The cut-asymmetry of a graph $G = (V, E)$ is*

$$\mathrm{ca}(G) = \max_{X \subset V, X \neq \emptyset} \{\mathrm{comp}(G[V \setminus X]) - \mathrm{comp}(G[X])\}.$$

Observe that this definition takes the maximum over all non-trivial cuts of G. Cut-asymmetry counts how much the two sides of a cut can differ in terms of the number of components.

4.3.1 Basic properties of cut-asymmetry

This definition immediately implies that $\mathrm{ca}(G) \geq \max\{\mathrm{sc}(G), 0\}$, and that $\mathrm{ca}(G) \leq |V|-2$, where equality holds if and only if G is a star. We mention here some further

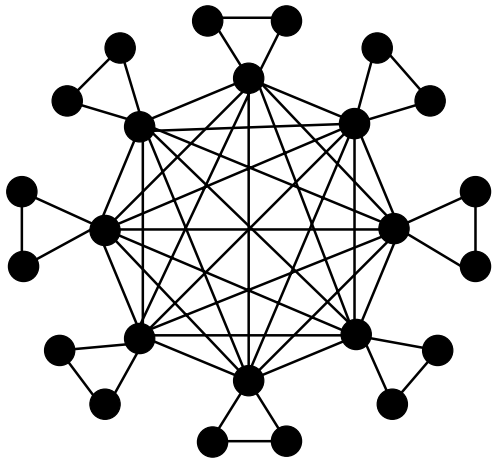


Figure 4.4: Difference between $\mathrm{sc}(G)$ and $\mathrm{ca}(G)$

properties of cut-asymmetry.

The following theorem is a consequence of Theorems 4.3.10 and 4.3.22, however a direct proof is also given below.

Theorem 4.3.2 $\mathrm{ca}(G) = 0$ *if and only if* G *is either a complete graph or a cycle.*

Proof: If G is complete or a cycle then its cut-asymmetry is trivially 0. To see the other direction let G be a non-complete graph such that $\mathrm{ca}(G) = 0$. Let $Z = \{z_1, z_2, \ldots, z_k\}$ be an arbitrary independent set. As $\mathrm{ca}(G) = 0$ the graph $G[V \setminus Z]$ has k components $C_1, C_2, \ldots, C_k$. Let us contract each component C_i to a single vertex c_i. By this transformation we obtain the connected bipartite graph G' with vertex classes Z and $C = \{c_1, c_2, \ldots, c_k\}$.

Claim 4.3.3 G' *is an even cycle.*

Proof: For every $1 \leq j \leq k$ we have $d_{G'}(z_j) = 2$. Otherwise if $d_{G'}(z_j) = 1$ for some j then $\mathrm{comp}(G[Z - z_j]) = k - 1$ and $\mathrm{comp}(G[V \setminus (Z - z_j)]) = k$ would imply $\mathrm{ca}(G) \geq 1$. If $d_{G'}(z_j) \geq 3$ for some j then $\mathrm{comp}(G[Z - z_j]) = k - 1$ and $\mathrm{comp}(G[V \setminus (Z - z_j)]) \leq k - 2$ would imply $\mathrm{ca}(G) \geq 1$. A similar reasoning shows that $d_{G'}(c_j) = 2$ for all $1 \leq j \leq k$. As a result G' is a 2-regular connected bipartite graph, that is, an even cycle. □

Claim 4.3.4 *If some vertex* v *is in the component* C_i*, for some* i*, such that* $2 \leq |V(C_i)|$ *then* v *has at most one* G*-neighbor in* Z*.*

Proof: Suppose that $z_{j_1} \in Z$ and $z_{j_2} \in Z$ are both G-neighbors of v. Then $\mathrm{comp}(G[Z + v]) = k - 1$ and $\mathrm{comp}(G[V \setminus (Z + v)]) = k$ imply $\mathrm{ca}(G) \geq 1$, a contradiction. □

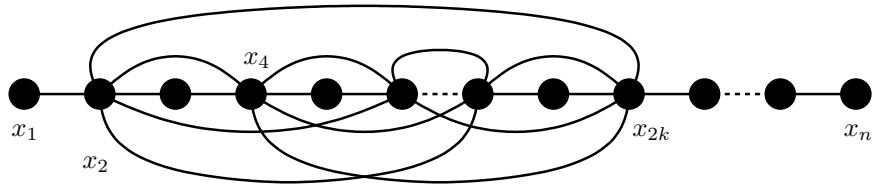


Figure 4.5: A graph on n vertices with $\mathrm{ca}(G) = k$

Claim 4.3.5 *There is at most one edge in G between vertex $z_j \in Z$ and component C_i for all i and j.*

PROOF: Suppose that there exist two vertices $u, v \in C_i$ and a vertex $z_j \in Z$ (for some i and j) such that both (u, z_j) and (v, z_j) are edges of G. Then, by Claim 4.3.4, $\mathrm{comp}(G[Z - z_j + u]) = k$ as u has no other neighbors in Z. We also have $\mathrm{comp}(G[V \setminus (Z - z_j + u)]) = k - 1$ as z_j is neighboring to some other component $C_{i'}$. This implies $ca(G) \geq 1$, a contradiction. □

As a result of Claims 4.3.3 and 4.3.5 we obtain that exactly two edges of G leave each component C_i. For a fixed i let these edges be (z_{j_1}, u_i) and (z_{j_2}, v_i), where $u_i, v_i \in V(C_i)$. Note that $u_i = v_i$ if and only if C_i has a single vertex. Suppose that C_i has a vertex x_i such that $C_i - x_i$ has a $u_i - v_i$ path. In this case $\mathrm{comp}(G[Z - z_{j_1} + x_i]) = k$, using Claim 4.3.3, and $\mathrm{comp}(G[V \setminus (Z - z_{j_1} + x_i)]) = k - 1$ implying $\mathrm{ca}(G) \geq 1$, a contradiction. We conclude that component C_i is either a single vertex or a simple path connecting u_i and v_i. This, together with Claim 4.3.3, proves the theorem. □

Let us recall that $\mathrm{sc}(G) \leq 1$ is a necessary condition for the existence of a Hamiltonian path. The following theorem shows a sufficient condition for traceability by the means of cut-asymmetry. We do not prove this theorem directly. It is a corollary of the later discussed Theorems 4.3.10 and 4.3.22.

Theorem 4.3.6 *If $\mathrm{ca}(G) \leq 1$ then G has a Hamiltonian path, that is, G is traceable.*

Unfortunately the traceability of G does not imply a low value of $\mathrm{ca}(G)$ as shown by the following theorem.

Theorem 4.3.7 *If $G = (V, E)$ is a traceable graph on n vertices then $\mathrm{ca}(G) \leq \lfloor \frac{n-1}{2} \rfloor$. Moreover, if k is an arbitrary integer such that $0 \leq k \leq \lfloor \frac{n-1}{2} \rfloor$ then there exists a traceable graph G on n vertices for which $\mathrm{ca}(G) = k$.*

PROOF: Theorem 4.1.3 implies that $\mathrm{sc}(G) \leq 1$, that is, for any proper subset $X \subset V$ we have $\mathrm{comp}(G[V \setminus X]) \leq |X| + 1$. As $\mathrm{comp}(G[V \setminus X]) \leq n - |X|$ we obtain $\mathrm{comp}(G[V \setminus X]) \leq \lfloor \frac{n+1}{2} \rfloor$. Thus $\mathrm{ca}(G) \leq \lfloor \frac{n-1}{2} \rfloor$.

To see the second part let us fix $k \leq \lfloor \frac{n-1}{2} \rfloor$. A graph on n vertices whose cut-asymmetry is exactly k is constructed as follows. If $k = 0$ then according to

Theorem 4.3.2 the graph $G = K_n$ is a good example. So we suppose that $k \geq 1$. Let $P = x_1, x_2, \ldots, x_n$ be a path of length n. Now we obtain G by adding the edges of the complete graph on $x_2, x_4, \ldots, x_{2k}$ to P (see Figure 4.5). It is easy to check that $\mathrm{ca}(G) = k$. □

After discussing the basic properties of cut-asymmetry, we now turn to its applications.

4.3.2 Cut-asymmetry of trees

In this subsection we prove that unlike scattering number, cut-asymmetry fully determines the number of leaves of a tree.

Let us emphasize that the original proof of Lemma 4.3.8 is entirely due to Gábor Wiener and the only reason for giving it here is to make the section self-containing. For the used notation, the reader is referred to Section 2.1.

Lemma 4.3.8 [8] *If T is a tree then $|V_1(T)| = \mathrm{ca}(T) + 1$.*

Proof: First observe that

$$\mathrm{comp}(T[V_1(T)]) - \mathrm{comp}(T[V \setminus V_1(T)]) = |V_1(T)| - 1,$$

since $V_1(T)$ is an independent set and $V \setminus V_1(T)$ spans a subtree. This implies $\mathrm{ca}(T) \geq |V_1(T)| - 1$.

In order to show that $\mathrm{ca}(T) \leq |V_1(T)| - 1$ let us have a set $X \subset V$ of vertices for which $\mathrm{ca}(T) = \mathrm{comp}_T(X) - \mathrm{comp}_T(V \setminus X)$. For the sake of convenience let $x = \mathrm{comp}_T(X)$ and $\overline{x} = \mathrm{comp}_T(V \setminus X)$. Then $e_T(X) = |X| - x$, and $e_T(V \setminus X) = n - |X| - \overline{x}$, thus the total degree of vertices in X is

$$\begin{aligned} d_T(X) = 2e_T(X) + e_T(X, V \setminus X) &= 2e_T(X) + n - 1 - (e_T(X) + e_T(V \setminus X)) \\ &= 2(|X| - x) + x + \overline{x} - 1 = 2|X| - x + \overline{x} - 1. \end{aligned}$$

If some $v \in X$ is not a leaf of T then it increases this total degree by at least 2. Therefore the number of leaves of T is

$$|V_1(T)| \geq |V_1(T) \cap X| \geq 2|X| - d_T(X) \geq x - \overline{x} + 1 = \mathrm{ca}(T) + 1,$$

as requested. □

Thus for any tree T, using Lemma 4.1.4 and Theorem 4.1.7, we obtain

$$\mathrm{sc}(T) + 1 \leq |V_1(T)| = \mathrm{ca}(T) + 1 \leq 2\,\mathrm{sc}(T).$$

For a connected graph G we have proved so far

$$\mathrm{sc}(G) + 1 \leq \min\{\mathrm{ml}(G), \mathrm{ca}(G) + 1\}.$$

In what follows we prove that $\mathrm{ml}(G) \leq \mathrm{ca}(G) + 1$ holds for all graphs but the complete graph K_n and the cycle C_n. For this purpose we introduce a new parameter called leaf-independence.

4.3.3 Leaf-independence

Definition 4.3.9 *Let G be any connected graph and T be a spanning tree of G. The* leaf-independence *of T in G, denoted by $\mathrm{li}_G(T)$, is the cardinality of a maximum G-independent subset of $V_1(T)$. The* leaf independence $\mathrm{li}(G)$ of the graph G *is the maximum of $\mathrm{li}_G(T)$ over all spanning trees of G.*

It is important to point out that a spanning tree T can have more than $\mathrm{li}_G(T)$ leaves. Indeed, $li_G(T) = |V_1(T)|$ if and only if T is an independence tree. Note that $\mathrm{li}(G)$ is defined for all connected graphs, however, not all connected graphs have an independence tree. Böhme et al. [16] characterized the graphs that have no independence trees. (These graphs are the complete graph, the cycle and the complete bipartite graph $K_{n,n}$, see Theorem 4.3.15) If an independence tree with α_{t} leaves exists then obviously $\mathrm{li}(G) \geq \alpha_{\mathrm{t}}$.

Notice that $\mathrm{li}(G)$ can be strictly greater than $\alpha_{\mathrm{t}}(G)$ as shown by the graph G obtained from $K_{4,4}$ removing one of its edges. It is easy to see that $\mathrm{li}(G) = 3$, and $\alpha_{\mathrm{t}}(G) = 2$.

Let us refer here to the recently published work of Levit and Mandrescu [52] dealing with the connection between the leaves and the intersection of all independent sets of a tree. The authors show that in an arbitrary tree T satisfying $\alpha(T) > |V(T)|/2$ there exist at least two pendant vertices an even distance apart belonging to all independent sets of T.

Now observe that the definition of leaf-independence directly yields $\mathrm{li}(G) \leq \alpha(G)$ and also implies that the leaf-independence of a tree T having at least 3 vertices is equal to the number of its leaves. Thus, by Lemma 4.3.8, $\mathrm{ca}(T) = |V_1(T)| - 1 = \mathrm{li}(T) - 1$. The following theorem states that this equality between cut-asymmetry and leaf-independence holds for every graph.

Theorem 4.3.10 *Let G be a connected graph. Then* $\mathrm{ca}(G) = \mathrm{li}(G) - 1$.

Proof: Let T be a spanning tree of G such that $\mathrm{li}_G(T) = \mathrm{li}(G)$. Let Z be a maximum independent set of $V_1(T)$, and let $X = V \setminus Z$. Then, on one hand,

$$\mathrm{comp}(G[V \setminus X]) = |V \setminus X| = \mathrm{li}(G),$$

and on the other hand

$$1 \leq \mathrm{comp}(G[X]) \leq \mathrm{comp}(T[X]) = 1.$$

Therefore

$$\mathrm{ca}(G) \geq \mathrm{comp}(G[V \setminus X]) - \mathrm{comp}(G[X]) = \mathrm{li}(G) - 1.$$

To prove the reverse direction let us choose X^* to be the set giving the maximum in Definition 4.3.1. If several such sets exist we take one of maximum cardinality. First we show that in this case $V \setminus X^*$ is an independent set of G. Suppose the

contrary, namely that there is an edge (u, v) in $G[V \setminus X^*]$. Then considering $X' = X^* + u$ we have

$$\text{comp}(G[X']) \leq \text{comp}(G[X^*]),$$

and

$$\text{comp}(G[V \setminus X']) \geq \text{comp}(G[V \setminus X^*]).$$

This forms a contradiction as

$$\text{comp}(G[V \setminus X']) - \text{comp}(G[X']) \geq \text{comp}(G[V \setminus X^*]) - \text{comp}(G[X^*]),$$

and $|X'| > |X^*|$.

In what follows we show that X^* spans a connected subgraph of G. Suppose for a contradiction that $G[X^*]$ has more than one components. Then, since $V \setminus X^*$ is an independent set, there must exist two components C_1 and C_2 of $G[X^*]$, and a vertex $u \in V \setminus X^*$ such that $G[V(C_1) \cup V(C_2) + u]$ is connected. Hence for the independent set $V \setminus X' = (V \setminus X^*) - u$ we have:

$$\text{comp}(G[V \setminus X']) = |V \setminus X'| = |V \setminus X^*| - 1 = \text{comp}(G[V \setminus X^*]) - 1,$$

and

$$\text{comp}(G[X']) \leq \text{comp}(G[X^*]) - 1.$$

Thus

$$\text{comp}(G[V \setminus X']) - \text{comp}(G[X']) \geq \text{comp}(G[V \setminus X^*]) - \text{comp}(G[X^*]),$$

a contradiction as $|X'| > |X^*|$.

To finish the proof let T^* be a spanning tree of $G[X^*]$. We then connect each vertex of $V \setminus X^*$ to T^*. Thus we obtain a spanning tree of G in which all vertices of $V \setminus X^*$ are leaves. Thus

$$\text{li}(G) \geq |V \setminus X^*| = \text{comp}(G[V \setminus X^*]) - \text{comp}(G[X^*]) + 1 = \text{ca}(G) + 1,$$

using that $V \setminus X^*$ is independent and that X^* spans a connected subgraph of G. □

In the proof of Theorem 4.3.10 we have seen that there exists an independent set $Z^* = V \setminus X^*$ such that $G[V \setminus Z^*]$ is connected and that $\text{comp}(G[Z^*]) - \text{comp}(G[V \setminus Z^*]) = \text{ca}(G)$. This yields us an alternative definition of cut-asymmetry, that is

Corollary 4.3.11 *Let Z be a maximum size independent set of V for which $G[V \setminus Z]$ is connected. Then* $\text{ca}(G) = |Z| - 1$.

Now we examine the computational complexity of calculating $\text{ca}(G)$, or equivalently $\text{li}(G)$. To this aim we first point out a connection between cut-asymmetry and connected vertex covers.

Recall that a connected vertex cover of G is a vertex-set that spans a connected subgraph and that meets all edges of G. Let us denote by $\text{cvc}(G)$ the size of a

minimum cardinality connected vertex cover of G. To find out the size of a minimum connected vertex cover is one of the first problems turned out to be NP-hard [30].

Observe that the set $Z^* = V \setminus X^*$ of the above proof is a maximum-size independent vertex-set whose complement X^* spans a connected subgraph. The following claim directly shows this connection.

Claim 4.3.12 *For any connected graph G we have*

$$\mathrm{li}(G) = \max\{|X| : X \text{ is independent, } V \setminus X \text{ is connected}\}.$$

PROOF: On one hand, let X be a set providing the maximum on the right hand side. Let us take a spanning tree of $G[V \setminus X]$ and connect each vertex of X to it by a single edge. Thus we obtain a spanning tree T of G. Every vertex of X is a leaf of T thus we have $\mathrm{li}(G) \geq |X|$. On the other hand, let T be a spanning tree with $\mathrm{li}(G)$ independent leaves forming the set X. Then $G[V \setminus X]$ is connected and so the maximum on the right hand side is at least $|X| = \mathrm{li}(G)$. □

This immediately proves the following connection between $\mathrm{li}(G)$ and $\mathrm{cvc}(G)$, the size of a minimum-size connected vertex cover of G:

Corollary 4.3.13 *For any connected graph G on n vertices* $\mathrm{li}(G) + \mathrm{cvc}(G) = n$.

Note that the above arguments also show that the complement of a minimum connected vertex cover is always a maximum size independent subset of leaves of a spanning tree.

Thus Corollary 4.3.13 and Theorem 4.3.10 imply:

Theorem 4.3.14 *Cut-asymmetry and leaf-independence of a graph are NP-hard to compute.*

Notice that it is also NP-hard to calculate the scattering number of a graph [73].

In what follows we show that leaf-independence provides an upper bound on the minimum number of leaves in almost all graphs. By Lemma 4.2.1 we obtain that if G has no Hamiltonian path then the leaves of any minimum leaf spanning tree of G are independent. In this case $\mathrm{ml}(G) \leq \mathrm{li}(G)$ is straightforward. If G is traceable then $\mathrm{ml}(G) = 2$ and so $\mathrm{ml}(G) \leq \mathrm{li}(G)$ is true if and only if G has a spanning tree with at least two independent leaves. We use a result of Böhme et al. to show that this condition is satisfied whenever G is neither the complete graph K_n nor the cycle C_n.

Theorem 4.3.15 [16] *If G is traceable and each Hamiltonian path of G is the part of a Hamiltonian cycle then G is either the complete graph K_n or the cycle C_n or the complete bipartite graph $K_{n,n}$.*

For the sake of completeness, we give here our own proof for this theorem.

Let G be a graph satisfying the conditions of Theorem 4.3.15, and C be a Hamiltonian cycle of G. Let the vertices of G be numbered according to their order in C, that is, $v_0, v_1, \ldots, v_{n-1}, v_n = v_0$. First we prove some technical claims. Note that in these claims the subscripts of vertices of C are always considered modulo n. A G-edge (v_i, v_{i+k}), for $2 \leq k \leq \lfloor \frac{n}{2} \rfloor$, is called a *chord* of length k.

Claim 4.3.16 *If G contains a chord of length k then it contains all possible chords of length k.*

PROOF: Let (v_i, v_{i+k}) be a G-edge. Suppose that (v_{i+1}, v_{i+k+1}) is not a G-edge. Then the path $v_{i+1}, v_{i+2}, \ldots, v_{i+k}, v_i, v_{i-1}, v_{i-2}, \ldots, v_{i+k+1}$ is a Hamiltonian path with independent end-vertices, a contradiction. (Fig. 4.6(a)) Thus (v_{i+1}, v_{i+k+1}) is a chord of C. Then the claim follows by induction. □

Claim 4.3.17 *If G has a chord of length k, for some $2 \leq k \leq \lfloor \frac{n}{2} \rfloor - 2$, then G has a chord of length $k+2$.*

PROOF: Let (v_i, v_{i+k}) be a G-edge. Using Claim 4.3.16 we obtain that both (v_{i-1}, v_{i+k-1}) and (v_{i+1}, v_{i+k+1}) are G-edges. Suppose that (v_i, v_{k+2}) is not a G-edge. Then the path $v_i, v_{i+k}, v_{i+k+1}, v_{i+1}, v_{i+2}, \ldots, v_{i+k-1}, v_{i-1}, v_{i-2}, \ldots, v_{i+k+2}$ is a Hamiltonian path with independent end-vertices, a contradiction. (Fig. 4.6(b)) □

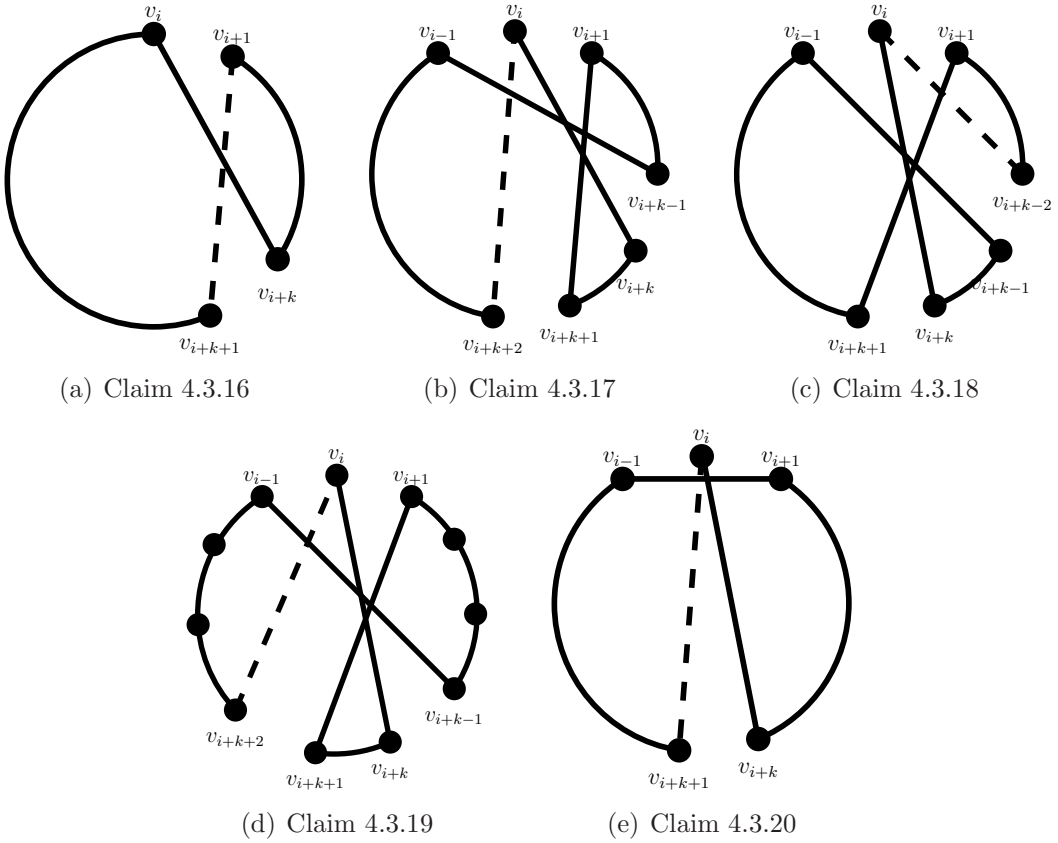
Claim 4.3.18 *If G has a chord of length k, for some $4 \leq k \leq \lfloor \frac{n}{2} \rfloor$, then G has a chord of length $k-2$.*

PROOF: Let (v_i, v_{i+k}) be a G-edge. Using Claim 4.3.16 we obtain that both (v_{i-1}, v_{i+k-1}) and (v_{i+1}, v_{i+k+1}) are G-edges. Suppose that (v_i, v_{k-2}) is not a G-edge. Then the path $v_i, v_{i+k}, v_{i+k-1}, v_{i-1}, v_{i-2}, \ldots, v_{i+k+1}, v_{i+1}, v_{i+2}, \ldots, v_{i+k-2}$ is a Hamiltonian path with independent end-vertices, a contradiction (Fig. 4.6(c)). □

Claim 4.3.19 *If n is odd and G contains a chord of length $2\lfloor \frac{n-3}{4} \rfloor + 1$ then G also contains a chord of length $2\lfloor \frac{n-1}{4} \rfloor$.*

PROOF: Let $k = 2\lfloor \frac{n-3}{4} \rfloor + 1$, and (v_i, v_{i+k}) be a G-edge. Using Claim 4.3.16 we obtain that both (v_{i-1}, v_{i+k-1}) and (v_{i+1}, v_{i+k+1}) are G-edges. Suppose that (v_i, v_{k+2}) is not a G-edge. Then the path $v_i, v_{i+k}, v_{i+k+1}, v_{i+1}, v_{i+2}, \ldots, v_{i+k-1}, v_{i-1}, v_{i-2}, \ldots, v_{i+k+2}$ is a Hamiltonian path with independent end-vertices, a contradiction. (Fig. 4.6(d)) This proves the claim as the length of the chord (v_i, v_{k+2}) is $2\lfloor \frac{n-1}{4} \rfloor$. □

Claim 4.3.20 *If G has a chord of length 2 then G is the complete graph K_n.*



(a) Claim 4.3.16 (b) Claim 4.3.17 (c) Claim 4.3.18

(d) Claim 4.3.19 (e) Claim 4.3.20

Figure 4.6: Dashed edges must be chords of G as no Hamiltonian path has independent ends

PROOF: Claim 4.3.16 implies that G contains all chords of length 2. Let (v_i, v_{i+k}) be a G-edge ($2 \leq k \leq \lfloor \frac{n}{2} \rfloor - 1$). We prove that in this case (v_i, v_{i+k+1}) is also a G-edge. This by induction on chord length and by Claim 4.3.16 proves the claim.

Suppose that (v_i, v_{i+k+1}) is not a G-edge. Then the path $v_i, v_{i+k}, v_{i+k-1}, \ldots, v_{i+1}$, $v_{i-1}, v_{i-2}, \ldots, v_{i+k+1}$ is a Hamiltonian path with independent end-vertices, a contradiction (Fig. 4.6(e)). □

Claim 4.3.21 *If G is neither bipartite nor an odd cycle then it contains an even-length chord.*

PROOF: If G is a cycle there is nothing to prove. Otherwise if n is odd let G contain a chord (v_i, v_{i+k}). If k is even we are done. If k is odd then, by Claim 4.3.17, the chord $2\lfloor \frac{n-3}{4} \rfloor + 1$ is in G and so, by Claim 4.3.19, G has a chord of length $2\lfloor \frac{n-1}{4} \rfloor$.

Suppose now that n is even. Let $C_{odd} = w_0, w_1, \ldots, w_s = w_0$ be an odd cycle in G. As s is odd, at least one chord (w_i, w_{i+1}) of C must have even length. This proves the claim. □

Now we can use the above claims to prove the theorem.

PROOF OF THEOREM 4.3.15: If $G = C_n$ we are done. If G has an odd-length chord then, by Claims 4.3.17, 4.3.18, and 4.3.16, G must have $K_{n,n}$ as a subgraph. Hence either $G = K_{n,n}$ or G has an odd cycle. In this latter case, by Claim 4.3.21, G must have an even-length chord and, by Claims 4.3.18, and 4.3.20, G is a complete graph. □

This theorem immediately implies that $\mathrm{li}(G) \geq 2$ if G is not isomorphic to K_n, C_n or $K_{n,n}$. It is easy to see that for these special graphs we have $\mathrm{li}(K_n) = \mathrm{li}(C_n) = 1$ and $\mathrm{li}(K_{n,n}) = n - 1$. Thus we obtain the following theorem.

Theorem 4.3.22 *Let G be any connected graph but the complete graph K_n and the cycle C_n. Then* $\mathrm{ml}(G) \leq \mathrm{li}(G)$.

4.3.4 Putting things together

The results of this chapter are used for proving approximation ratios in Chapter 5. To achieve this, we need one more claim.

Claim 4.3.23 *If G is a graph on n vertices then* $2(n - \alpha(G)) \geq n - \mathrm{sc}(G)$.

PROOF: Let X be an independent set of size $\alpha(G)$. Then

$$\mathrm{sc}(G) \geq \mathrm{comp}(G[X]) - |V \setminus X| = \alpha(G) - (n - \alpha(G)).$$

Subtracting both sides from n yields the claim. □

Summarizing the results of the chapter, Theorem 4.1.5, Theorem 4.3.22, Theorem 4.3.10, and Definition 4.3.9 imply

Corollary 4.3.24 *For any connected graph G but K_n and C_n:*

$$\mathrm{sc}(G) + 1 \leq \mathrm{ml}(G) \leq \mathrm{li}(G) = \mathrm{ca}(G) + 1 \leq \alpha(G).$$

Subtracting each measure from n and using Claim 4.3.23 we conclude:

Corollary 4.3.25 *For any connected graph G but K_n and C_n:*

$$n - \mathrm{sc}(G) - 1 \geq n - \mathrm{ml}(G) \geq n - \mathrm{li}(G) = n - \mathrm{ca}(G) - 1 \geq n - \alpha(G) \geq \frac{1}{2}(n - \mathrm{sc}(G)).$$

Observe that there is a difference of factor 2 between the two ends of this series of inequalities. This fact forms the basis of our approximation ratio proofs in Chapter 5. Using Corollary 4.3.13, it also shows that the internal vertices of any independence tree provide a 2-approximation for the MINIMUM CONNECTED VERTEX COVER problem. The original proof of this latter statement is due to Savage [60].

Chapter 5

Maximizing the Number of Internal Vertices

In this chapter we deal with the Maximum Internal Spanning Tree problem: given an undirected connected graph G we aim to find a spanning tree of G which maximizes the number of internal vertices. Clearly, this degree-based spanning tree optimization problem is a generalization of the Hamiltonian Path problem, since Hamiltonian paths (if exist) are exactly those spanning trees which have $|V(G)| - 2$ internal vertices. (As a tree always has at least 2 leaves, the number of its internal vertices can be at most $|V(G)| - 2$.)

In Section 5.2, we first present Algorithm ILST, an improvement of Algorithm DFS which yields a spanning tree with independent leaves. Then we prove in three different ways that Algorithm ILST provides a linear-time 2-approximation for the Maximum Internal Spanning Tree problem. The first proof shows that this approximation factor is a straightforward consequence of the results of Chapter 4. The second proof is a compact and self-contained one (having been our original proof for the approximation factor). The third proof uses a primal-dual linear programming approach which is further refined in Section 5.5 to prove that Algorithm LOST is a 7/4-approximation for the Maximum Internal Spanning Tree problem for graphs with no pendant vertices.

In Section 5.4, we consider two special classes of input graphs: claw-free graphs (graphs not containing $K_{1,3}$ as an induced subgraph) and cubic graphs (3-regular graphs). We introduce Algorithm RDFS, which is a refined version of Algorithm DFS, and which ensures an approximation factor of 3/2 if the input graph is claw-free, and a factor of 6/5 if the input graph is cubic. The running time of Algorithm RDFS is $\mathcal{O}(\Delta(G)|V(G)|)$.

In Section 5.5, we present Algorithm LOST which provides a 7/4-approximation for the Maximum Internal Spanning Tree problem, in $\mathcal{O}(|V(G)|^4)$ time, for input graphs with no pendant vertices. This algorithm is based on the successive execution of local improvement steps. Local improvement techniques have already

been applied for many NP-hard spanning tree optimization problems [29, 53, 54, 64]. To prove the approximation ratio, we use a linear-programming formulation and a primal-dual technique, the basic idea of which is introduced in Section 5.2.

Section 5.6 considers the vertex-weighted case: the MAXIMUM WEIGHTED INTERNAL SPANNING TREE problem. Here, the total weight of internal vertices is to be maximized. We present Algorithm WLOST, another local improvement based method, which provides a $(2\Delta(G) - 3)$-approximation in $\mathcal{O}(|V(G)|^4)$-time. A slight modification of this algorithm (called Algorithm RWLOST) ensures an $\mathcal{O}(|V(G)|^4)$-time 2-approximation whenever the input graph is claw-free. Both of these approximation factors are guaranteed only if the input graph has no pendant vertex.

Section 5.7 considers the MAXIMUM INTERNAL SPANNING TREE problem from an opposite point of view. Instead of looking for a spanning tree with as many internal vertices (or equivalently as few leaves) as possible, we upper bound the number of leaves at a number q and search for an $\leq q$-leaf subtree spanning as many vertices as possible. For this aim, we use a density condition on the input graph. In Section 5.8, we examine how density conditions can guarantee the existence of a q-leaf spanning tree in claw-free graphs.

In Section 5.9 we present the results of our experimental analysis on the MAXIMUM INTERNAL SPANNING TREE problem.

In what follows we give some basic definitions and introduce the notation used throughout this chapter. Let T be a spanning tree of G and l be a leaf of T. The branch of l is *short* if $b^-(l) = l$, that is, there is no forwarding vertex in $br(l)$. Otherwise the branch is called *long*. The leaf l itself is referred as short (long) if its branch is short (long). $L_{\rm s}(T)$ stands for the set of short leaves of T and $L_{\rm g}(T)$ for the set of long leaves of T. A leaf l is called *(x-)supported* if there is a non-tree edge (l, x) with $x \notin br(l)$. If l is a long leaf and there exists a non-tree edge (l, z) such that $z \in br(l)$ then $z^{\rightarrow l}$ is called an *l-leafish vertex*. In this case, the vertex z is called the *base* of $z^{\rightarrow l}$. The set of l-leafish vertices is denoted by $F(l)$. Observe that not every long leaf has a leafish vertex in its branch. We denote by $L_{\rm p}(T)$ the set of long leaves of T having no leafish vertex in their branch. We recall that $I(T)$ and $L(T)$ denote the internal vertices and the leaves of tree T, respectively.

5.1 Related Results

The MAXIMUM INTERNAL SPANNING TREE problem was already considered in the literature from different points of view. Fernau et al. [25, 26] gave exponential-time exact algorithms to solve it. The running time they achieve is $\mathcal{O}^*(c^n)$ for some $c \leq 3$ for general graphs, and $\mathcal{O}^*(1.8916^n)$ for the case when the input graph has $\Delta \leq 3$. Here we use the notation $\mathcal{O}^*(t(n))$ to denote the time complexity of $\mathcal{O}(t(n)\,\mathrm{poly}(n))$. Prieto and Sloper proved that the MAXIMUM INTERNAL SPANNING TREE problem is fixed parameter tractable [57, 58], that is, a spanning tree with at least k internal

vertices can be found in $f(k)\mathcal{O}(n^c) = \mathcal{O}^*(f(k))$ time if it exists. The exact running time of their algoritm is $\mathcal{O}^*(2^{3.5k \log k})$.

In the rest of this section, we recall a few optimization problems related to the Maximum Internal Spanning Tree problem. We also mention some problems where the methods we are using (local improvement, primal-dual LP-optimization) have been successfully applied to obtain approximability results.

The Minimum Leaf Spanning Tree problem consists of finding a spanning tree with a minimum number of leaves. This degree-based spanning tree optimization problem is clearly NP-hard, being a generalization of the Hamiltonian Path problem. Moreover, it is even hard to approximate: using a result of Karger et al. [45] on the Longest Path problem, Lu and Ravi [54] showed that no constant-factor approximation exists for it, unless P=NP. Observe that from an optimization point of view, the Minimum Leaf Spanning Tree problem is equivalent to the Maximum Internal Spanning Tree problem. The set of optimum solutions is the same for both problems, only the measure function is complemented yielding different approximability properties, as shown by the results of this chapter.

We describe our 7/4-approximation algorithm for the Maximum Internal Spanning Tree problem (for graphs with no pendant vertices) in Section 5.5. It applies local improvement rules in order to decrease the number of leaves. The idea of using local improvement to achieve a global result is quite common within the field of combinatorial optimization. Particularly, it was successfully used for many degree-based spanning tree optimization problems [29, 53, 54, 64]. One of them is the Minimum Degree Spanning Tree problem, which is to find a spanning tree T where the maximum T-degree is minimized. For this problem, the local improvement based algorithm of Fürer and Raghavachari [29] is the best possible (unless P=NP) as it finds a spanning tree whose maximum degree is at most one more than the optimum solution. Our second example is the Maximum Leaf Spanning Tree problem which is to find a spanning tree with a maximum number of leaves. Lu and Ravi have used the local improvement technique on a pre-built spanning tree to give an $\mathcal{O}(n^7)$-time 3-approximation for this problem. Later, they have proposed the idea of manipulating the local structure of the underlying graph [55] and so they have reduced the running time to be almost linear, at the same time preserving the approximation factor of 3. The currently known best approximation factor is 2: the algorithm of Solis-Oba [64] is based on the local improvement idea presented in [55].

Finally we mention that the proof of the approximation factor in Section 5.5 is based on a linear programming formulation of the Maximum Internal Spanning Tree problem, and on a primal-dual bounding approach. The main idea of this proof was inspired by the framework of Goemans and Williamson [36] who have successfully applied primal-dual approximation for a general spanning forest optimization problem.

5.2 A 2-approximation Algorithm

In this section we provide a linear-time 2-approximation algorithm for the Maximum Internal Spanning Tree problem. Given a graph G, our algorithm creates either a Hamiltonian path or an independence tree of the input graph G. The algorithm itself is based on Algorithm DFS.

Recall that a DFS-tree T is a spanning tree of G rooted at some vertex r. According to Claim 2.2.1, the d-leaves of T are G-independent. However, this does not guarantee that the leaves of T form a G-independent set if we consider T as a non-rooted spanning tree. It might happen that r is a leaf of T and is adjacent to some d-leaves. In this case, an additional replacement step is executed to make the leaves independent. Note that this step decreases the number of leaves by one. Thus the resulting spanning tree has at most as many leaves as the number of d-leaves of T.

Algorithm ILST (Independent Leaves Spanning Tree)

Input: An undirected graph $G = (V, E)$
Output: A Hamiltonian path or an independence tree of G
$T \leftarrow$ `DFSTraversal(`G`)` ; `// an arbitrary DFS-tree of` G
$r \leftarrow$ the root of T
if *T is not a Hamiltonian path and $d_T(r) = 1$ and there exists a d-leaf l such that $(r, l) \in E(G)$* **then**
  `//` r `is a leaf and is adjacent to another leaf` l
  Add edge (l, r) to T
  Delete edge $(b(l), b^-(l))$ from T
return T

Algorithm ILST produces a spanning tree. Indeed, the replacement step first creates a unique cycle by adding an edge to the tree and then removes an edge from this cycle. If the replacement step is executed then l and r become internal vertices and $b^-(l)$ becomes a leaf. Since $b^-(l)$ is not an ancestor of any other leaves, it is G-independent from them, see Claim 2.2.2, and so the obtained spanning tree T is an independence tree. The initial DFS-tree can be found in linear time. If we check $(r, l) \in E(G)$ for each d-leaf l during the traversal then the evaluation of the "if" condition needs only constant extra time. Once l is found, finding $b(l)$ and $b^-(l)$ and executing the replacement step needs linear time.

Thus we have proved

Claim 5.2.1 *Algorithm ILST gives either a Hamiltonian path or an independence tree of G in $\mathcal{O}(m)$ time.* □

In what follows we prove that Algorithm ILST is a 2-approximation algorithm for the Maximum Internal Spanning Tree problem. We provide three different

proofs for the approximation ratio. The first one is based on vulnerability parameters and uses the results of Chapter 4. The second one is a direct proof, while the third one is a linear programming based approach which is further enhanced in Section 5.5 where we give a 7/4-approximation algorithm for the MAXIMUM INTERNAL SPANNING TREE problem for graphs with no pendant vertices.

Theorem 5.2.2 *Algorithm ILST is a linear-time 2-approximation algorithm for the* MAXIMUM INTERNAL SPANNING TREE *problem.*

Note that the running time is already proved in Claim 5.2.1, we only have to show that any independence tree ensures the required approximation ratio, that is,

Claim 5.2.3 *If T is an independence tree of G, and T^* is an optimum solution of the* MAXIMUM INTERNAL SPANNING TREE *problem in G then $|I(T^*)| \leq 2|I(T)|$.*

We prove this claim in three different ways in the next three subsections.

5.2.1 Proof 1: via vulnerability parameters

Let T be the independence tree of G that Algorithm ILST outputs. Then we have $\mathrm{ml}(G) \leq |V_1(T)| \leq \mathrm{li}(G)$. Thus using Corollary 4.3.25 we have

$$\frac{|I(T^*)|}{|I(T)|} = \frac{n - \mathrm{ml}(G)}{n - |V_1(T)|} \leq \frac{n - \mathrm{ml}(G)}{n - \mathrm{li}(G)} \leq \frac{2(n - \alpha(G))}{n - \alpha(G)} = 2.$$

We can conclude that the approximation ratio is a trivial consequence of the results of Chapter 4.

5.2.2 Proof 2: the direct way

Let the spanning tree T be the output of Algorithm ILST, and T^* be an optimum solution of the MAXIMUM INTERNAL SPANNING TREE problem in G.

If $|V_1(T)| \leq n/2$ then we are done, as in this case

$$|I(T)| = n - |V_1(T)| \geq n/2 > |I(T^*)|/2.$$

Thus, we can suppose that $|V_1(T)| = n/2 + k$, for some $k > 0$, and so we have $|I(T)| = n/2 - k$. Now we count those edges of T^* that are between $V_1(T)$ and $I(T)$. To this aim, let $S = e_{T^*}(V_1(T), I(T))$ denote the set of these edges. Observe that an arbitrary vertex $v \in V_1(T)$ is either a leaf or an internal vertex of T^*. In the former case, there is a single edge in S which is incident to v, in the latter case there are at least two of them. (Here we use the fact that $V_1(T)$ is a G-independent set by the construction of Algorithm ILST.) Hence we have

$$n - 1 \geq |S| \geq |V_1(T)| + |V_1(T) \cap I(T^*)|.$$

We can reformulate this as

$$|V_1(T) \cap I(T^*)| \leq n - |V_1(T)| = n/2 - k.$$

This implies that

$$|I(T^*)| = |I(T^*) \cap V_1(T)| + |I(T^*) \cap I(T)| \leq n/2 - k + n/2 - k = 2|I(T)|,$$

yielding the approximation factor of 2.

5.2.3 Proof 3: via linear programming

To prove the approximation ratio we construct a linear programming formulation of the problem such that each spanning tree T' of G defines a feasible solution with a value of $|I(T')|$. Especially, the solution corresponding to the optimal tree T^* has a value of $|I(T^*)|$. Our goal is to establish an approximation factor β, that is, to prove that for a spanning tree T created by some polynomial-time algorithm, we have $\beta|I(T)| \geq |I(T^*)|$. For this aim we use a primal-dual LP-approach. It is enough to give a particular dual solution with a value of $\beta|I(T)|$, as in this case

$$|I(T^*)| \leq \text{ LP-optimum } \leq \beta|I(T)|$$

ensures the approximation factor.

In this section, we provide an LP-description of both the primal and the dual problems. Based on the spanning tree T, we give a simple primal and a simple dual solution. As a result, we obtain $\beta = 2$ for Algorithm ILST. In Section 5.5, when analyzing Algorithm LOST, we will use the same problem formulations, and the same primal solution. However, by giving two more complex dual solutions, we will achieve $\beta = 7/4$ for graphs with no pendant vertices.

Let us recall a formulation of the spanning tree polyhedron [62]:

$$\mathcal{SP}(G) = \{x \mid \forall S \subseteq V : x(S) \leq |S| - 1, -x(V) \leq -(|V| - 1), \forall e \in E : 0 \leq x(e)\},$$

where $x(S) = \sum_{e \in E(G[S])} x(e)$ is the sum of x over all edges spanned by S.

Let us consider the following linear program (called primal problem):

$$\begin{aligned}
&\text{maximize} && \sum_{v \in V} z(v) && \\
&\text{subject to} && x \in \mathcal{SP}(G) && \\
& && -\sum_{e \in \delta(v)} x(e) + z(v) \leq -1 && \text{for all } v \in V \qquad (5.1)\\
& && 0 \leq z(v) \leq 1 && \text{for all } v \in V
\end{aligned}$$

Notice that each integer solution of this program is composed of the characteristic vector of edges (x) and internal vertices (z) of a spanning tree. We use this fact

to create a feasible primal LP-solution $\mathbf{P}$ based on an optimum solution T^* of the MAXIMUM INTERNAL SPANNING TREE problem.

We set $x(e) = 1$ when e is an edge of T^*, and $z(v) = 1$ when v is an internal vertex of T^*. All other variables are set to 0. Since T^* is a spanning tree, it is easy to see that this solution is feasible and has a value of $\text{val}(\mathbf{P}) = |I(T^*)|$. Indeed, the characteristic vector of the T^*-edges is by definition in the spanning tree polyhedron $\mathcal{SP}(G)$, so we only have to check (5.1). By the choice of $x(e)$, we know that $\sum_{e\in\delta(v)} x(e)$ is exactly the T^*-degree of v, and so (5.1) requires $z(v)$ to be at most $d_{T^*}(v) - 1$, which is true by its definition.

If $\mathbf{P}^*$ is the optimum solution of the primal LP-problem itself then we obtain

$$|I(T^*)| = \text{val}(\mathbf{P}) \leq \text{val}(\mathbf{P}^*). \tag{5.2}$$

The dual of the above program is:

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{S\subseteq V}(|S|-1)y(S) - (|V|-1)t - \sum_{v\in V} w(v) + \sum_{v\in V} r(v) & \\
\text{subject to} & \displaystyle\sum_{e\in E(G[S])} y(S) - t - \sum_{e\in\delta(v)} w(v) \geq 0 & \text{for all } e \in E \\
& w(v) + r(v) \geq 1 & \text{for all } v \in V \\
& y(S), t, w(v), r(v) \geq 0 & \text{for all } S \subseteq V, v \in V
\end{array} \tag{5.3}$$

Based on the spanning tree T, we construct a dual solution $\mathbf{D}$ as follows. Let $y(V) = 1$, $w(v) = 1$ for each $v \in L(T)$, and $r(v) = 1$ for each $v \in V \setminus (L(T))$. All other variables are set to 0.

To see the feasibility of this solution, it is enough to check (5.3) for all G-edges. If (u, v) is such an edge that $w(u) = 0$ or $w(v) = 0$ then (5.3) is satisfied due to $y(V) = 1$. Thus only the edges with $w(u) = w(v) = 1$, that is, the edges of $G[L(T)]$ could violate the inequality. However, as T is an independence tree, there is no such G-edge, the leaves of T form a G-independent set. As a result, the dual solution $\mathbf{D}$ is feasible. Its value is

$$\text{val}(\mathbf{D}) = n - 1 - |L(T)| + n - |L(T)| = 2(n - |L(T)|) - 1 = 2|I(T)| - 1.$$

Using the duality theorem of linear programming we obtain

$$|I^*(T)| = \text{val}(\mathbf{P}) \leq \text{val}(\mathbf{P}^*) \leq \text{val}(\mathbf{D}) \leq 2|I(T)|,$$

and thus

$$\frac{|I(T^*)|}{|I(T)|} \leq 2,$$

proving the approximation ratio of 2.

5.2.4 Algorithm ILST in regular graphs

This approximation ratio of 2 can be further improved if the input graph is cubic or 4-regular. In this case, we have

Theorem 5.2.4 *Algorithm ILST provides an $\frac{r+1}{3}$-approximation for the r-regular graphs. In particular, the approximation factor is 4/3 for cubic graphs and 5/3 for 4-regular graphs.*

PROOF: Let T be a spanning tree of G provided by Algorithm ILST. We double count the non-tree edges between $L(T)$ and $I(T)$. On one hand, due to the r-regularity, $(r-1)|L(T)|$ non-tree edges leave $L(T)$. On the other hand, at most $(r-2)|V_2(T)| + (r-3)|V_{\geq 3}(T)|$ non-tree edges can enter into $I(T)$. Thus, using Claim 2.1.1, we have

$$(r-1)|L(T)| \leq (r-2)|V_2(T)| + (r-3)|V_{\geq 3}(T)| \leq (r-2)|V_2(T)| + (r-3)(|L(T)|-2),$$

or equivalently

$$|V_2(T)| \geq \frac{2}{r-2}|L(T)| + \frac{2(r-3)}{r-2}. \tag{5.4}$$

By Claim 2.1.2, we have

$$|L(T)| = n - |V_2(T)| - |V_{\geq 3}(T)| \leq n - |V_2(T)| - \frac{1}{r-2}|L(T)| + \frac{2}{r-2}.$$

Hence

$$\left(1 + \frac{1}{r-2}\right)|L(T)| \leq n - |V_2(T)| + \frac{2}{r-2},$$

and so

$$|I(T)| = n - |L(T)| \geq \frac{1}{r-1}(n + (r-2)|V_2(T)| - 2).$$

Substituting (5.4) we get

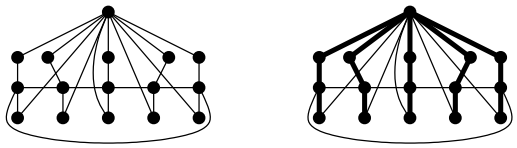
$$|I(T)| \geq \frac{1}{r-1}(n + 2(n - |I(T)|) + 2(r-4)),$$

which yields

$$|I(T)| \geq \frac{3n + 2r - 8}{r+1} > \frac{3}{r+1}(n-2),$$

proving the theorem. □

In Section 5.4, we present Algorithm RDFS which improves the approximation ratio for cubic graphs from 4/3 to 6/5.



(a) (b)

Figure 5.1: A graph where no run of Algorithm DFS can find an optimum solution

5.3 More About Traversals

In this section, we focus on traversal algorithms discussed in Section 2.2 and compare them from the point of view of the Maximum Internal Spanning Tree problem. As mentioned earlier, Algorithm DFS and Algorithm FIFO-DFS are both specializations of Algorithm Greedy Traversal. The difference is in the mechanism how we get the next edge to traverse after stepping back from a vertex with no unvisited neighbors. This edge comes from the EdgeRepository which has deterministic behavior in the two specialized algorithms and indeterministic in the more general one.

We have seen in Section 5.2 that slightly modifying Algorithm DFS we obtain a 2-approximation algorithm for the Maximum Internal Spanning Tree problem. Now let us consider the graph on $3k+1$ vertices shown in Figure 5.1(a). It is easy to check that Algorithm ILST can lead to $k+1$, $k+2$, or $k+3$ internal vertices depending on the traversing order. This shows two things. First, the fact that the bound for the approximation ratio is tight. Indeed, Figure 5.1(b) shows an optimum solution having $2k+1$ internal vertices. Second, there exists a graph where the optimum solution of the Maximum Internal Spanning Tree problem cannot be obtained by using Algorithm DFS as the main building block of Algorithm ILST. Notice that if we change the traversal algorithm used in Algorithm ILST to Algorithm FIFO-DFS, we can achieve the optimum spanning tree of this graph. However, it is still not known, if this is the case for every graph.

In the rest of this section we prove that an optimum solution can always be obtained by an appropriate run of Algorithm Greedy Traversal. Obviously, this does not imply that the problem itself is optimally solvable in polynomial time, since the algorithm is non-deterministic.

Theorem 5.3.1 *There exists an optimum solution T of the* Maximum Internal Spanning Tree *problem which can be obtained as the output of a run of Algorithm Greedy Traversal.*

Proof: Let T be an optimum solution of the Maximum Internal Spanning Tree problem, and let us choose a vertex r to be the root of T. Let $h_T(l)$ denote the number of edges of $P_T(l,r)$ for a leaf l. Subject to the optimality of T, we choose T and r such that $\sum_{l \in L(T)} |h_T(l)|^2$ is maximized.

We can suppose that the d-leaves of T are sorted in the non-increasing order of their distance from r, namely we have $|h_T(l_1)| \geq |h_T(l_2)| \geq \cdots \geq |h_T(l_k)|$. Now we traverse T from r by visiting the unvisited part of each path $P_T(r, l_i)$ starting with $i = 1$ up to $i = k$. We now show that this traversal is a greedy one, or equivalently, that we never step back from a vertex which has unvisited G-neighbors. As we step back only from the d-leaves, it is enough to show that all G-neighbors of a d-leaf l_i are in $V(\cup_{j=1}^{i} P_T(r, l_j))$.

Suppose that l_i has a G-neighbor $x \notin V(\cup_{j=1}^{i} P_T(r, l_j))$. Let us create the tree $T' = T + (l_i, x) - (x, x^{\to r})$. If $d_T(x^{\to r}) > 2$ then T' has less leaves than T, which is a contradiction. Thus $x^{\to r}$ is a leaf of T' with $h_{T'}(x^{\to r}) < h_T(l_i)$. Moreover, all leaves which are descendents of x are farther from r in T' than in T. Therefore $\sum_{l \in L(T')} |h_{T'}(l)|^2 > \sum_{l \in L(T)} |h_T(l)|^2$, which is a contradiction. □

Suppose that we know how to resolve the non-determinism occurring in our traversal algorithms when looking for an optimum solution of the Maximum Internal Spanning Tree problem. As we have seen, in this case, the general Algorithm Greedy Traversal can always find an optimum solution, while all run of Algorithm DFS might provide a spanning tree being equally far from the optimum. We conclude this section by an open question asking how well Algorithm FIFO-DFS works from this point of view. The experimental behavior of the aforementioned specializations of Algorithm Greedy Traversal is analyzed in Section 5.9.

5.4 Claw-Free and Cubic Graphs

In this section we consider two special classes of input graphs, claw-free graphs (graphs not containing $K_{1,3}$ as an induced subgraph) and cubic graphs (3-regular graphs). We can make use of the special structure of such graphs in order to improve our approximation algorithm and so the approximation ratio for the Maximum Internal Spanning Tree problem. More precisely, we present Algorithm RDFS, a refined version of Algorithm DFS and then we prove that it approximates the Maximum Internal Spanning Tree problem within a factor of 3/2 for claw-free graphs and within a factor of 6/5 for cubic graphs. This latter factor improves that of 4/3 presented in Subsection 5.2.4.

The research leading to the results of this section is joint work with Gábor Wiener. In particular, the RDFS-rule of Algorithm RDFS (see below) was originally his idea.

Algorithm RDFS is a depth first search in which we specify how to choose the next vertex of the traversal in the cases when Algorithm DFS itself would choose arbitrarily from several candidates. The main idea is to select the vertex that has the minimum number of non-visited neighbors. For this purpose, we use the array ActDeg to maintain the number of non-visited neighbors of each vertex.

Algorithm RDFS differs from Algorithm DFS only at line $(*)$ where Algorithm DFS would choose a non-visited neighbor of v arbitrarily while Algorithm RDFS

Algorithm RDFS (Refined Depth First Search)

```
Input: A simple connected graph G
Output: A rooted spanning tree T of G (called RDFS-tree)
begin
    T ← (V, ∅)
    foreach v ∈ V(G) do
        VisitingRank[v] ← 0
        ActDeg[v] ← d_G(v)
    NextVisitingRank ← 1
    EdgeStack ← ∅
    r ← an arbitrary vertex of G
    VisitVertex(r)
    return T
end

// Traversing from a vertex v
function VisitVertex(v)
begin
    VisitingRank[v] ← NextVisitingRank
    foreach neighbor z of v do
        ActDeg[z] ← ActDeg[z] − 1
    NextVisitingRank ← NextVisitingRank + 1
    if v has a neighbor w such that VisitingRank[w] = 0 then
*       from all such w's, choose the one minimizing ActDeg[.]
        x ← v; y ← w
    else
        if EdgeStack is not empty then
            (x, y) ← the "first" element of EdgeStack such that
            NextVisitingRank[y] = 0
        else
            return
    Add (x, y) to T
    foreach neighbor w of x such that w ≠ y and NextVisitingRank[w] = 0 do
        Add (x, w) to EdgeStack
    VisitVertex(y)
end
```

uses its own selection rule, the so called RDFS-rule. Regarding the running times, recall that Algorithm DFS runs in linear time. At line $(*)$ of Algorithm RDFS, we make at most Δ steps to find the minimum and this line is executed at most once for each edge of T. Thus the total running time of Algorithm RDFS is $\mathcal{O}(\Delta n)$.

A tree produced by Algorithm RDFS is called an *RDFS-tree*. As Algorithm RDFS is a special version of Algorithm DFS, Claim 2.2.2 can be applied to RDFS-trees. That is, each G-edge connects two vertices of which one is the ancestor of the other in the RDFS-tree.

We use the following notation. Let T be an RDFS-tree, and let l be a d-leaf of T such that $d_G(l) \geq 2$. (If T has no such l then T is a minimum leaf spanning tree.) Let $c(l)$ stand for the neighbor of l having the highest VisitingRank value such that $(l, c(l))$ is in $E(G) \setminus E(T)$. Since $d_G(l) \geq 2$, such $c(l)$ must exist. Let $g(l) = c(l)^{\rightarrow l}$ denote the neighbor of $c(l)$ along the path $P_T(c(l), l)$. Claim 2.2.2 implies that $c(l)$ must be an ancestor of l and so $g(l)$ is a child of $c(l)$.

Now we prove a useful lemma about the T-degree of $g(l)$.

Lemma 5.4.1 *Let T be an RDFS-tree and let l be a d-leaf of T. Then $d_T(g(l)) = 2$.*

Proof: For a given vertex v, let us denote by Y_v the set of vertices having VisitingRank greater than or equal to VisitingRank$[v]$. It is obvious that $l \in Y_{c(l)}$, and also $l \in Y_{g(l)}$.

Consider now the step of Algorithm RDFS when we choose $g(l)$ to be the next visited vertex. By the RDFS-rule,

$$d_{G[Y_{g(l)}]}(g(l)) \leq d_{G[Y_{g(l)}]}(l).$$

By the definition of $c(l)$ and $g(l)$, there is no non-tree edge incident to l in $G[Y_{g(l)}]$, and thus

$$d_{G[Y_{g(l)}]}(l) = 1,$$

implying

$$d_{G[Y_{g(l)}]}(g(l)) = 1,$$

since $d_{G[Y_{g(l)}]}(g(l)) \geq 1$ is obvious. Therefore $g(l)$ has only one child, and one parent, namely $c(l)$, and so

$$d_T(g(l)) = 2.$$

□

Now we prove the approximation ratio for claw-free graphs.

Theorem 5.4.2 *Algorithm RDFS is an $\mathcal{O}(\Delta n)$-time 3/2-approximation algorithm for the* Maximum Internal Spanning Tree *problem for claw-free graphs.*

Proof: We have already seen the running time, so we only have to check the approximation ratio. Let G be an arbitrary connected claw-free graph on n vertices and let T be an RDFS-tree of G. First notice that $d_T(v) \leq 3$ for any $v \in V(T)$.

Otherwise, v and three of its children would induce a subgraph $K_{1,3}$ in G because of Claim 2.2.2. Thus our aim is to show that

$$|V_2(T)| + |V_{\geq 3}(T)| \geq \frac{2}{3}|I(T^*)|,$$

where T^* is an optimal spanning tree. Since T is a tree (having a total vertex degree of $2n-2$), and $d_T(v) \leq 3$ for any $v \in V(T)$, we have

$$|V_1(T)| = |V_{\geq 3}(T)| + 2.$$

Now we would like to find many vertices of T-degree 2 in order to show that the number of internal vertices is large. For this aim, we use Lemma 5.4.1. The problem is that the vertices $g(l)$, having T-degree 2, are not necessarily distinct for every d-leaf of T with G-degree ≥ 2. However, we show that they are all distinct in claw-free graphs.

Lemma 5.4.3 *Let T be an RDFS-tree of G and let l and l' be d-leaves of T such that $d_G(l) \geq 2$ and $d_G(l') \geq 2$. Then $g(l) \neq g(l')$.*

Proof: Suppose to the contrary that $g(l) = g(l')$. This implies $c(l) = c(l')$ as $c(l)$ is the parent of $g(l)$ and $c'(l)$ is the parent of $g'(l)$. Now consider the induced subgraph $S = G[\{c(l), l, l', g(l)\}]$. On one hand, the vertices l, l' and $g(l)$ are all G-neighbors of $c(l)$. On the other hand, l and l' are d-leaves of T, thus they cannot be adjacent in G. Moreover, $g(l)$ can be adjacent neither to l nor to l' in $G \setminus T$, because of the selection of $c(l)$ as clearly $\mathsf{VisitingRank}[g(l)] > \mathsf{VisitingRank}[c(l) = c(l')]$. Since $(l, g(l))$ and $(l', g(l))$ are not T-edges either (otherwise $g(l)$ could not be a common ancestor of l and l'), the induced subgraph S is isomorphic to $K_{1,3}$, a contradiction. □

So we have found as many vertices of T-degree 2 as the number of those d-leaves that have a G-degree of at least 2. Let us denote by p the number of pendant vertices of G. These vertices are clearly leaves of any spanning tree of G, so the optimum spanning tree has at most $\min(n-p, n-2)$ internal vertices.

We consider two cases.

Case 1: $p = 0$. Now every d-leaf has a G-degree of at least 2, thus

$$|V_2(T)| \geq |V_1(T)| - 1.$$

Since

$$|V_1(T)| = |V_{\geq 3}(T)| + 2$$

and

$$n = |V_1(T)| + |V_2(T)| + |V_{\geq 3}(T)|,$$

after some elementary computation we obtain

$$|V_2(T)| + |V_{\geq 3}(T)| \geq \frac{2}{3}(n-2) \geq \frac{2}{3}|I(T^*)|.$$

Case 2: $p \geq 1$. It suffices to show that

$$|V_2(T)| + |V_{\geq 3}(T)| \geq \frac{2}{3}(|V_1(T)| + |V_2(T)| + |V_{\geq 3}(T)| - p).$$

Since the G-degree of the root of the RDFS-tree T is minimum, the root is a pendant vertex of G. Thus by Lemma 5.4.3 we have

$$|V_2(T)| \geq |V_1(T)| - p,$$

so it suffices to show that

$$|V_2(T)| + |V_{\geq 3}(T)| \geq \frac{2}{3}(2|V_2(T)| + |V_{\geq 3}(T)|),$$

which is equivalent to

$$|V_{\geq 3}(T)| \geq |V_2(T)|,$$

and to

$$|V_1(T)| - 2 \geq |V_2(T)|.$$

If this inequality holds then we are done. Otherwise

$$|V_2(T)| \geq |V_1(T)| - 1.$$

From here

$$|V_2(T)| + |V_{\geq 3}(T)| \geq \frac{2}{3}(n-2) \geq \frac{2}{3}|I(T^*)|$$

follows just like in Case 1. □

Now we turn to cubic input graphs and prove that Algorithm RDFS guarantees an even better approximation factor for them.

Theorem 5.4.4 *Algorithm RDFS is a linear time 6/5-approximation algorithm for the* Maximum Internal Spanning Tree *problem for cubic graphs.*

Proof: We have seen that the running time is $\mathcal{O}(n\Delta)$, which is linear for cubic graphs where $\Delta = 3$. Now we check the approximation ratio. Let G be an arbitrary connected cubic graph on n vertices and let T be an RDFS-tree of G.

Let l be a d-leaf of T. Now l has 2 neighbors in $G \setminus T$, one of them is $c(l)$, call the other one $c'(l)$. It is obvious that $d_T(c(l)) = 2$ and also $d_T(c'(l)) = 2$ if $c'(l)$ is not the root. Furthermore, $d_T(g(l)) = 2$ by Lemma 5.4.1. Now let $h(l)$ be the only neighbor of $g(l)$ in $G \setminus T$. As a consequence of Claim 2.2.2 we obtain that one of $g(l)$ and $h(l)$ is an ancestor of the other. Now, $h(l)$ must be an ancestor of $g(l)$. Indeed, consider the moment when we visit $c(l)$ and choose the next vertex to be visited according to the RDFS-rule. At that moment, l has a single unvisited neighbor, and so the fact that $g(l)$ is chosen implies that it has a single unvisited neighbor, too, that is, $h(l)$ must have already been visited before $g(l)$. This yields that either $h(l)$ is the root or $d_T(h(l)) = 2$.

We conclude that vertices $c(l), c'(l), g(l), h(l)$ are all distinct and have a T-degree of 2 with the only possible exception of $c'(l) = h(l)$ being the root of T and having T-degree of 1. If we take another d-leaf l' of T, we find that $c(l) \neq c(l')$, $c'(l) \neq c'(l')$ if $c'(l)$ is not the root, $g(l) \neq g(l')$, and $h(l) \neq h(l')$ if $h(l)$ is not the root. The first two of these are due to the fact that the graph is cubic, the third is an implication of the first, and the fourth is implied by the third. Based on this, it is easy to check that if l and l' are two distinct d-leaves then the sets $\{c(l), c'(l), g(l), h(l)\}$ and $\{c(l'), c'(l'), g(l'), h(l')\}$ can only have the root of T as a common element.

This way we associate 4 vertices (namely $c(l), c'(l), g(l)$, and $h(l)$) to each and every d-leaf l. Among these vertices only the root can show up more than once and all the other vertices have degree 2. Since the root can occur at most twice (because of 3-regularity) and the number of d-leaves is at least $|V_1(T)| - 1$, we have found $4(|V_1(T)| - 1) - 2$ distinct vertices of degree 2, that is,

$$|V_2(T)| \geq 4|V_1(T)| - 6. \tag{5.5}$$

On the other hand, $d_T(v) \leq 3$ for any $v \in V(T) = V(G)$, thus, as the total T-degree of the vertices is $2n - 2$, we have

$$|V_1(T)| = |V_{\geq 3}(T)| + 2. \tag{5.6}$$

Using (5.5) and (5.6) we obtain

$$n = |V_1(T)| + |V_2(T)| + |V_{\geq 3}(T)| \geq |V_1(T)| + 4|V_1(T)| - 6 + |V_1(T)| - 2 = 6|V_1(T)| - 8,$$

and so

$$|V_1(T)| \leq \frac{n}{6} + \frac{4}{3}.$$

Therefore

$$|I(T)| = n - |V_1(T)| \geq \frac{5}{6}n - \frac{4}{3} \geq \frac{5}{6}(n-2) \geq \frac{5}{6}|I(T^*)|,$$

as the optimum tree T^* can have at most $(n-2)$ internal vertices. □

5.5 A 7/4-approximation Algorithm

In this section we give an approximation algorithm for the Maximum Internal Spanning Tree problem which improves the approximation ratio of 2 to 7/4 whenever the input graph has no pendant vertices.

Our algorithm is based on the technique of local improvement rules. It starts by building a spanning tree then it successively executes local changes (defined by improvement rules) as far as possible. When no more rule can be executed we obtain a locally optimal spanning tree (LOST). A LOST has some specific properties which can be used to prove the approximation ratio.

The proof of the approximation ratio is based on a primal-dual technique of linear programming which has already been presented in Subsection 5.2.3. Recall that the main idea is to build a primal program such that each of its integer solutions is composed of the characteristic vector of a spanning tree and its internal vertices. Thus, an optimum solution T^* of the MAXIMUM INTERNAL SPANNING TREE problem defines a feasible integer primal solution with a value of $|I(T^*)|$. Then we use dual solutions to upper bound this quantity by means of $|I(T)|$, where T is the LOST that our algorithm outputs. Finally, the upper bounds are used to prove the approximation ratio.

The primal and dual problem formulations, and the primal solution used in this section are identical to those of Subsection 5.2.3. However, in this section we construct two more complex dual solutions in order to enhance the upper bound on $|I(T^*)|$ and so the approximation ratio itself. The first dual solution bounds the approximation factor if T has many short branches, the second one does this job if T has many long branches. Both dual solutions are based on a set S of forwarding vertices of T that are G-independent from the leaves.

5.5.1 Local improvement rules

The structure of the improvement rules is as follows. A precondition part determines when the particular rule can be executed. An action part gives the replacement of some (1 or 2) edges of T by non-tree edges and it might execute another (embedded) rule, too. The execution of a rule yields a new spanning tree T'. In the case of Rules 1–13 the tree T' has less leaves than T had, while in the case of Rules 14–15 trees T' and T have the same number of leaves.

We define a LOST to be a spanning tree T in which none of these improvement rules can be executed, that is, the preconditions of the rules are not satisfied. If T has many short branches then the violated preconditions of Rules 1–7, while if T has many long branches then the violated preconditions of Rules 1, 2 and 8–15 ensure a set S which is G-independent from the leaves and whose size is big enough to yield the approximation ratio via a suitable solution of the dual LP-problem. We now turn to give the local improvement rules. Notice that Rules 2–4 have already been used in Section 4.2 to show some basic properties of a minimum leaf spanning tree.

For the definitions and the used notation, see pp. 12–13 and p. 48.

Rule 1. Precondition: T has two leaves l_1 and l_2 such that $(l_1, l_2) \in E(G)$ and that $P_T(l_1, l_2)$ has two neighboring branchings x, y. **Action:** Let $E(T') = E(T) + (l_1, l_2) - (x, y)$. (See Fig. 5.2(a)).

Rule 2. Precondition: T has two leaves l_1 and l_2 such that $(l_1, l_2) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, l_2) - (b(l_1), b^-(l_1))$. (See Fig. 5.2(b)).

Rule 3. Precondition: T has an x-supported leaf l_1 such that $d_T(x^{\rightarrow l_1}) > 2$. **Action:** Let $E(T') = E(T) + (l_1, x) - (x, x^{\rightarrow l_1})$. (See Fig. 5.2(c)).

Rule 4. Precondition: T has an x-supported leaf l_1 and a leaf l_2 such that

$d_T(x^{\to l_1}) = 2$, and that $(l_2, x^{\to l_1})$ is a non-tree edge. **Action:** Let $E(T') = E(T) + (l_1, x) - (x, x^{\to l_1})$. Then execute Rule 1 or Rule 2 on leaves l_2, $x^{\to l_1}$. (See Fig. 5.2(d)).

Rule 5. Precondition: T has an x-supported leaf l such that $d_T(b(l)^{\to x}) > 2$. **Action:** Let $E(T') = E(T) + (l, x) - (b(l), b(l)^{\to x})$. (See Fig. 5.2(e)).

Rule 6. Precondition: T has an x-supported leaf l_1 and a leaf l_2 such that $d_T(b(l_1)^{\to x}) = 2$, and that $(l_2, b(l_1)^{\to x})$ is a non-tree edge. **Action:** Let $E(T') = E(T) + (l_1, x) - (b(l_1), b(l_1)^{\to x})$. Then execute Rule 1 or Rule 2 on leaves l_2, $b(l_1)^{\to x}$. (See Fig. 5.2(f)).

Rule 7. Precondition: T has a short leaf l, and an edge (x, y) such that (l, x) and (l, y) are both non-tree edges. **Action:** Let $E(T') = E(T) + (l, x) + (l, y) - (x, y) - (l, b(l))$. (See Fig. 5.2(g)).

Rule 8. Precondition: T has a long leaf l_1 and a leaf l_2 such that $(b^-(l_1), l_2) \in E(G)$. **Action:** Let $E(T') = E(T) + (b^-(l_1), l_2) - (b^-(l_1), b(l_1))$. (See Fig. 5.2(h)).

Rule 9. Precondition: T has an x-supported long leaf l_1 and a long leaf l_2 such that $x \notin br(l_2) - b(l_2)$, $b(l_1) \neq b(l_2)$, and $(b^-(l_1), b^-(l_2)) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, x) + (b^-(l_1), b^-(l_2)) - (b(l_1), b^-(l_1)) - (b(l_2), b^-(l_2))$. (See Fig. 5.2(i)).

Rule 10. Precondition: T has an x-supported long leaf l_1 and a long leaf l_2 with $b(l_1) = b(l_2)$ such that $d_T(b(l_1)) \geq 4$, $x \notin br(l_2)$, and $(b^-(l_1), b^-(l_2)) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, x) + (b^-(l_1), b^-(l_2)) - (b(l_1), b^-(l_1)) - (b(l_2), b^-(l_2))$. (See Fig. 5.2(j)).

The following rule differs from the above ones as, while executed on the long branch of a leaf l, it changes neither the trunk nor any other branch of T. Only the branch of l is modified such that one of its leafish vertices becomes leaf and l becomes an internal vertex. This rule is not used in its own but as a building block of Rules 11–13 which can decrease the number of leaves using a leafish vertex.

Rule A. Precondition: T has a leaf l and an l-leafish vertex $x^{\to l}$ with base x. **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$. (See Fig. 5.2(k)).

We can use Rule A to decrease the number of leaves as follows.

Rule 11. Precondition: T has two leaves l_1 and l_2, and an l_1-leafish vertex u such that (u, l_2) is a non-tree edge. **Action:** Execute Rule A on u to make it a leaf and l_1 an internal vertex. Then execute Rule 1 or Rule 2 on leaves l_2, u.

Rule 12. Precondition: T has two leaves l_1 and l_2, an l_1-leafish vertex u, and an l_2-leafish vertex v such that (u, v) is a non-tree edge. **Action:** Execute Rule A on u and on v to make both of them a leaf while l_1 and l_2 an internal vertex. Then execute Rule 1 or Rule 2 on leaves u, v.

Rule 13. Precondition: T has two leaves l_1 and l_2, and an l_1-leafish vertex u such that $(u, b^-(l_2))$ is a non-tree edge. **Action:** Execute Rule A on u to make it a leaf and l_1 an internal vertex. Then execute Rule 8 on leaves u, l_2.

Rules 14 and 15 can be executed only on pairs of branches that contain no leafish vertices. Although the number of leaves does not change when executing them, the sum $\sum_{l \in L_p} |br(l)|$ is always decreased. This is a crucial point of our running time analysis that can be found in Subsection 5.5.4.

Rule 14. Precondition: T has two leaves $l_1, l_2 \in L_p(T)$ and two non-tree edges (l_1, x) and (l_2, y) such that $x \in br(l_2)$ and $y \in br(l_1)$. **Action:** Let $E(T') = E(T) + (l_1, x) - (x, x^{\rightarrow}(l_1))$. (See Fig. 5.2(l)).

Rule 15. Precondition: T has at least four leaves, and there exist $l_1, l_2 \in L_p(T)$ such that $b(l_1) = b(l_2)$, $d_T(b(l_1)) = 3$ and $(b^-(l_1), b^-(l_2))$ is a non-tree edge. **Action:** Let $E(T') = E(T) + (b^-(l_1), b^-(l_2)) - (b(l_2), b^-(l_2))$. (See Fig. 5.2(m)).

5.5.2 Locally optimal spanning trees, The algorithm

Definition 5.5.1 *A spanning tree T is a* locally optimal spanning tree (LOST) *if none of Rules 1–15 can be executed on it.*

Now the approximation algorithm for the Maximum Internal Spanning Tree problem is straightforward using the above improvement rules.

Algorithm LOST (Locally Optimal Spanning Tree)

Input: A simple connected graph G
Output: A locally optimal spanning tree (LOST) T of G
begin
  Run Algorithm Greedy Traversal to obtain an initial spanning tree T
  Execute Rules 1–15 as long as possible. If several rules can be executed, choose the one with the lowest number
end

Now we can state the main theorem of this section.

Theorem 5.5.2 *Algorithm LOST is an $\mathcal{O}(|V|^4)$-time 7/4-approximation for the* Maximum Internal Spanning Tree *problem for graphs that have no pendant vertices.*

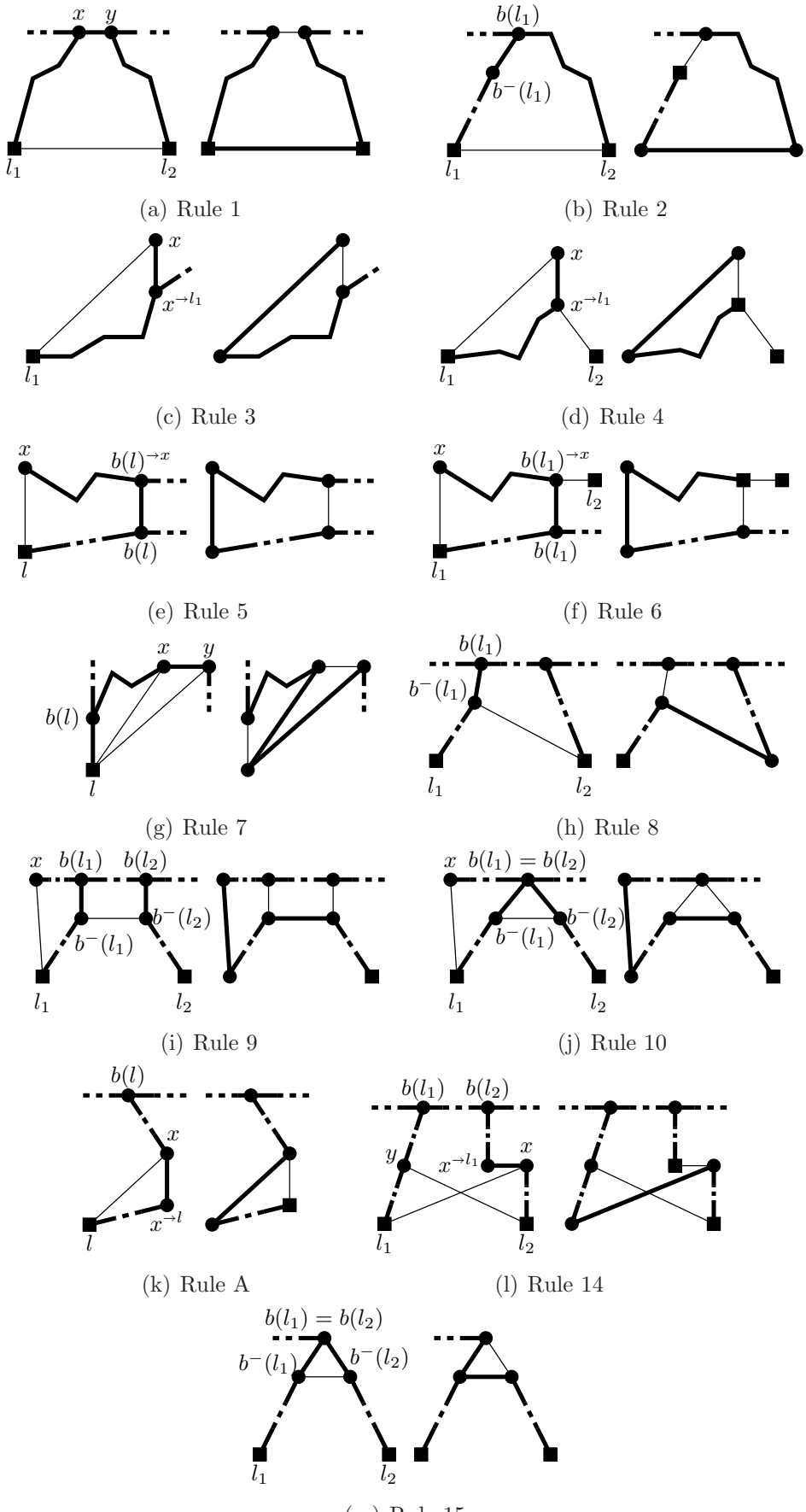
The proof of the running time is postponed to Subsection 5.5.4. The approximation ratio is established by the following lemma.

Lemma 5.5.3 *Let T be a LOST of a graph G that has no pendant vertices, and let T^* be a spanning tree of G with a maximum number of internal vertices. Then*

$$\frac{|I(T^*)|}{|I(T)|} \leq 7/4.$$

First observe some basic properties of T which are immediate consequences of the definition of a LOST.

Property 1. $L(T)$ forms a G-independent set. (As Rules 1 and 2 are no more applicable in T.)



(a) Rule 1 (b) Rule 2

(c) Rule 3 (d) Rule 4

(e) Rule 5 (f) Rule 6

(g) Rule 7 (h) Rule 8

(i) Rule 9 (j) Rule 10

(k) Rule A (l) Rule 14

(m) Rule 15

Figure 5.2: Local improvement steps for creating a LOST (squares represent leaves, circles represent internal vertices)

Property 2. Let l be an x-supported leaf. Then $d_T(x^{\to l}) = d_T(b(l)^{\to x}) = 2$. Furthermore, both $x^{\to l}$ and $b(l)^{\to x}$ are G-independent from the set $L(T) - l$. (As Rules 3–6 are no more applicable in T.)

Property 3. Let l be a short leaf. Then no two G-neighbors of l are T-neighbors. (As Rule 7 is no more applicable in T.)

Property 4. Let l_1 be a long leaf. Then no leaf $l_2 \neq l_1$ is a G-neighbor of $b^-(l_1)$. (As Rule 8 is no more applicable in T.)

Property 5. Let l_1 and l_2 be leaves. Then l_2 is G-independent from the set of l_1-leafish vertices. (As Rule 11 is no more applicable in T.)

Property 6. Let l_1 and l_2 be leaves. Then each l_1-leafish vertex is G-independent from the set of l_2-leafish vertices. (As Rule 12 is no more applicable in T.)

Property 7. Let l_1 and l_2 be leaves. Then $b^-(l_2)$ is G-independent from the set of l_1-leafish vertices. (As Rule 13 is no more applicable in T.)

Observe that minimum leaf spanning trees must have Properties 1–7, as the rules which decrease the number of leaves cannot be applied on them.

Property 8. Let l_1 and l_2 be long leaves such that their branches do not contain leafish vertices, and $(b^-(l_1), b^-(l_2))$ is a non-tree edge. Then $b(l_1) = b(l_2)$ and $d_T(b(l_1)) = 3$. (As Rules 9, 10, and 14 are no more applicable in T.) Moreover, since Rule 15 is no more applicable, T must have exactly 3 leaves. From this point we suppose that T has at least 4 leaves. As a result, $l_1, l_2 \in L_{\mathrm{p}}(T)$ implies $(b^-(l_1), b^-(l_2)) \notin E(G)$.

If the LOST T has only 3 leaves then it is trivially a 7/4-approximation for the Maximum Internal Spanning Tree problem. Indeed, $\frac{n-2}{n-3} < \frac{7}{4}$ holds for $n > 4$, and if $n \leq 4$ then T is a LOST only if it is an optimum tree.

5.5.3 Proof of the approximation factor with a primal-dual technique

Though we have already given the description of our primal and dual programs in Subsection 5.2.3, we repeat them here for the sake of readability.

The spanning tree polyhedron [62] is defined as:

$$\mathcal{SP}(G) = \{x \mid \forall S \subseteq V : x(S) \leq |S| - 1, -x(V) \leq -(|V| - 1), \forall e \in E : 0 \leq x(e)\},$$

where $x(S) = \sum_{e \in E(G[S])} x(e)$ is the sum of x over all edges spanned by S.

The primal linear program is:

$$\begin{aligned}
&\text{maximize} && \sum_{v \in V} z(v) && \\
&\text{subject to} && x \in \mathcal{SP}(G) && \\
& && -\sum_{e \in \delta(v)} x(e) + z(v) \leq -1 && \text{for all } v \in V \\
& && 0 \leq z(v) \leq 1 && \text{for all } v \in V
\end{aligned}$$

As we have seen in Subsection 5.2.3, this has a solution $\mathbf{P}$ with value $\mathrm{val}(\mathbf{P}) = |I(T^*)|$, where T^* is an optimum solution of the MAXIMUM INTERNAL SPANNING TREE problem. Therefore we have

$$|I(T^*)| = \mathrm{val}(\mathbf{P}) \leq \mathrm{val}(\mathbf{P}^*), \tag{5.7}$$

where P^* is an optimum solution of the primal LP-problem itself.

We have also seen that the dual of the above program is:

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{S\subseteq V}(|S|-1)y(S) - (|V|-1)t - \sum_{v\in V} w(v) + \sum_{v\in V} r(v) & \\
\text{subject to} & \displaystyle\sum_{e\in E(G[S])} y(S) - t - \sum_{e\in\delta(v)} w(v) \geq 0 & \text{for all } e \in E \\
& w(v) + r(v) \geq 1 & \text{for all } v \in V \\
& y(S), t, w(v), r(v) \geq 0 & \text{for all } S \subseteq V, v \in V
\end{array} \tag{5.8}$$

Now let T be the tree created by Algorithm LOST. We consider two different dual solutions corresponding to T. The first one is used when T has many short branches, and the second one is used when T has many long branches. At first, let us assume that T has both short and long branches. Later, we will examine how to modify the proof when this assumption fails.

Let l be a short leaf. We define the set

$$Q(l) = \left\{x^{\rightarrow l} \mid (l,x) \in E(G) \setminus E(T)\right\} \cup \left\{b(l)^{\rightarrow x} \mid (l,x) \in E(G) \setminus E(T)\right\}.$$

Note that $Q(l) \neq \emptyset$, since $d_G(l) \geq 2$. Let

$$Q = \cup_{l\in L_s(T)} Q(l).$$

The first dual solution $\mathbf{D}_1$ is constructed as follows. Let $y(V) = 1$, $y(Q) = 1$, $w(v) = 1$ for each $v \in L(T) \cup Q$, and $r(v) = 1$ for each $v \in V \setminus (L(T) \cup Q)$. All other variables are set to 0. Figure 5.3(a) shows the way how vertices with $w(.) = 1$ are determined.

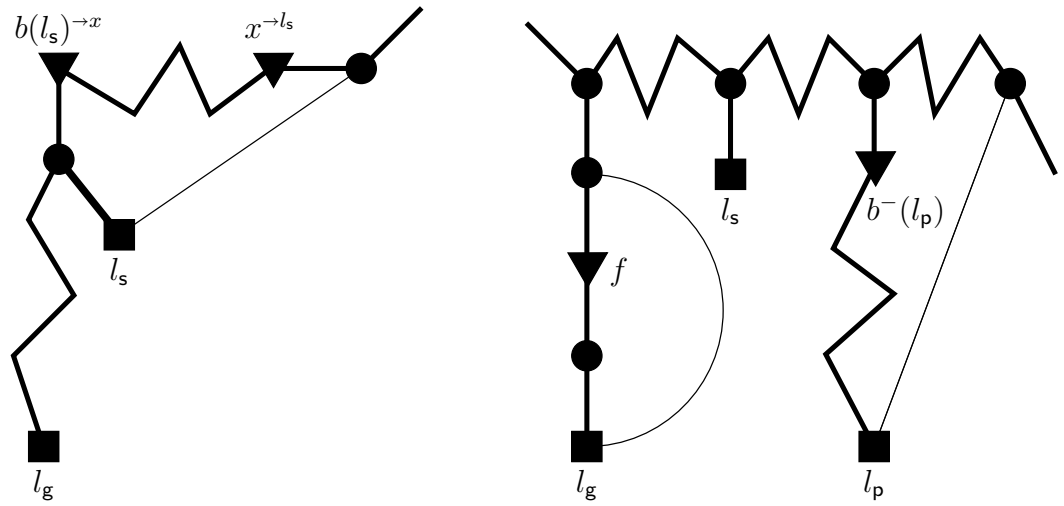
To see the feasibility of this solution, it is enough to check (5.8) for all edges of G. Since $y(V) = 1$, only the edges of $G[L(T) \cup Q]$ could violate the inequality as they are the only edges with $w = 1$ on both ends. However, by Property 1, there is no edge spanned by $L(T)$, and by Properties 2 and 3, there is no edge between $L(T)$ and Q. Thus we only have to cover the edges inside Q by a set with $y = 1$. We choose Q itself for this role, that is, $y(Q) = 1$ ensures the feasibility.

Let us define

$$c_1 = \frac{|Q|}{|I(T)|}.$$

The value of the first dual solution is

$$\begin{aligned}
\mathrm{val}(\mathbf{D}_1) &= |V| - 1 + |Q| - 1 - |L(T)| - |Q| + |V| - |L(T)| - |Q| \\
&= 2(|I(T)| - 1) - |Q| < (2 - c_1)|I(T)|.
\end{aligned}$$



(a) (b)

Figure 5.3: Construction of the dual solutions; Squares represent leaves, triangles represent other vertices with $w(.) = 1$

To construct the second dual solution $\mathbf{D}_2$ we denote the set of l-leafish vertices by $F(l)$, and define the set of leafish vertices to be

$$F = \cup_{l \in L_g(T)} F(l).$$

Recall that $L_p(T)$ denotes the set of long leaves having no leafish vertex in their branch. We use B_p^- to denote the set $\cup_{l \in L_p(T)} b^-(l)$. We immediately obtain $|B_p^-| = |L_p(T)|$.

Dual variables are set as: $y(V) = 1$, $y(F(l) + l) = 1$ for each $l \in L_g(T) \setminus L_p(T)$, $y(\{l, b^-(l)\}) = 1$ for each $l \in L_p(T)$, $w(v) = 1$ for each $v \in (L(T) \cup F \cup B_p^-)$, $r(v) = 1$ for each $v \in V \setminus (L(T) \cup F \cup B_p^-)$. All other variables are set to 0. Figure 5.3(a) shows the way how vertices with $w(.) = 1$ are determined.

To see the feasibility of this solution, it is enough to check (5.8) for all edges of G. As $y(V) = 1$, only the edges of $G[(L(T) \cup F \cup B_p^-)]$ could violate the inequality. However, by Properties 1 and 4–8, the graph $G[(L(T) \cup F \cup B_p^-)]$ has no edge between different branches of T. On the other hand, the edges of $G[(L(T) \cup F \cup B_p^-)]$ within a single branch of T are also covered by some set having $y = 1$.

Now let us define

$$c_2 = \frac{|V| - |L_s(T)|}{|I(T)|}.$$

Since $|I(T)| = |V| - |L_s(T)| - |L_g(T)|$ we obtain $|L_g(T)| = (c_2 - 1)|I(T)|$.

The value of the solution is

$$\begin{aligned}\mathrm{val}(\mathbf{D}_2) = |V| - 1 + |F| + |L_{\mathrm{p}}(T)| - |L(T)| - |F| - |L_{\mathrm{p}}(T)| \\ + |V| - |L(T)| - |F| - |L_{\mathrm{p}}(T)| < 2|I(T)| - |F| - |L_{\mathrm{p}}(T)| \\ \leq 2|I(T)| - |L_{\mathrm{g}}(T)| = (3 - c_2)|I(T)|.\end{aligned}$$

Here we have used that $|L_{\mathrm{g}}(T)| \leq |F| + |L_{\mathrm{p}}(T)|$. The duality theorem of linear programming yields

$$\mathrm{val}(\mathbf{P}^*) \leq \min\left(\mathrm{val}(\mathbf{D}_1), \mathrm{val}(\mathbf{D}_2)\right).$$

Then (5.2) implies that

$$|I(T^*)| \leq \min\left(\mathrm{val}(\mathbf{D}_1), \mathrm{val}(\mathbf{D}_2)\right).$$

As a result we have

$$\frac{|I(T^*)|}{|I(T)|} \leq \min\left(2 - c_1, 3 - c_2\right). \tag{5.9}$$

Now let $N(L_{\mathrm{s}}(T))$ be the set of G-neighbors of short leaves. Observe that by the definition of Q, each element of $N(L_{\mathrm{s}}(T))$ has a T-neighbor in Q. Moreover, by Property 2, all vertices of Q are forwarding vertices of T. Thus

$$|N(L_{\mathrm{s}}(T))| \leq 2|Q| = 2c_1|I(T)|. \tag{5.10}$$

Let us emphasize here that the condition $d_G(l) \geq 2$ is used to ensure that the set $Q(l)$ is not empty, for each leaf l. This latter fact is necessary to upper bound $|N(L_{\mathrm{s}}(T))|$ by a function of $|Q|$.

Theorem 4.1.5 can be formulated to our case as

$$|I(T^*)| \leq |V| - \mathrm{sc}(G) - 1. \tag{5.11}$$

As the short leaves of T are G-independent, $G[V - N(L_{\mathrm{s}}(T))]$ has at least $|L_{\mathrm{s}}(T)|$ components implying that $\mathrm{sc}(G) \geq |L_{\mathrm{s}}(T)| - |N(L_{\mathrm{s}}(T))|$.

Thus, by (5.10), Definition 4.1.1 and (5.11), we have

$$|I(T^*)| < |V| - \mathrm{sc}(G) \leq |V| - |L_{\mathrm{s}}(T)| + |N(L_{\mathrm{s}}(T))| \leq (c_2 + 2c_1)|I(T)|. \tag{5.12}$$

If $c_1 \geq 1/4$ or $c_2 \geq 5/4$ then by (5.9), otherwise by (5.12), we obtain $\frac{|I(T^*)|}{|I(T)|} \leq 7/4$.

In order to finish the proof we have to consider the cases when T has only short or only long branches.

If all branches of T are short then the first dual solution still shows

$$\frac{|I^*(T)|}{|I(T)|} \leq 2 - c_1. \tag{5.13}$$

However, the second dual solution is not defined as it is based on long leaves of T. Also, Inequality (5.12) must be reformulated to

$$\begin{aligned}|I(T^*)| < |V| - \mathrm{sc}(G) \leq |V| - |L_{\mathrm{s}}(T)| + |N(L_{\mathrm{s}}(T))| \\ = |I(T)| + |N(L_{\mathrm{s}}(T))| \leq (1 + 2c_1)|I(T)|.\end{aligned} \tag{5.14}$$

Now, if $c_1 \geq 1/3$ then Inequality (5.13), otherwise Inequality (5.14) yields $\frac{|I^*(T)|}{|I(T)|} \leq 5/3 < 7/4$.

If T has only long branches then both the first dual solution, and the considerations leading to Inequality (5.12) are pointless. However, the upper bound

$$\frac{|I^*(T)|}{|I(T)|} \leq 3 - c_2 \tag{5.15}$$

is still valid with $c_2 = \frac{|V|}{|I(T)|}$, as $L_s(T) = \emptyset$. Now, if $c_2 \geq 3/2$ then $\frac{|I^*(T)|}{|I(T)|} \leq 3/2 < 7/4$ and we are done. If $c_2 \leq 3/2$ then $|I(T)| \geq 2/3|V| \geq 2/3|I^*(T)|$, that is, again $\frac{|I^*(T)|}{|I(T)|} \leq 3/2 < 7/4$.

5.5.4 Running time analysis

In this section we give some implementation details and a brief running time analysis of Algorithm LOST. We suppose that graph G is given by its adjacency matrix. For all of our algorithms, we maintain some additional data structures representing the spanning tree T under construction. The purpose of these data structures is to ease the precondition-checking, and to find the next rule to execute. The data structures are updated after each rule execution step.

Beside the adjacency matrix of G we use the following data structures while running our algorithms.

1. We maintain the edge-list of the current spanning tree T. This represents the solution to be constructed. This list is built in $\mathcal{O}(|V|^2)$-time during the initial traversal. As each rule adds and removes only a constant number of edges, the list can be updated in $\mathcal{O}(|V|)$ time after each rule execution step.

2. We have a list of leaves of T and in addition for each leaf l we store:

   (a) $b(l)$

   (b) the vertex $x^{\rightarrow l}$ for every vertex x

   (c) the vertex $b(l)^{\rightarrow x}$ for every vertex x

   This *leaf-information* structure is a navigation tool which is useful when an improvement step is based on a non-tree edge incident to a leaf. The construction of this structure can be done by running a traversal from each leaf. This must be done both initially and after each rule execution and needs $\mathcal{O}(|V|^2)$ time per run. The maintenance of the leaf-list itself needs only constant time after each rule execution.

3. For each forwarding vertex v, we store the leaf whose branch contains v. A special indicator is used instead when v is a trunk-vertex. This structure is created and maintained together with the leaf-information structure without any extra time need.

As a consequence, we obtain that the update of the used data structures needs $\mathcal{O}(|V|^2)$ time after each rule execution.

Some of the preconditions need to determine the list of short (or long) leaves. This can be done in $\mathcal{O}(|V|)$ time by iterating through the list of leaves and checking for each leaf l whether $b^-(l) = l$. If yes then l is short otherwise l is long. Note that $b^-(l)$ can be determined in constant time using the leaf-information structure as $b^-(l) = b(l)^{\to l}$.

Leafish vertices play an important role in Algorithm LOST. Their list can be built in $\mathcal{O}(|V|)$ time. To this aim, firstly we check for each long leaf l whether $(l, b(l)) \in E(G)$. If yes then we add $b^-(l)$ to the list of leafish vertices. Secondly, for each forwarding vertex x we determine the branch that contains x. If $x \in br(l)$ for some l and $(x, l) \in E(G)$ then we add $x^{\to l}$ to the list of leafish vertices. When building the list of leafish vertices, we can also create a list to represent the leaves in $L_{\mathrm{p}}(T)$. Initially this list contains all leaves of T. Then when a vertex $x \in br(l)$ is found to be leafish then l is removed from the list. As a result, we can build the list representing $L_{\mathrm{p}}(T)$ in $\mathcal{O}(|V|)$ time.

The precondition part of each rule is tested as follows.

Rule 1: We consult the leaf-list and for every pair l_1, l_2 of leaves we check in the adjacency matrix whether $(l_1, l_2) \in E(G)$. Then using the leaf-information structure, we walk through path $P_T(l_1, l_2)$ and look for a neighboring pair of branchings on it. The total time requirement is $\mathcal{O}(|V|^2)$.

Rule 2: We consult the leaf-list and for every pair l_1, l_2 of leaves we check in the adjacency matrix whether $(l_1, l_2) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time.

Rule 3: For each leaf l we get the list of non-tree edges from the adjacency matrix. Then for each non-tree edge (l, x), we check $x \notin br(l)$ and $d_T(x^{\to l}) > 2$. We use the leaf-information structure to get $x^{\to l}$ from x. The degree is checked in the edge-list of T. Thus we need constant time for a given (l, x) pair, and $\mathcal{O}(|V|^2)$ time in total.

Rule 4: For each leaf l_1 we get the list of non-tree edges from the adjacency matrix. Then for each non-tree edge (l_1, x), we check $x \notin br(l)$ and $d_T(x^{\to l_1}) = 2$. Unfortunately, if we directly check the adjacency between $x^{\to l_1}$ and each leaf l_2 then we need $\mathcal{O}(|V|^3)$ time in total. Therefore instead of doing this, we build a list from vertices $x^{\to l_1}$. This can be done in $\mathcal{O}(|V|^2)$ time. Note that each vertex is contained at most once in this list. The rule can be executed if there is a leaf l_2 and an element v of the list such that $(l_2, v) \in E(G)$. This check can be done in $\mathcal{O}(|V|^2)$ time which is also the total time requirement for this rule.

Rule 5: For every leaf l, and for every non-tree edge (l, x) we check $x \notin br(l)$ then determine $b(l)^{\to x}$ and check $d_T(b(l)^{\to x}) > 2$ in constant time using the leaf-information structure. The total time needed is $\mathcal{O}(|V|^2)$.

Rule 6: For every leaf l_1, and for every non-tree edge (l_1, x) we check $x \notin br(l_1)$ then determine $b(l_1)^{\to x}$ and check $d_T(b(l_1)^{\to x}) = 2$ in constant time using the leaf-information structure. Then, analogously to the case of Rule 4, we use an extra data

structure to find a leaf l_2 which is neighboring to $b(l_1)^{\to x}$. The total time requirement is $\mathcal{O}(|V|^2)$.

Rule 7: For every tree edge (x, y), and for every short leaf l, we check whether both (x, l), and (y, l) are edges of G. This needs $\mathcal{O}(|V|^2)$ time.

Rule 8: For every pair of long leaves l_1, l_2 we check whether $(l_1, b^-(l_2)) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time.

Rule 9: We need some extra consideration at this rule to avoid the unnecessary increase of time-complexity. First we build a $|L_{\rm g}(T)| \times |L_{\rm g}(T)|$-matrix M to indicate the pairs of long leaves for which the rule is possibly applicable. $\mathrm{M}(l_1, l_2)$ is set to 1 whenever $(b^-(l_1), b^-(l_2)) \in E(G)$ and $b(l_1) \neq b(l_2)$. All other elements of M are 0. This can be done in $\mathcal{O}(|V|^2)$ time. Then for each leaf l_1 we consider each non-tree edge (l_1, x) for which $x \notin br(l_1)$. There are three cases: (i) there is a non-tree edge (l_1, x) with x being a trunk-vertex; (ii) there are two non-tree edges (l_1, x) and (l_1, y) with x and y in different branches; (iii) all non-tree edges (l_1, x) end in the same branch of l_2. In the first two cases, the rule can be executed to l_1 and any other leaf. In the third case we set $\mathrm{M}(l_1, l_2)$ to 0 indicating that l_1 and l_2 cannot be used together to execute the rule. This checking process needs $\mathcal{O}(|V|^2)$ time. Finally, we look for a pair (l_1, l_2) with $\mathrm{M}(l_1, l_2) = 1$, and execute the rule on them. The total time requirement is $\mathcal{O}(|V|^2)$.

Rule 10: We apply the idea of Rule 9. The only difference is in the initial construction of matrix M. Here, we set $\mathrm{M}(l_1, l_2) = 1$ if $d_T(b(l_1)) \geq 4$, $(b^-(l_1), b^-(l_2)) \in E(G)$ and $b(l_1) = b(l_2)$. The total time requirement is again $\mathcal{O}(|V|^2)$.

Rule 11: For each leaf l and leafish vertex u we check whether $(l, u) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time. The total time requirement after the implied execution of Rule 1 or Rule 2 remains $\mathcal{O}(|V|^2)$.

Rule 12: For every pair u, v of leafish vertices we check that u and v are in different branches and that $(u, v) \in E(G)$. This is done in $\mathcal{O}(|V|^2)$ time. The implied execution of Rule 1 or Rule 2 does not increase this time need.

Rule 13: For each leaf l and for each leafish vertex u we check $u \notin br(l)$ and $(u, b^-(l)) \in E(G)$. This needs $\mathcal{O}(|V|^2)$ time. This time complexity is not increased by the implied execution of Rule 8.

Rule 14: Firstly we create an $|L_{\rm p}(T)| \times |L_{\rm p}(T)|$ matrix M and for each leaf $l_1 \in L_{\rm p}(T)$ we set $\mathrm{M}(l_1, l_2)$ to 1 if there is a non-tree edge (l_1, x) such that $x \in br(l_2)$. All other elements of M are 0. This is done in $\mathcal{O}(|V|^2)$ time. Then we check whether there exist two leaves $l_1, l_2 \in L_{\rm p}(T)$ such that $\mathrm{M}(l_1, l_2) = \mathrm{M}(l_2, l_1) = 1$. If yes then the rule can be executed to l_1 and l_2. The total time requirement is $\mathcal{O}(|V|^2)$.

Rule 15: For each pair of leaves $l_1, l_2 \in L_{\rm p}(T)$ we check in constant time the followings: $b(l_1) = b(l_2)$, $d_T(b(l_1)) = 3$ and $(b^-(l_1), b^-(l_2)) \in E(G)$. The total time requirement is $\mathcal{O}(|V|^2)$.

As a result we can conclude that the precondition of every single rule can be checked in $\mathcal{O}(|V|^2)$ time. After each rule execution data structures can be updated in $\mathcal{O}(|V|^2)$ time. Thus, to establish the time requirement of $\mathcal{O}(|V|^4)$ for Algorithm LOST, it is enough to see that at most $\mathcal{O}(|V|^2)$ rules are executed before the algo-

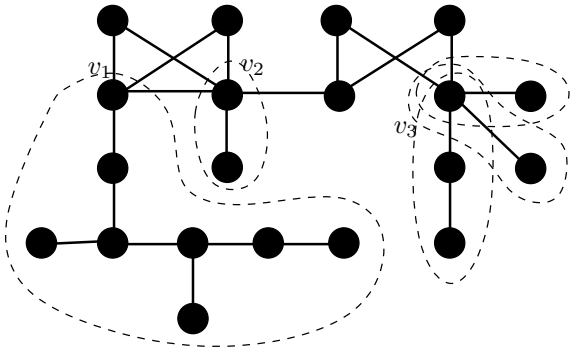


Figure 5.4: Pendant branches

rithm stops and a LOST is found. Clearly, all of Rules 1–13 decrease the number of leaves as a result of their execution. Therefore, the total number of their executions is $\mathcal{O}(|V|)$. Rules 14 and 15 do not change the number of leaves. They strictly decrease, however, the sum $\sum_{l \in L_p} |br(l)|$, that is, the total length of branches having no leafish vertices. Without changing the number of leaves, this sum can be decreased $\mathcal{O}(|V|)$ times. (It can happen that the execution of one of Rules 1–13 increases this sum, but in this case the number of leaves is decreased.) We conclude that there can be $\mathcal{O}(|V|^2)$ rule execution steps and so the running time of Algorithm LOST is $\mathcal{O}(|V|^4)$.

5.5.5 Pendant vertices

The analysis of the approximation ratio for the Algorithm LOST of Subsection 5.5.2 works only if the graph has no pendant vertices. In this subsection we refine this condition. To this aim, we recursively define *pendant branches* of a graph G as follows. Let P_0 be the set of pendant vertices of G. For $i = 1, 2, \ldots$, let P'_i be the pendant vertices of $G[V - P_{i-1}]$ and let $P_i = P'_i \cup P_{i-1}$. As G is not a tree, there is a smallest k for which $P_k = P_{k-1}$. Let $C_1, C_2, \ldots, C_r$ be the components of $G[P_k]$. Each C_j has a unique neighbor in $G[V - P_k]$, say $v_{f(j)}$. Then the pendant branches of G are $C_j + v_{f(j)}$ for $j = 1, 2, \ldots, r$. A pendant branch is long if it has at least three vertices. Figure 5.4 shows a graph and its pendant branches of which two are long. Notice that several pendant branches contain vertex v_3.

We build a graph G' from G as follows: first we remove all vertices of $\bigcup_{j=1}^{r} C_j$. Then for each vertex $v_{f(j)} \in R$ we add two vertices $x_{f(j)}$ and $y_{f(j)}$, and three edges $(v_{f(j)}, x_{f(j)})$, $(v_{f(j)}, y_{f(j)})$ and $(x_{f(j)}, y_{f(j)})$. If G is the graph of Figure 5.4, we obtain G' as shown in Figure 5.5(a). We run Algorithm LOST on G' instead of G obtaining a spanning tree T'. Since G' has no pendant vertices the approximation ratio of 7/4 applies to it. Let the spanning tree T of G be the union of the pendant branches of G and the restriction of T' to $V(G)$. Figure 5.5(b) shows the spanning tree yielded from the graph of Figure 5.4.

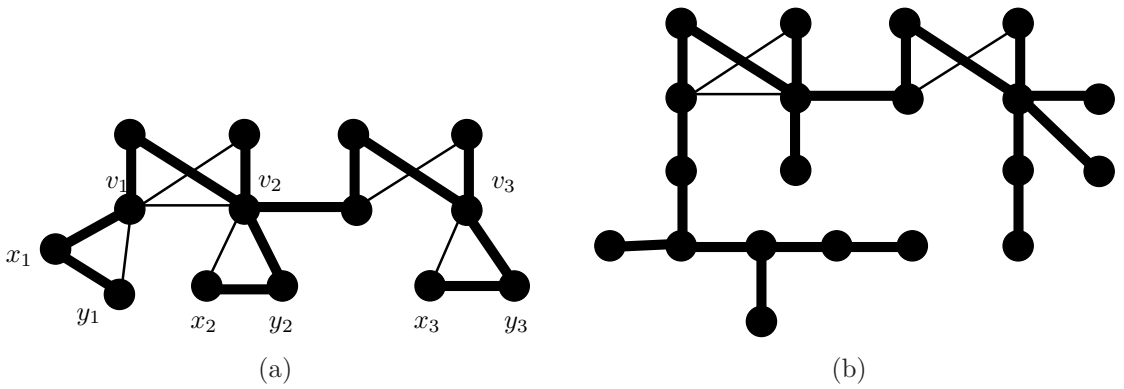


Figure 5.5: Running Algorithm LOST for graphs with pendant vertices

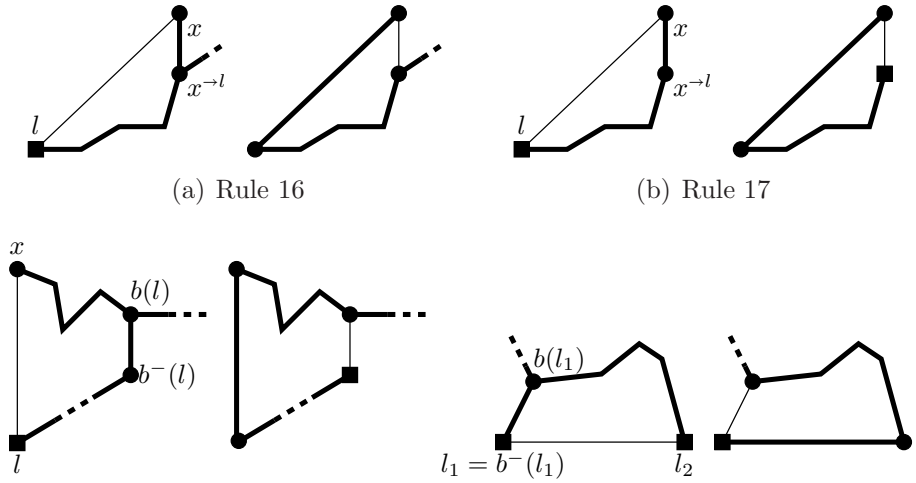
Now let p be the total number of internal vertices of the long pendant branches of T. We show that this algorithm provides a 7/4-approximation for the MAXIMUM INTERNAL SPANNING TREE problem whenever $p \geq r$.

By Lemma 5.5.3, we have $\frac{|I(T'^*)|}{|I(T')|} \leq 7/4$ as G' has no pendant vertices. Here, $I(T'^*)$ is an optimum solution of the MAXIMUM INTERNAL SPANNING TREE problem in G'.

Observe that, for all j, vertex v_j is an internal vertex of T as a pendant branch is attached to it, and is an internal vertex of T' due to the above construction. Also, for each j, one of x_j, y_j is a leaf, the other is an internal vertex of T'. Using the definition of p we obtain $|I(T)| = |I(T')| - r + p$, and $|I(T^*)| = |I(T'^*)| - r + p$, and so $\frac{|I(T^*)|}{|I(T)|} \leq \frac{|I(T'^*)|}{|I(T')|} \leq 7/4$ as $p \geq r$.

5.6 Vertex-Weighted Case

Let $G = (V, E)$ be a graph without pendant vertices and with a positive weight-function $c : V \longrightarrow \mathbb{Q}_+$ on its vertices. The MAXIMUM WEIGHTED INTERNAL SPANNING TREE problem aims to find a spanning tree T of G that maximizes the weight-sum of internal vertices, that is, $c(I(T)) = \sum_{v \in I(T)} c(v)$. Obviously, this problem is NP-hard, as it contains the unweighted version, the MAXIMUM INTERNAL SPANNING TREE problem, as a special case. In this section we present a $(2\Delta - 3)$-approximation algorithm for general graphs which is further refined to get a 2-approximation algorithm for claw-free graphs. The main idea of the algorithms is similar to the one we have seen in the unweighted case. We use local improvement steps to obtain a locally optimal tree. Then we prove that certain properties of such a tree guarantee the desired approximation ratio. The proof is, however, different from that of Section 5.5. Instead of using linear programming, it is based on mapping every leaf of the spanning tree into an internal vertex with larger weight.



(a) Rule 16 (b) Rule 17

(c) Rule 18 (d) Rule 19

Figure 5.6: Local improvement rules for creating a WLOST (squares represent leaves, circles represent internal vertices)

5.6.1 General graphs

Let us consider an arbitrary spanning tree T of G. In order to get a good approximation we execute local improvement rules as long as possible. Each such rule either turns a leaf into an internal vertex or it replaces a leaf with another one of smaller weight. The weighted sum of leaves decreases in both cases. Similarly to the unweighted case, each of the following rules has a precondition and an action part of the same semantics. For the definitions and the used notation, see pp. 12–13 and p. 48.

Rule 16. Precondition: T has an x-supported leaf l such that $d_T(x^{\to l}) > 2$. **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$ (Fig. 5.6(a)).

Rule 17. Precondition: T has a leaf l, and a non-tree edge (l, x) such that $d_T(x^{\to l}) = 2$, and $c(x^{\to l}) < c(l)$. **Action:** Let $E(T') = E(T) + (l, x) - (x, x^{\to l})$ (Fig. 5.6(b)).

Rule 18. Precondition: T has an x-supported leaf l such that $c(b^-(l)) < c(l)$ **Action:** Let $E(T') = E(T) + (l, x) - (b(l), b^-(l))$ (Fig. 5.6(c)).

Rule 19. Precondition: T has a short leaf l_1 and a leaf l_2 such that $(l_1, l_2) \in E(G)$. **Action:** Let $E(T') = E(T) + (l_1, l_2) - (b(l_1), b^-(l_1))$ (Fig. 5.6(d)).

We can construct an approximation algorithm for the Maximum Weighted Internal Spanning Tree problem using the above rules. First we have a definition.

Definition 5.6.1 *A spanning tree T is a* weighted locally optimal spanning tree (WLOST) *if none of Rules 16–19 can be executed on it.*

Algorithm WLOST (Weighted Locally Optimal Spanning Tree)

Input: A simple connected vertex weighted graph G
Output: A weighted locally optimal spanning tree (WLOST) T of G
begin
Run Algorithm Greedy Traversal to obtain an initial spanning tree T
Execute Rules 16–19 as long as possible. If several rules can be executed, choose the one with the lowest number
end

Theorem 5.6.2 *Algorithm WLOST is an $\mathcal{O}(|V|^4)$-time $(2\Delta-3)$-approximation for the* Maximum Weighted Internal Spanning Tree *problem for graphs with no pendant vertices.*

The proof of the running time is postponed to Subsection 5.6.3. The following lemma establishes the approximation ratio.

Lemma 5.6.3 *Let T be a WLOST of a vertex-weighted graph $G = (V, E, c)$ that has no pendant vertices. Then $(2\Delta - 3)c(I(T)) \geq c(V)$.*

Proof: First, observe some basic properties of T that are immediate consequences of the definition of a WLOST.

Property 9. If l is a leaf of T and (l, x) is a non-tree edge then $d_T(x^{\rightarrow l}) = 2$ and $c(x^{\rightarrow l}) \geq c(l)$. (As Rules 16 and 17 are no more applicable in T.)

Property 10. If l is a supported leaf of T then $c(b^-(l)) \geq c(l)$. (As Rule 18 is no more applicable in T.)

Property 11. If l is a short leaf of T then l is G-independent from the set $L(T)$. (As Rule 19 is no more applicable in T.)

We define a mapping $f : L(T) \longrightarrow I(T)$ as follows. For a leaf l, let x be the vertex such that (l, x) is a non-tree edge and the length of path $P_T(l, x)$ is the maximum possible. Such an x always exists as $d_G(l) \geq 2$. If $br(l)$ is a short branch or $x \in br(l)$ then let $f(l) = x^{\rightarrow l}$. Otherwise, let $f(l) = b^-(l)$. This means that each long leaf l is mapped to an internal vertex of its own branch $br(l)$. Thus the images of long leaves are disjoint. By the above properties, it is easy to see that for every leaf l, we have $c(f(l)) \geq c(l)$, and $d_T(f(l)) = 2$.

The mapping f can be used to establish the approximation ratio. Unfortunately, several short leaves can be mapped to the same vertex y. The following claim is used to upper bound the number of such leaves.

Claim 5.6.4 *For any vertex $y \in V$, we have $|\{l : y = f(l)\}| \leq 2(\Delta - 2)$.*

Proof: Let $l_1, l_2, \ldots, l_r$ be leaves of T with $f(l_i) = y$ $(1 \leq i \leq r)$. As shown above, $d_T(y) = 2$. Let y_1 and y_2 be the T-neighbors of y. Recall that a long leaf is mapped to a vertex of its own branch. This implies that neither y_1 nor y_2 is a leaf of T. Suppose the contrary, namely e.g. $d_T(y_1) = 1$. Then y must be in the long branch

of y_1 and at least $r-1$ of the branches $br(l_i)$ $(1 \leq i \leq r)$ must be short. Moreover, by the definition of the mapping f, the graph G must have edges (l_i, y_1) which is a contradiction to Property 11. Hence, at least 2 edges incident to y_1 (and y_2) are tree edges, and so at most $\Delta - 2$ are non-tree edges. This shows that $r \leq 2(\Delta - 2)$, yielding the claim. □

Using Claim 5.6.4, and the fact that the images of long leaves are G-independent, we obtain

$$\sum_{v \in L(T)} c(v) = \sum_{l \in L_g(T)} c(l) + \sum_{l \in L_s(T)} c(l)$$
$$\leq \sum_{l \in L_g(T)} c(f(l)) + \sum_{l \in L_s(T)} c(f(l)) \leq 2(\Delta - 2) \sum_{v \in I(T)} c(v). \quad (5.16)$$

Hence adding $\sum_{v \in I(T)} c(v)$ to both sides we obtain

$$\sum_{v \in V} c(v) \leq (2\Delta - 3) \sum_{v \in I(T)} c(v)$$

which proves the approximation ratio. □

5.6.2 Claw-free graphs

In this subsection we extend Algorithm WLOST by an additional improvement step in order to get a 2-approximation algorithm for the Maximum Weighted Internal Spanning Tree problem for claw-free graphs, that is, graphs without induced $K_{1,3}$. Observe that (5.16) would guarantee an approximation ratio of 2 if we could find a WLOST without short branches. The new improvement rule does exactly this job, namely it converts short branches to long ones. Throughout this subsection, G is supposed to be claw-free. First we point out a property of WLOST's of claw-free graphs.

Let T be a WLOST of G, and l be a short leaf of T. Furthermore, let the T-neighbors of $b(l)$ be $l, x_1, x_2, \ldots, x_k$. Then by Property 9, none of $x_1, x_2, \ldots, x_k$ is a G-neighbor of l. Thus, vertices $x_1, x_2, \ldots, x_k$ must span a complete subgraph of G, since otherwise $b(l), l, x_i, x_j$ would induce a $K_{1,3}$ for some i, j. As a result, by Property 9, all the vertices x_i are internal vertices of T, for $i \geq 1$. Thus we have

Property 12. If l is a short leaf of T and the T-neighbors of $b(l)$ are $l, x_1, \ldots, x_k$, then $x_1, x_2, \ldots, x_k$ are all internal vertices of T and they induce a complete subgraph of G.

Using this property, we can now give an additional improvement rule to decrease the number of short branches while not changing the set of leaves.

Rule 20. Precondition: T has a short leaf l, and the T-neighbors of $b(l)$ are $l, x_1, \ldots, x_k$ such that for some $1 \leq i \leq k$ the vertex x_i is a branching, or x_i has a T-neighbor $v_i \neq b(l)$ which is an internal vertex. **Action:** Let $E(T') = E(T) \setminus \{(b(l), x_j)\}_{j=1..k, j \neq i} \cup \{(x_i, x_j)\}_{j=1..k, j \neq i}$.

Definition 5.6.5 *A WLOST is called a* Refined WLOST (RWLOST) *if Rule 20 cannot be executed on it.*

Using Rule 20, we can improve Algorithm WLOST to obtain a better approximation ratio for claw-free graphs.

Algorithm RWLOST (Refined Weighted Locally Optimal Spanning Tree)
Input: A simple connected vertex-weighted graph G
Output: A weighted locally optimal spanning tree (RWLOST) T of G
begin
  Run Algorithm Greedy Traversal to obtain an initial spanning tree T
  Execute Rules 16–19 and Rule 20 as long as possible. If several rules can be executed, choose the one with the lowest number
end

Theorem 5.6.6 *Algorithm RWLOST is an $\mathcal{O}(|V|^4)$-time 2-approximation for the* Maximum Weighted Internal Spanning Tree *problem for claw-free graphs that have no pendant vertices.*

Proof: Let T be an RWLOST of a claw-free graph $G = (V, E, c)$ that has no pendant vertices. We prove that in this case $c(I(T)) \geq \frac{1}{2}c(V)$. This yields the approximation ratio of 2. At first, we show that T has no short branches. Suppose the contrary, namely let l be a short leaf. Let $l, x_1, \ldots, x_k$ be the T-neighbors of $b(l)$. As T is an RWLOST, Rule 20 cannot be executed in it. Hence, each vertex x_i is a forwarding vertex with T-neighbors $b(l)$ and v_i, where v_i is a leaf of T (for $1 \leq i \leq k$). As a result, G has only $2k + 2$ vertices (l, $b(l)$, x_i's, and v_i's). Then Property 9 and Property 11 imply that l is not a G-neighbor of x_i's or v_i's, respectively. This gives $d_G(l) = 1$, a contradiction. Therefore all branches of T are long. Now recall (5.16) and reformulate it for a WLOST without short branches.

$$\sum_{v \in L(T)} c(v) = \sum_{l \in L_g(T)} c(l) \leq \sum_{l \in L_g(T)} c(f(l)) \leq \sum_{v \in I(T)} c(v).$$

Again, adding $\sum_{v \in I(T)} c(v)$ to both sides we conclude that

$$\sum_{v \in V} c(v) \leq 2 \sum_{v \in I(T)} c(v).$$

□

5.6.3 Running time analysis of Algorithms WLOST and RWLOST

The data structures used to execute Algorithm WLOST and Algorithm RWLOST are the same as the ones used for Algorithm LOST. Using them, the precondition part of each individual rule can be tested as follows.

Rule 16: This rule is the same as Rule 3 of Algorithm LOST. Checking its precondition needs $\mathcal{O}(|V|^2)$ time.

Rule 17: For every leaf l and for every non-tree edge (l, x) we check whether $c(x^{\to l}) < c(l)$. This requires $\mathcal{O}(|V|^2)$ time.

Rule 18: For every leaf l we check whether $c(b^-(l)) < c(l)$ and we look for a non-tree edge (l, x) such that $x \notin br(l)$. This needs $\mathcal{O}(|V|^2)$ time.

Rule 19: For every short leaf l_1 and for every leaf l_2 we check whether $(l_1, l_2) \in E(G)$. This requires $\mathcal{O}(|V|^2)$ time.

Rule 20 (only for Algorithm RWLOST): For every short leaf l it is enough to check the local neighborhood of $b(l)$. This can be done in $\mathcal{O}(|V|^2)$ time.

As a result, the precondition part of each particular rule can be checked in $\mathcal{O}(|V|^2)$ time. It remains to show that there are at most $\mathcal{O}(|V|^2)$ rule execution steps before the algorithms stop.

For this purpose, let the vertices of G be sorted in descending order of their weights, that is, $c(v_1) \geq c(v_2) \geq \cdots \geq c(v_n) \geq 0$. Let T be the initial spanning tree that we have before rule executions. Let us use $|L(T)|$ pieces of markers to select the leaves of T from $v_1, v_2, \ldots, v_n$. When an improvement rule is executed in the current spanning tree, we change the position of the appropriate markers such that they always point to the leaves. The execution of Rules 16 and 19 turns a leaf l into an internal vertex, that is, the marker of l is completely removed and is not used anymore. The execution of Rules 17 and 18 changes a leaf l_1 to another leaf l_2 such that $c(l_2) < c(l_1)$. This results that the marker of l_1 is moved to a higher rank element (l_2) of the sequence $v_1, v_2, \ldots, v_n$. Observe that such a way, every marker is moved at most n times. Thus, at most $\mathcal{O}(|V|^2)$ rule execution steps are enough to obtain a WLOST.

In Algorithm RWLOST, we alternate the execution of Rule 20 and Rules 16–19. We use the above method to mark the leaves of the current spanning tree. Rule 20 does not change the set of leaves, and so the position of markers. However, it decreases the number of short branches. It is easy to see that Rules 16 and 19 do not increase the number of short branches, while Rules 17 and 18 increase it by at most one. Putting these facts together we conclude that Rule 16 is executed at most $|L_s(T)| + r$ times, where r is the total number of executions of Rules 17 and 18, that is, $r = \mathcal{O}(|V|^2)$. This proves that after $\mathcal{O}(|V|^2)$ improvement steps we obtain an RWLOST.

As a result, we conclude that both algorithms use $\mathcal{O}(|V|^2)$ time for a single rule execution, and thus run in $\mathcal{O}(|V|^4)$ time.

5.7 Spanning Many Vertices with a q-Leaf Tree

Up to now we were focusing on spanning trees with few leaves. In this section we do the contrary, we fix the number of leaves to q and examine how many vertices can be spanned by an $\leq q$-leaf subtree. This approach is a generalization of finding a longest path in a graph, as a path is just a 2-leaf subtree. Let $\sigma_q(G)$ denote the minimum degree-sum of a q-element independent subset of $V(G)$, that is,

$$\sigma_q(G) = \min_{X \subseteq V(G)} \left\{ \sum_{v \in X} d(v) : |X| = q, X \text{ is } G\text{-independent} \right\}.$$

Let us recall Ore's theorem:

Theorem 5.7.1 [56] *Let G be a graph on n vertices. If $\sigma_2(G) \geq n$ then G has a Hamiltonian path.*

Notice that the original version of this theorem states the existence of a Hamiltonian cycle supposing the same sufficient condition. However, in this thesis, we only need the path version of which Bermond proved the following generalization for 2-connected graphs:

Theorem 5.7.2 [15] *Let G be a 2-connected graph on n vertices. Then G has a path of length* $\min\{n, \sigma_2(G)\}$.

Broersma and Tuinstra used a different point of view to generalize Ore's result. They examined the existence of q-leaf spanning trees and obtained the following:

Theorem 5.7.3 [19] *Let G be a graph on n vertices. If $\sigma_2(G) \geq n - q + 1$, for some integer $2 \leq q \leq n - 1$, then G has a q-leaf spanning tree.*

In what follows we give a common generalization of the above results. We give a sufficient condition on the existence of a q-leaf subtree that spans *many* vertices.

To formulate our statement on subtrees we use the following notation. Let S_q be a q-element independent set and let $k \leq q$ be an integer. Furthermore, let $X_k(S_q)$ be the set of the k highest degree vertices of S_q. Then we denote by $\rho_{q,k}(G)$ the minimum $\min_{S_q} \sum_{x \in X_k(S_q)} d(x)$, where the minimum is taken over all q-element independent sets. Clearly $\sigma_q(G) = \rho_{q,q}(G)$ and $\frac{\sigma_q(G)}{q} \leq \frac{\rho_{q,k}(G)}{k}$ holds for $k \leq q$.

The main result of this section is the algorithmic proof of the following theorem:

Theorem 5.7.4 *Let $G = (V, E)$ be a connected graph and let $2 \leq q < \alpha(G)$ be an integer. Then G has a subtree with at most q leaves that spans at least* $\min\{\rho_{q,2}(G) + q - 1, n\}$ *vertices of G.*

PROOF: Let T be a maximum cardinality subtree of G with at most q leaves. If T spans G then we are done. Otherwise let $R = V(G) \setminus V(T) \neq \emptyset$. As T is maximal, none of its leaves is adjacent to R and thus for each leaf l we have $d_{G[V(T)]}(l) = d_G(l)$. Moreover, T must be a minimum leaf spanning tree of $G[V(T)]$. Suppose that this is not the case and there exists a tree T' spanning $V(T)$ with less than q leaves. Let e be an edge between $V(T)$ and R. Then $T' + e$ is a tree with at most q leaves that spans more vertices than T. Observe that $G[V(T)]$ has no Hamiltonian cycle. Otherwise we could form a Hamiltonian path of $G[V(T)]$ with one of its leaves adjacent to R.

Therefore, T must have all properties of a minimum leaf spanning tree, and it has exactly q leaves. The set L of leaves of T is G-independent by Lemma 4.2.1 if T is not a Hamiltonian path of $G[V(T)]$, and by the fact that there is no Hamiltonian cycle in $G[V(T)]$ otherwise. Let l_1 and l_2 be the first and the second highest G-degree vertex in L, respectively. For the leaf l_2 we define the set $S = \left\{v : \exists u \in V(T) \text{ s.t. } (l_2, u) \in E(G) \setminus E(T), v = u^{\to l_2}\right\}$. If T is not a Hamiltonian path, and so $G[V(T)]$ is non-traceable, then by Lemma 4.2.2 we obtain that the vertices of S are forwarding vertices of T and so by the definition of S we have $|S| = d_G(l_2) - 1$. Moreover, in this case Lemma 4.2.2 also shows that the vertices of S are not adjacent to l_1, that is, S and $N_G(l_1)$ are disjoint. If T is a Hamiltonian path then the vertices of S are trivially forwarding vertices and disjoint from $N_G(l_1)$, as otherwise $V[G(T)]$ has a Hamiltonian cycle.

Thus

$$|V(T)| \geq |V_1(T)| + |S| + |N_G(l_1)| = q + d_G(l_2) - 1 + d_G(l_1) \geq \rho_{q,2}(G) + q - 1,$$

using the fact that the leaves of T form a q-size independent set. □

The above bound is strict as shown by the complete bipartite graph $G = (A \cup B, E) = K_{\sigma,n-\sigma}$ (for any $\sigma < n/2$). To see this, let T be any non-spanning subtree of G having exactly q leaves and $t = |V(T)|$ vertices.

If the leaves (being independent in G) are all in B then $|E(T)| = t - 1 = e_T(A, B) = e_T(A, B \cap V_{\geq 2}(T)) + q$, and each internal vertex of B has at least 2 neighbors in A, so we have $2|B \cap V_{\geq 2}(T)| \leq e_T(A, B \cap V_{\geq 2}(T)) = t - q - 1$. This, combined with $t \leq q + |B \cap V_{\geq 2}(T)| + \sigma$ results $t \leq 2\sigma + q - 1 = 2\frac{\sigma_q(G)}{q} + q - 1$. If the leaves are all in A then we take $G' = G - V_1(T)$ and a subtree $T' = T - V_1(T)$. It is easy to see that T' has all of its $q' \leq q$ leaves in B and following to the above argument we have $|V(T')| \leq 2(\sigma - q) + q' - 1 \leq 2\sigma - q - 1$ and so $t = |V(T')| + q \leq 2\sigma - 1$.

As a result, at most $2\frac{\sigma_q(G)}{q} + q - 1 \leq \rho_{q,2}(G) + q - 1$ vertices of G can be spanned by a subtree of at most q leaves.

Putting together Corollary 4.2.3 and Theorem 5.7.4 yields the following:

Corollary 5.7.5 *Let $G = (V, E)$ be a connected graph and $q \geq 2$ be an integer. If $q \geq \alpha(G)$ or $\rho_{q,2}(G) \geq n - q + 1$ then G has a spanning tree with at most q leaves.*

This is a generalization of the result of Broersma and Tuinstra [19] as $\rho_{q,2}(G)$, if defined, is an upper bound on $\sigma_2(G)$.

In the rest of this section we mention some further results related to the topic of density conditions and q-leaf spanning trees.

Win considered the case of k-connected graphs and proved the following:

Theorem 5.7.6 [71] *Let $q \geq 2$ and G be a k-connected graph. If $\alpha(G) \leq q+k-1$ then G has a spanning tree with at most q leaves.*

Tsugaki and Yamashita proved a common generalization of Theorems 5.7.3 and 5.7.6:

Theorem 5.7.7 [66] *Let $q \geq 2$ and G be a k-connected graph on n vertices. If $q+k \geq \alpha(G)$ or $\rho_{q+k,2} \geq n-q+1$ then G has a spanning tree with at most q leaves.*

Observe that this theorem, published independently of our work, also generalizes Corollary 5.7.5 for k connected graphs.

Instead of dealing with the degree sum of independent vertices, one can consider their neighborhood unions. If we have a vertex set $S \subset V(G)$, its neighborhood union is $N(S) = \bigcup_{v \in S} N(v)$, that is, the set of vertices having a neighbor in S. Let us denote by N_q the minimum of $|N(S)|$ among all q-element independent sets of $V(G)$. Then $N_2 \geq \frac{2}{3}(n-2)$ is a sufficient condition for traceability [14, 24, 27]. Based on this result, Flandrin et al. proved a neighborhood union based sufficient condition for the existence of a $(\leq q)$-leaf spanning tree:

Theorem 5.7.8 [27] *Let $q \geq 2$ and G be a connected graph on n vertices. If $N_q > \frac{q}{q+1}(n-q)$ then G has a spanning tree with at most q leaves.*

We finish this section by a theorem which shows how to construct maximum size q-leaf subtrees from an approximate solution T of the Maximum Internal Spanning Tree problem ($q = 2, 3, \ldots, |V_1(T)|$). Indeed, it is enough to find a maximum length path (2-leaf subtree) of T and then subsequently add $q-2$ longest paths to it. Notice that this construction provides no approximation ratio guarantee for the number of vertices spanned by a q-leaf subtree. It only shows how an algorithm designed for the Maximum Internal Spanning Tree problem can be used as a heuristic for finding a maximum size q-leaf subtree.

Theorem 5.7.9 *Let T be a tree having more than q leaves. Let T_q be a maximum size q-leaf subtree of T. Then there exists a $(q+1)$-leaf subtree T_{q+1} of T such that T_q is a subtree of T_{q+1}, and T_{q+1} has a maximum number of vertices among all $(q+1)$-leaf subtrees of T. Moreover, T_{q+1} can be obtained by adding to T_q a longest path of $E(T) - E(T_q)$ that has one of its ends in T_q.*

Proof: Observe that, for $2 \leq i \leq |V_1(T)|$, each leaf of a maximum cardinality i-leaf tree is a leaf of T, and that such a subtree is fully determined by enumerating its leaves.

Let us given T_q, a maximum size q-leaf subtree of T, and let $P(u,v)$ be a longest path of $E(T) - E(T_q)$ so that $u \in V(T_q)$. Then P and T_q has a single vertex in common, namely, $V(P) \cap V(T_q) = \{u\}$. We construct T_{q+1} by adding path P to T_q, that is, $T_{q+1} = T_q \cup P$. Clearly, due to the maximality of T_q, the vertex u cannot be its leaf, and so T_{q+1} has $q+1$ leaves: $V_1(T_{q+1}) = V_1(T_q) + v$. It remains to prove that T_{q+1} has maximum cardinality among all subtrees of T having $q+1$ leaves.

Suppose for a contradiction that there exists a $(q+1)$-leaf subtree $\hat{T}_{q+1}$ of T which has more vertices than T_{q+1}. The maximality of T_q implies that $T_{q+1} = T_q \cup P_1$, where P_1 is the shortest branch of T_{q+1}. Similarly, let $\hat{P}_1$ be the shortest branch of $\hat{T}_{q+1}$ and let $\hat{T}_q$ be the q-leaf subtree with edge set $E(\hat{T}_{q+1}) \setminus E(P)$. As $|V(T_q)| \geq |V(\hat{T}_q)|$, according to our assumption $|V(P_1)| < |V(\hat{P}_1)|$, that is, the shortest branch of T_{q+1} is shorter than that of $\hat{T}_{q+1}$. Now we consider two cases.

Case 1. The trees T_q and $\hat{T}_q$ are disjoint. In this case, let Q be the path of T connecting T_q and $\hat{T}_q$, and let l be a leaf of $\hat{T}_q$ such that Q contains no forwarding vertices of the branch of l in $\hat{T}_{q+1}$. Note that $V(br_{T_{q+1}}(l)) \subseteq V(br_{T_q}(l))$. Let R be the path connecting l to T_q. Now observe that R contains $br(l)$, a branch of $\hat{T}_q$, and so $|V(R)| \geq |V(\hat{P}_1)|$. Thus we have

$$|V(T_{q+1})| \geq |V(T_q)| + |V(R)| - 1 > |V(\hat{T}_q)| + |V(\hat{P}_1)| - 1 = |V(\hat{T}_{q+1})|,$$

contradicting to our assumption.

Case 2. The trees T_q and $\hat{T}_q$ intersect. In this case the maximality of T_q implies that for each leaf l of $\hat{T}_q$ and its branch, the intersection $V(T_q) \cap V(br_{\hat{T}_q}(l))$ is either the whole branch $V(br_{\hat{T}_q}(l))$, or only the branching $b_{\hat{T}_q}(l)$, or the empty set. Now, let R be the longest branch of $\hat{T}_q \setminus T_q$. Then clearly, by the minimality of $\hat{P}_1$, we have $|R| \geq |\hat{P}_1|$ and so

$$|V(T_{q+1})| \geq |V(T_q)| + |V(R)| - 1 \geq |V(\hat{T}_q)| + |V(\hat{P}_1)| - 1 = |V(\hat{T}_{q+1})|,$$

forming a contradiction to our assumption. □

5.8 Dense Claw-Free Graphs

Now recall Theorem 5.7.3 which states that if the density condition $\sigma_2(G) \geq n-q+1$ holds then G has a q-leaf spanning tree. In what follows, we give an algorithmic proof to a stronger version of this theorem for claw-free graphs.

Theorem 5.8.1 *Let G be a connected claw-free graph on n vertices. For any integer $2 \leq q$, if $\sigma_{q+1}(G) \geq n - q$ then G has a spanning tree with at most q leaves.*

The proof of this theorem is analogous to that of Gargano et al. [32] giving a similar condition to the existence of a k-branching spanning tree:

Theorem 5.8.2 [32] *Given a connected claw-free graph G. For any integer k, if $\sigma_{k+3}(G) \geq n - k - 2$ then G has a spanning tree with at most k branchings.*

Proof of Theorem 5.8.1: Let T be a subtree of G with at most q leaves such that

(i) T spans a maximum number of vertices;

(ii) with respect to (i), T minimizes $\sum_{l \in L(T)} |br(l)|$, that is, the total length of branches;

(iii) with respect to (i) and (ii), T maximizes $\sum_{l \in L(T)} |br(l)|^2$, that is, the square sum of branch lengths.

Clearly, (i) implies that T is either a spanning tree or a subtree with exactly q leaves. If T is a spanning tree then we are done. Thus, from now on we suppose that $|V(T)| < |V(G)|$ and $|L(T)| = q$. We show that this contradicts to the condition $\sigma_{q+1}(G) \geq n - q$, proving that T is a spanning tree.

Let $R = V(G) \setminus V(T)$ be the set of vertices not spanned by T. Observe that T must be a minimum leaf spanning tree of $G[V(T)]$. Suppose the contrary and let T' be a minimum leaf spanning tree of $G[V(T)]$. By its definition, T' has less than q leaves, therefore we can add an edge $e \in e_G(R, V(T))$ to T' thus obtaining a subtree T'' of G with $|L(T'')| \leq q$. As T'' spans more vertices than T, we get a contradiction to (i).

Since T is a minimum leaf spanning tree of $G[V(T)]$, it has Properties 1–7 of LOST's, though T might not be a LOST by itself. We rephrase here some of these properties and give some extra ones that T also has.

Property 13. $L(T)$ forms a G-independent set. This is a trivial consequence of Property 1 of LOST's.

Property 14. Let l_1 and l_2 be leaves of T. Then l_2 and $b^-(l_1)$ are not G-neighbors. This is a trivial consequence of Property 4 of LOST's if l_1 is long and of Property 13 if l_1 is short.

Property 15. Let l_1 and l_2 be leaves of T. Then l_2 is G-independent from the set of l_1-leafish vertices. This is a trivial consequence of Property 5 of LOST's.

In addition to these, (i)–(iii) also imply the following properties of T:

Property 16. Let l be a leaf and (x, y) be a trunk-edge of T. Then x and y cannot be both G-neighbors of l.

Otherwise, if both (l, x) and (l, y) are non-tree edges then we have a spanning tree T' of $G[V(T)]$ with edge set $E(T') = E(T) - (x, y) - (l, z) + (l, x) + (l, y)$, where z is the only T-neighbor of l. If l is a short leaf of T then T' has less leaves which contradicts the fact that T is a minimum leaf spanning tree of $G[V(T)]$. If l is a long leaf of T then both T and T' have q leaves. However, as we cut l down from

its branch, the total branch length $\sum_{v\in L(T')} |br(v)|$ in T' is less than $\sum_{v\in L(T)} |br(v)|$ in T contradicting (ii).

Property 17. Let l_1 and l_2 be leaves of T such that $|br(l_1)| \geq |br(l_2)|$. Then l_1 is not a G-neighbor of any vertex in $br(l_2) - b(l_2)$.

Otherwise, if G has a non-tree edge (l_1, x) where $x \in br(l_2) - b(l_2)$ then we have a spanning tree T' of $G[V(T)]$ with edge set $E(T') = E(T) + (l_1, x) - (x, x^{\to b(l_2)})$. Then Property 13 implies $x \neq l_2$ and so l_2 is a long leaf. If $x^{\to b(l_2)} = b(l_2)$ then T' has less leaves than T had, contradicting Property 13. Thus we can suppose that $d_T(x^{\to b(l_2)}) = 2$. In this case, T and T' have the same number of leaves and the same total length of branches. However, the square sum $\sum_{l\in L(T')} |br(l)|^2$ in T' is strictly greater than $\sum_{l\in L(T)} |br(l)|^2$ in T. This contradicts to (iii).

Property 18. Let l_1 and l_2 be leaves of T such that $|br(l_1)| < |br(l_2)|$. Then there are no vertices $x, y \in br(l_2) - b(l_2)$ such that (x, y) is a tree-edge and both (l_1, x) and (l_1, y) are non-tree edges.

Otherwise, if G has such edges then we have a spanning tree T' of $G[V(T)]$ with edge set $E(T') = E(T) + (l_1, x) + (l_1, y) - (x, y) - (l_1, z)$, where z is the only T-neighbor of l_1. Then Property 13 implies $x \neq l_2$ and $y \neq l_2$. If l_1 is a short leaf then $|L(T')| < |L(T)|$, contradicting the fact that T is a minimum leaf spanning tree of $G[V(T)]$. Therefore l_1 must be a long leaf. The number of leaves and the total length of branches is the same in T and in T'. However, the square sum $\sum_{l\in L(T')} |br(l)|^2$ in T' is strictly greater than $\sum_{l\in L(T)} |br(l)|^2$ in T. This contradicts to (iii).

Property 19. G has no edge between R and $L(T)$.

Otherwise, if such an edge (r, l) exists ($r \in R$, $l \in L(T)$) then the tree T' obtained by adding edge (r, l) to T has the same number of leaves as T. Since T' spans more vertices than T, this is a contradiction.

Property 20. Let (x, y) be an edge of T. Then there is no vertex $r \in R$ being G-neighbor of both x and y.

Otherwise, the tree T' with $V(T') = V(T) + r$ and $E(T') = E(T) + (r, x) + (r, y) - (x, y)$ has the same number of leaves as T and it spans more vertices. This, again, forms a contradiction.

Property 21. Let l be a leaf of T and v be an l-leafish vertex of $br(l)$. Then there is no vertex $r \in R$ being G-neighbor of x.

Recall Rule A of Algorithm LOST. It can be used to turn v into a leaf and l to an internal vertex. Suppose for a contradiction that Property 21 is not satisfied, that is, $(r, v) \in E(G)$. If we apply Rule A on T and add the non-tree edge (r, v) to it then we obtain a tree T' which has $|L(T')| = |L(T)|$ leaves and which spans more vertices than T. This is a contradiction.

Using these properties, we can now give a few claims on T.

Claim 5.8.3 *For each vertex $r \in R$, the set $L(T) + r$ is G-independent.*

This is an immediate consequence of Properties 13 and 19.

Claim 5.8.4 *Let l_1 and l_2 be leaves of T. Then there is no vertex x such that both l_1 and l_2 are G-neighbors of x.*

Indeed, let us suppose that $(l_1, x) \in E(G)$ and $(l_2, x) \in E(G)$, and let r be an arbitrary vertex in R. First notice that Property 19 implies that there is no edge between l_1 and r or l_2 and r. We consider five cases.

1. If x is a leaf then the contradiction is immediate as shown by Property 13.

2. If x is a trunk vertex and there is a trunk-edge (x, y) incident to it then by Property 16 neither (l_1, y) nor (l_2, y) is a G-edge. As l_1 and l_2 are not adjacent, vertices l_1, l_2, x, y span a claw, which is a contradiction.

3. If x is a trunk vertex but there is no trunk edge incident to it then T is a spider which has a third leaf l_3. As x is the only branching of T we have $b(l_3) = x$. By Property 14, leaves l_1 or l_2 are not G-neighbors of $b^-(l_3)$. Therefore l_1, l_2, x and $b^-(l_3)$ form a claw, a contradiction.

4. If x is an internal vertex of $br(l_2)$ then, by Property 17, we have $|br(l_1)| < |br(l_2)|$. Moreover, by Property 14, $x \neq b^-(l_2)$. Let y abbreviate $x^{\to b(l_2)}$. By Property 18, y is not a G-neighbor of l_1 and, by Property 15, y is not a G-neighbor of l_2. Therefore, l_1, l_2, x and y form a claw which is a contradiction.

5. If x is an internal vertex of $br(l_3)$, where l_3 is a leaf different from l_1 and l_2, then, by Property 17, we have $|br(l_1)| < |br(l_3)|$ and $|br(l_2)| < |br(l_3)|$. Property 14 implies $x \neq b^-(l_3)$ and hence $y = x^{\to b(l_3)}$ is an internal vertex of $br(l_3)$. This yields that l_1, l_2, x, y form a claw as, by Property 18, neither l_1 nor l_2 is a G-neighbor of y. This is a contradiction.

Claim 5.8.5 *Let l be a leaf of T and $r \in R$ be a vertex of G not spanned by T. Then there is no vertex x of T such that x is a G-neighbor of both l and r.*

The proof of this claim is similar to that of Claim 5.8.4 and is done by contradiction. Let us suppose that $(l, x) \in E(G)$ and $(r, x) \in E(G)$. Again, we consider five cases.

1. If x is a leaf then the contradiction is immediate as shown by Property 13.

2. If x is a trunk vertex and there is a trunk-edge (x, y) incident to it then, by Properties 16 and 20, neither (l, y) nor (r, y) is a G-edge. Thus vertices l, r, x, y span a claw, which is a contradiction.

3. If x is a trunk vertex but there is no trunk edge incident to it then T is a spider. Let $l_2 \neq l$ be a leaf of T. As x is the only branching of T we have $b(l_2) = x$. By Property 14, the leaf l is not a G-neighbor of $b^-(l_2)$. Property 20 implies that r and $b^-(l_2)$ are not G-neighbors. Therefore l, r, x and $b^-(l_2)$ form a claw, a contradiction.

4. If x is an internal vertex of $br(l)$ then it is different from $b^-(l)$ by Property 14. Let y abbreviate $x^{\to b(l)}$. As $(x,r) \in E(G)$, by Property 21, x cannot be l-leafish. Hence, y and l are not G-neighbors. Property 20 shows that $(y,r) \notin E(G)$. Therefore, l, r, x and y form a claw which is a contradiction.

5. If x is an internal vertex of $br(l_2)$ where l_2 is a leaf different from l then let us denote $x^{\to b(l_2)}$ by y. Property 20 shows that $(y,r) \notin E(G)$. As l is a G-neighbor of x, Property 17 gives $|br(l)| < |br(l_2)|$. Then, by Property 18, l is not a G-neighbor of y. This yields that l, r, x, y form a claw. This is a contradiction.

Claim 5.8.6 *Let $r \in R$ be a vertex of G not spanned by T and let x be an arbitrary vertex of G. Then x has at most one G-neighbor in $L(T) + r$.*

If $x \in R$ then, by Property 19, x has no G-neighbor in $L(T)$. If $x \in V(T)$ then Claim 5.8.4 implies that x can have at most one neighbor in $L(T)$. In this case, Claim 5.8.5 shows that if $(r,x) \in E(G)$ then x has no G-neighbor in $L(T)$.

Now we can turn to finish the proof of Theorem 5.8.1 using the above claims.

Recall that we supposed $V(T) \subset V(G)$. This implies $|L(T)| = q$ and that there exists a vertex $r \in V(G) \setminus V(T)$. Let $X = V(G) \setminus (L(T) + r)$. By Claim 5.8.6, each vertex of X has at most one G-neighbor in $L(T) + r$. Therefore $e_G(X, L(T) + r) \leq |X| = n - (q+1)$. As $L(T) + r$ is a G-independent set, we have $\sum_{v \in L(T)+r} d_G(v) \leq n - q - 1$. Therefore, by its definition, $\sigma_{q+1}(G) \leq n - q - 1$. This concludes the proof. □

5.9 Experimental Analysis

In this section we present the results of our experimental analysis on the Maximum Internal Spanning Tree problem. Our aim is to compare the performance of the different traversal algorithms used for obtaining the initial spanning tree, and to measure the approximation. We consider four algorithms. Each starts with the construction of an initial spanning tree and then applies Rules 1–3 of Algorithm LOST as long as possible. The difference among the four algorithms is the way of finding the initial spanning tree:

- Algorithm 1 creates a random spanning tree;
- Algorithm 2 creates a DFS-tree;
- Algorithm 3 creates a FIFO-DFS-tree;
- Algorithm 4 creates an RDFS-tree.

To create the set of input graphs, we use two different graph generation methods. Both of these methods start with the creation of a simple path $P = v_1 v_2 \dots v_n$. As a result, we can test our algorithms on traceable graphs where the value of the optimum solution is known to be $n - 2$. Next, both methods randomly add some other edges to the graph. The two methods differ in the way how these random edges are selected.

Method 1 considers one by one all pairs of vertices (v_i, v_j) for $i < j - 1$. For each such pair, the edge (v_i, v_j) is added to the input graph with a probability of p. Observe that in this case the creation of a new edge is independent of the current structure of the graph.

Method 2, in contrast, uses an approach which prefers higher degree vertices to lower degree ones when it has to choose the end vertices of a new edge. This method considers vertices v_i one by one for $i = 3, 4, \dots, n$. For vertex v_i, it generates a random number r according to the $(i-2, p)$-parameter Binomial distribution. Then it chooses r vertices among $v_1, v_2, \dots, v_{i-2}$ and connects each of them to v_i by a new edge. When choosing these r vertices, each of $v_1, v_2, \dots, v_{i-2}$ is considered with a probability proportional to its current degree.

Notice that both methods result to the same expected number of edges. Namely, on one hand, Method 1 yields an expected number of edges of

$$\mathbf{E}(|E(G)|) = n - 1 + p\left[\binom{n}{2} - (n-1)\right] = n - 1 + p\binom{n-1}{2}.$$

On the other hand, Method 2 provides an expected number of edges of

$$\mathbf{E}(|E(G)|) = n - 1 + \sum_{i=3}^{n} p(i-2) = n - 1 + p\sum_{i=1}^{n-2} i = n - 1 + p\binom{n-1}{2}.$$

Using both methods, we build graphs of 100, 300, and 500 vertices having different edge densities. We create 10 random graphs for each $(\mathcal{M}, n, p)$ triplet, where $\mathcal{M}$ is the method used for generating the input graph. Then we run every algorithm 10 times on each of these input graphs. As a result each $(\mathcal{M}, n, p)$ triplet is represented by 100 runs of each algorithm. Tables A.1, A.2, and A.3 in Appendix A show the experimental results for graphs of $n = 100, 300, 500$ vertices, respectively. Let $\mathcal{A}$ denote the algorithm used to solve the Maximum Internal Spanning Tree problem. Then for each tuple $(\mathcal{M}, n, p, \mathcal{A})$ we have two numbers in the tables. The one in the column labelled by $|L|_1$ corresponds to the phase when the algorithm has just found the initial spanning tree but has not applied any local improvement rule yet. The number in the column labelled by $|L|_2$, in contrast, corresponds to the final phase, namely when local improvement Rules 1–3 of Algorithm LOST have already been applied on the initial spanning tree. Both of these numbers show the

average number of leaves based on the 100 runs corresponding to the given tuple $(\mathcal{M}, n, p, \mathcal{A})$.

In what follows we make some observations based on Tables A.1–A.3.

1. There is a strong connection between the average degree of the input graph and the average number of leaves of the output spanning tree. Our algorithms performs the best on dense graphs. This is somewhat we expect, as in these graphs the traversals themselves need to step back much more rarely and the local improvement rules can be executed for more subgraphs. On the other hand, if the average degree of the input graph is very close to 2 then we find a good enough solution with high probability since the number of edges and so the number of spanning trees is low. The most interesting part is when the average degree is between these two extremities. Observe that even in this range, there is no significant difference in the results obtained using the two different input graph generation methods.

2. The algorithm using Algorithm RDFS for finding the initial spanning tree overperforms the others for every input graph. Moreover, not considering the small and sparse graphs, this is true even if we do not execute any local improvement rule on the initial RDFS-tree. This surprising fact shows that in many situations we can gain more by the appropriate choice of the initial traversal than by the application of the more and more sophisticated local improvements. Algorithm RDFS provides a good approximation factor for claw-free and cubic graphs. As our experiments show, it can be used effectively for general graphs, as well.

3. The ranking of the four algorithms based on their average performance is the same for almost all considered input graphs. If we compare the number of leaves after the local improvement steps (columns labelled by $|L|_2$) then we find the RDFS-tree approach being the best choice, followed by random tree, FIFO-DFS-tree, and DFS-tree approaches, respectively. The fact that the random-based approach yields better results than the DFS-tree and the FIFO-DFS-tree ones shows the power of the applied local improvement technique. It is important to see that the theoretical approximation factor of 7/4 for the Maximum Internal Spanning Tree problem is highly overperformed in terms of the average behavior of our algorithms for the input graphs considered in this section.

4. Algorithm RDFS has the weakest performance for input graphs with an average degree of about 3.5–4, while this measure is about 4–4.5 for the other 3 algorithms. Algorithm RDFS gives a solution being close to the optimum for most of the cases where this average degree is at least 10. From this point of view, it is much better than the other 3 algorithms: the random tree approach usually gives a close-to-optimum tree when the average degree of the input

graph is at least $n/5$, and the FIFO-DFS- and DFS-tree approaches give such a tree when the average degree is at least $n/3$.

Chapter 6

Conclusion and Future Work

In this thesis, we considered some degree-based spanning tree optimization problems. One common point of these problems is the special property of the measure function, namely, that it depends only on the degree distribution of the obtained spanning tree. More particularly, we dealt with problems which are generalizations of the Hamiltonian Path problem, immediately yielding that our optimization problems are all NP-hard.

The first of them, the Minimum Branching Spanning Tree problem was originally inspired by a design problem of DWDM networks: electronic cross connect devices have to be placed to nodes of a communication network such that all possible demands be satisfied with the lowest cost. For the Minimum Branching Spanning Tree problem, we gave both negative and positive approximability results. We proved that $\Omega(\log n)$ is the best possible approximation ratio (Theorem 3.2.3), and gave Algorithm MinBST which achieves this ratio whenever the input graph is evenly dense (Theorem 3.3.1). Further work can be done in order to find approximation algorithms for general graphs.

Then we investigated the Maximum Internal Spanning Tree problem. Firstly, we presented Algorithm ILST which yields an independence tree of the input graph. We proved that such a tree always provides a 2-approximation for general graphs (Theorem 5.2.2), a 4/3-approximation for cubic graphs and a 5/3-approximation for 4-regular graphs (Theorem 5.2.4). Our main result for the Maximum Internal Spanning Tree problem is a local improvement based algorithm which ensures an approximation factor of 7/4 for graphs without pendant vertices (Theorem 5.5.2). Moreover, we gave a traversal algorithm which has an approximation factor of 6/5 for cubic graphs (Theorem 5.4.4), and 3/2 for claw-free graphs (Theorem 5.4.2). For the weighted version, we showed a $(2\Delta - 3)$-approximation for general graphs (Theorem 5.6.2), and a 2-approximation for claw-free graphs (Theorem 5.6.6). We also considered the Maximum Internal Spanning Tree problem from another point of view: instead of looking for a spanning tree with few leaves, we searched for a q-leaf subtree covering as many vertices as possible (Theorem 5.7.4).

This approach is a generalization of the LONGEST PATH problem.

Further research can be done in several directions. Approximation ratios can be improved for general graphs or for special graph classes. Knauer and Spoerhase [48] have recently further developed our local improvement based algorithms. Based on our work, they provided a 5/3-approximation algorithm for the MAXIMUM INTERNAL SPANNING TREE problem and a $(3+\varepsilon)$-approximation algorithm for the MAXIMUM WEIGHTED INTERNAL SPANNING TREE problem, for any $\varepsilon > 0$.

Keeping in mind that our research is motivated by a network design application, we can consider the case when only a few dedicated network nodes must be connected. This can be modelled by the Steiner-tree version of our problems where only a predefined set $T \subset V(G)$ must be spanned by a tree which possibly uses some vertices from $V(G) - T$, as well. All degree-based spanning tree problems can be generalized this way.

If we want our networks to be protected against failures, we can create multiple paths between its nodes. In order to model this case, we can search for a k-(edge)-connected subgraph rather than a spanning tree (which is the $k = 1$ case). The same degree-based cost functions can be applied to these problems, too.

A more theoretic open question is whether there always exists an optimum solution which can arise as the output of a run of Algorithm FIFO-DFS. We know that the answer for the same question is yes for the more general Algorithm Greedy Traversal (Theorem 5.3.1), and no for Algorithm DFS. We also examined how the choice of the embedded graph traversal algorithm influence the performance of Algorithm LOST. We found that the local improvement rules are more effective when applied on a random spanning tree than in the cases where we choose Algorithm DFS or Algorithm FIFO-DFS to be the initial traversal. The best behavior was experienced using Algorithm RDFS.

Let us briefly summarize here the results on a set of degree based spanning tree optimization problems being the generalizations of the HAMILTONIAN PATH problem. We have already mentioned the MINIMUM BRANCHING SPANNING TREE problem, and the MAXIMUM INTERNAL SPANNING TREE problem. From an optimization point of view, the former one is equivalent to the so called MAXIMUM NON-BRANCHING SPANNING TREE problem (where the aim is to maximize the total number of leaves and forwarding vertices), and the latter one is to the MINIMUM LEAF SPANNING TREE problem. We have seen that the MINIMUM LEAF SPANNING TREE problem is NP-hard to approximate within a constant factor [53], however, its complement, the MAXIMUM INTERNAL SPANNING TREE problem has much better approximation properties. Concerning the MINIMUM BRANCHING SPANNING TREE problem, we found that it is very unlikely that an approximation algorithm exists with a ratio better than $\Omega(\log n)$. For the MAXIMUM NON-BRANCHING SPANNING TREE problem, however, we can find a trivial 2-approximation. Indeed, any spanning tree achieves this ratio, since, by Claim 2.1.3, the number of non-branchings is always larger than $n/2$. We have a third self-complement pair of problems to mention here. The MAXIMUM FORWARDING SPANNING TREE problem aims to

find a spanning tree with as many forwarding vertices as possible, while the Minimum Non-Forwarding Spanning Tree problem is about to minimize the total number of leaves and branchings. It is still an open question whether the Maximum Forwarding Spanning Tree problem can be constant-factor approximated. On the other hand, the following explanation shows that for its complement, the Minimum Non-Forwarding Spanning Tree problem, we very unlikely have such an algorithm.

Indeed, the following reduction from the NP-hard (u, v)-Hamiltonian Path problem shows that, for any fixed k, the existence of a k-approximation algorithm for the Minimum Non-Forwarding Spanning Tree problem implies P=NP. Let us have a graph G and two distinguished vertices $u, v \in V(G)$. We create k copies of G and connect them by adding edges (v_i, u_{i+1}) for $i \in 1, 2, \ldots, k-1$, that is, between the neighboring copies of u and v. We denote the obtained graph by H. Now, if G has a Hamiltonian path between end vertices u and v then H has a Hamiltonian path, too. Therefore, the value of the optimum solution for the Minimum Non-Forwarding Spanning Tree problem in H is 2. If G has no Hamiltonian path, then any spanning tree of H has at least one branching and one leaf in each G_i in addition to the above 2 leaves, and so the optimum solution has a value of at least $2k+2$. As a consequence, we can use our k-approximation algorithm on H to decide the traceability of G. This shows that there is no constant factor approximation for the Minimum Non-Forwarding Spanning Tree problem, unless P=NP.

Table 6.1 summarizes all of the positive and negative approximability results just mentioned.

We mention here another possible direction of further research. There is a connection between the Maximum Internal Spanning Tree problem and the problem of finding a minimum size 2-connected or 2-edge-connected subgraph [22, 41, 51, 69]. 2-connected and 2-edge-connected graphs can be characterized by two slightly different versions of an ear-decomposition method [70]. Indeed, these graphs can be built from a cycle by successively adding "ears", that is, edges (trivial ears) and simple paths (non-trivial ears) whose both ends are already existing vertices. The total number of ears, including the initial cycle, is always $|E(G)| - |V(G)| + 1$. If we choose the decomposition which has a minimum number of non-trivial ears, we get a minimum size 2-(edge)-connected subgraph. In addition, if we have an arbitrary ear-decomposition of a 2-(edge)-connected graph and we remove the last edge of each ear, we get a spanning tree whose number of leaves is exactly the number of non-trivial ears.

Beside the algorithmic aspects of degree-based optimization, we also dealt with the connection among vulnerability parameters, Hamiltonicity and spanning tree leaves. On one hand, we proved that the scattering number of a graph is always a lower bound on the number of spanning tree leaves (Theorem 4.1.5). On the other hand, we defined a new vulnerability measure, cut-asymmetry (Definition 4.3.1), and showed that it determines the size of the biggest independent subset of

Original measure function		**Complemented measure function**	
problem	**result**	**problem**	**result**
Minimum Leaf Spanning Tree	$\nexists$ constant approx. [53]	Maximum Internal Spanning Tree	2-approx. (Theorem 5.2.2) 7/4-approx. for graphs with no pendant vertices (Theorem 5.5.2)
Maximum Forwarding Spanning Tree	open	Minimum Non-Forwarding Spanning Tree	$\nexists$ constant approx.
Minimum Branching Spanning Tree	$\nexists$ constant approx. (Theorem 3.2.2)	Maximum Non-Branching Spanning Tree	2-approx. (Claim 2.1.3)

Table 6.1: Summary of results on degree based spanning tree approximation

spanning tree leaves (Theorem 4.3.10). The connection between these parameters directly implies that an independence tree is always a 2-approximation for both the Maximum Internal Spanning Tree problem and the Minimum Connected Dominating Set problem (Corollary 4.3.25). Moreover, using the notion of cut-asymmetry, we provided a new sufficient condition for traceability (Theorem 4.3.6). Further research can be focused on this new vulnerability parameter and its further connection to hamiltonicity.

Acknowledgement

I would like to express my gratitude to everyone who helped me, in one way or another, preparing this thesis. First of all, I thank my family for their long lasting patience, support and inspiration. I am particularly grateful to my supervisor, András Recski for teaching me many interesting topics in graph theory, for giving me the possibility to become a member of Department of Computer Science and Information Theory, for introducing me to the research topic of Steiner- and spanning trees, and last but not least, for supporting my research with his indispensable pieces of idea and advice. My special thanks to my co-author, Gábor Wiener for working with me on some topics discussed in this thesis, and for giving me his professional support. I wish he could play Carcassonne much better in the future :) . I thank Katalin Friedl, Jácint Szabó, and Ferenc Wettl for improving the quality of this thesis by reading it through and making several valuable comments on it. I am grateful to András Sebő for his help and indications in the field of linear programming and approximation algorithms, and also for welcoming me as a visitor of the Graph Theory and Combinatorial Optimization Group of IMAG, in the beautiful city of Grenoble, France. I also express my thanks to András Frank whose classes on linear programming, combinatorial optimization and graph theory gave a solid foundation for my research work. At last, let me mention that the research leading to the results discussed in this thesis was supported by Grant Nos. 042559, 044733, and 67651 of the Hungarian National Science Foundation (OTKA), and by the Grant No. 2003-5044438 of the European MCRTN Adonet Contract.

Appendix A

Tables of Experimental Results

Input graph:		**Hamiltonian path + random edges**								**Hamiltonian path + internet graph model**							
Initial spanning tree:		**Random**		**DFS**		**FIFO-DFS**		**RDFS**		**Random**		**DFS**		**FIFO-DFS**		**RDFS**	
p	$\mathbf{E}(d_{avg})$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$
0.0001	1.990	2.41	2	2.16	2	2.14	2	2.1	2	3.31	2.04	2.84	2.16	2.3	2	2.6	2
0.0005	2.029	5.39	2.14	3.72	2.34	2.87	2.12	2.87	2.05	5.74	2.13	3.73	2.34	3.01	2.11	2.98	2.19
0.0010	2.077	9.82	2.76	5.41	3.58	4.23	2.71	3.6	2.41	9.64	2.42	5.35	3.45	4.11	2.7	3.95	2.78
0.0030	2.271	20.1	5.32	9.16	6.45	7.89	4.9	4.43	3.38	18.78	5.02	8.59	6.02	6.95	4.71	4.93	4.13
0.0050	2.465	24.64	7.17	10.93	7.96	10.37	6.57	5.37	4.64	24.68	6.88	11.46	8.36	10.31	6.55	5.07	4.19
0.0080	2.756	29.4	8.59	12.7	9.57	12.29	8.3	5.72	5.14	28.21	8.51	12.76	9.45	12.51	8.21	5.68	5.07
0.0100	2.950	30.72	8.91	14.07	10.31	13.46	9.01	6.15	5.51	32.09	9.41	13.83	10.7	13.5	8.89	6.72	6.06
0.0300	4.891	40.92	9.56	16.76	12.67	15.73	10.18	3.24	3.05	39.47	9.38	16.18	12.14	15.6	10.59	3.84	3.53
0.0500	6.831	41.9	8.24	15.5	11.38	14.89	9.57	2.43	2.31	42.19	7.97	15.84	11.65	14.98	9.68	2.48	2.34
0.0800	9.742	42.16	5.83	12.47	8.87	12.18	7.82	2	2	43.24	6.31	13.64	9.29	12.59	7.94	2.3	2.23
0.1000	11.682	43.08	5.28	11.74	8.16	10.93	7.09	2.32	2.2	42.68	5.18	11.74	8.19	10.84	7.15	2.2	2.1
0.2000	21.384	42.74	2.78	7.55	4.76	6.81	4.37	2	2	42.97	2.74	7.31	4.91	6.78	4.49	2.1	2.04
0.2500	26.235	41.89	2.33	5.91	3.78	5.16	3.39	2.1	2.03	42.18	2.45	6.47	4.07	5.69	3.65	2.1	2.05
0.3000	31.086	41.94	2.23	5.22	3.33	4.47	2.8	2	2	42.08	2.11	5.48	3.24	4.52	2.88	2.02	2
0.3500	35.937	41.79	2.09	4.64	2.84	3.93	2.67	2.03	2	42.01	2.05	4.53	2.89	3.94	2.79	2	2
0.4000	40.788	40.97	2.03	4.02	2.53	3.45	2.38	2	2	41.18	2.01	3.92	2.5	3.36	2.29	2	2
0.4500	45.639	41.29	2.01	3.53	2.27	2.81	2.25	2	2	41.43	2.03	3.87	2.36	3.22	2.31	2.1	2
0.5000	50.490	41.48	2	3.19	2.17	2.5	2.1	2	2	40.83	2	3.12	2.17	2.8	2.14	2	2
0.5500	55.341	41.07	2	3.31	2.14	2.54	2.09	2.1	2.01	40.68	2.01	3.02	2.16	2.5	2.1	2	2
0.6000	60.192	41.64	2	2.8	2.05	2.52	2.05	2	2	40.47	2.01	2.79	2.05	2.4	2.02	2	2
0.7000	69.894	40.71	2	2.55	2.03	2.2	2	2	2	40.68	2	2.47	2	2.13	2	2	2
0.8000	79.596	41.14	2	2.24	2	2.09	2.01	2	2	40.86	2	2.29	2.01	2.05	2	2	2
0.9000	89.298	40.66	2	2.1	2	2.01	2	2	2	41.4	2	2.09	2	2.02	2	2	2
0.9500	94.149	41.49	2	2.04	2	2	2	2	2	41.06	2	2.04	2	2	2	2	2
0.9900	98.030	40.77	2	2.05	2	2	2	2	2	40.94	2	2.02	2	2	2	2	2

Table A.1: Experimental results, the $n = 100$ case

Input graph:		**Hamiltonian path + random edges**								**Hamiltonian path + internet graph model**							
Initial spanning tree:		**Random**		**DFS**		**FIFO-DFS**		**RDFS**		**Random**		**DFS**		**FIFO-DFS**		**RDFS**	
p	$\mathbf{E}(d_{avg})$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$
0.0001	2.023	8.54	2.24	4.92	3.2	3.78	2.61	3.4	2.42	10.39	2.47	5.53	3.14	4.12	2.51	3.94	2.64
0.0005	2.142	33.36	5.31	13.44	9.83	12.16	7.37	5.86	4.54	34.31	6.22	12.84	9.64	11.88	7.27	5.6	4.48
0.0010	2.290	56.3	11.75	21.55	16.95	21.13	12.84	8.49	7.31	54.25	10.91	20.44	16.1	19.37	12.27	8.1	7.05
0.0030	2.884	91.4	24.61	37.83	30.71	36.68	23.4	12.77	11.92	90.98	23.26	36.25	29.41	36.92	23.41	13.07	12.2
0.0050	3.478	104.27	27.06	43.88	35.51	43.61	28.28	12.55	11.89	105.34	28.22	43.73	35.58	44.11	28.83	12.98	12.38
0.0080	4.369	117.36	28.47	47.19	38.3	47.07	30.11	9.26	8.79	115.89	27.91	47.31	38.09	46.39	29.94	9.2	8.66
0.0100	4.963	118.63	26.37	46.89	37.33	46.56	29.96	6.48	6.2	117.84	25.9	46.6	36.87	44.32	28.9	6.64	6.27
0.0300	10.903	127.87	15.74	35.59	26.96	34.6	21.47	2.12	2.1	127.84	15.4	35.22	26.75	33.97	21.04	2.25	2.19
0.0500	16.843	129.25	9.51	26.87	19.88	26.04	16.38	2.1	2.08	128.97	9.42	26.25	19.6	25.06	15.89	2	2
0.0800	25.754	128.46	5.29	18.09	13.28	17.02	10.45	2	2	128	5.5	17.92	12.86	17.37	10.57	2.16	2.15
0.1000	31.694	127.99	4.32	15.07	10.38	14.34	8.97	2	2	127.95	4.34	15.44	10.62	14.6	9	2	2
0.2000	61.394	125.46	2.42	8.23	5.31	7.1	4.56	2	2	124.44	2.58	8.11	5.12	7.07	4.46	2.01	2.01
0.2500	76.245	124.05	2.19	6.22	3.87	5.31	3.44	2	2	124.48	2.19	6.38	4.11	5.47	3.49	2	2
0.3000	91.095	124.42	2.14	5.52	3.46	4.73	3.05	2.07	2.05	123.59	2.12	5.23	3.3	4.4	3.04	2	2
0.3500	105.945	123.47	2.02	4.65	2.89	3.86	2.58	2.09	2.05	123.09	2.04	4.63	2.83	3.85	2.63	2	2
0.4000	120.796	123.09	2.02	3.92	2.44	3.37	2.43	2	2	124.09	2	3.97	2.46	3.2	2.32	2	2
0.4500	135.646	121.84	2	3.34	2.22	2.86	2.27	2	2	122.75	2.02	3.47	2.34	2.78	2.17	2	2
0.5000	150.497	122.09	2	3.27	2.24	2.79	2.12	2	2	122.72	2	3.08	2.15	2.81	2.16	2	2
0.5500	165.347	123.16	2.01	3.12	2.18	2.51	2.07	2	2	122.54	2	2.93	2.12	2.39	2.04	2	2
0.6000	180.197	122.46	2	2.61	2.06	2.32	2.03	2	2	122.37	2	2.69	2.05	2.3	2.02	2	2
0.7000	209.898	123	2	2.45	2.02	2.22	2.02	2	2	122.22	2	2.37	2	2.14	2	2	2
0.8000	239.599	123.2	2	2.32	2	2.05	2	2	2	122.12	2	2.31	2	2.09	2	2	2
0.9000	269.299	124.14	2	2.15	2	2.01	2	2	2	124.63	2	2.12	2	2.01	2	2	2
0.9500	284.150	123.82	2	2.08	2	2	2	2	2	124.5	2	2.08	2	2	2	2	2
0.9900	296.030	123.64	2	2.01	2	2	2	2	2	124.16	2	2	2	2	2	2	2

Table A.2: Experimental results, the $n = 300$ case

Input graph:		**Hamiltonian path + random edges**								**Hamiltonian path + internet graph model**							
Initial spanning tree:		**Random**		**DFS**		**FIFO-DFS**		**RDFS**		**Random**		**DFS**		**FIFO-DFS**		**RDFS**	
p	$\mathbf{E}(d_{avg})$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$	$\|L\|_1$	$\|L\|_2$
0.0001	2.046	23.53	3.1	9.79	6.91	8.51	5.01	4.52	3.15	20.2	2.65	9.26	6.1	7.86	4.6	4.61	3.19
0.0005	2.245	76.59	13.28	26.3	20.75	25.72	15.53	8.76	7.45	80.03	14.05	27.03	21.76	27.2	16.78	9.47	8.26
0.0010	2.493	118.82	27.18	43.77	35.29	43.5	27	14.52	13.48	117.7	27.53	45.06	37.06	44.65	28.05	15.16	14.2
0.0030	3.487	171.94	43.94	70.07	57.81	71.3	45.98	19.75	18.99	174.65	44.83	71.47	58.9	71.57	46.89	19.82	19.11
0.0050	4.481	191.62	44.67	76.41	62.45	75.75	48.75	12.68	12.38	191.49	44.15	77.19	62.51	76.2	49.02	13.17	12.68
0.0080	5.972	203.17	39.51	75.13	60.42	74.43	47.54	6.35	6.05	203.33	39.44	74.23	59.58	73.73	47.28	5.7	5.53
0.0100	6.966	208.31	35.41	69.9	55.45	70.48	44.22	3.94	3.83	207.54	35.76	70.11	55.51	69.98	44.2	3.77	3.64
0.0300	16.906	215.64	15	43.25	32.85	42.21	26.07	2.1	2.08	214.85	14.83	42.65	32.17	41.99	26.09	2	2
0.0500	26.846	212.95	8.59	30.11	21.99	28.42	17.37	2.1	2.1	213.47	8.41	28.63	20.95	27.89	16.72	2	2
0.0800	41.756	212.52	5.19	19.1	13.39	18.14	11.29	2.09	2.07	211.63	5.32	19.05	13.68	18.8	11.3	2.1	2.09
0.1000	51.696	211.02	4.22	15.7	11.14	14.43	8.95	2	2	211.51	4.39	15.68	10.98	14.53	8.86	2.1	2.1
0.2000	101.397	207.71	2.43	7.49	4.95	6.59	4.19	2	2	207.73	2.41	7.25	4.74	6.79	4.41	2	2
0.2500	126.247	207.98	2.21	5.92	3.65	5.34	3.56	2	2	206.86	2.1	5.99	3.62	5.39	3.48	2	2
0.3000	151.097	207.07	2.09	5.02	3.14	4.37	2.99	2	2	205.88	2.1	5.22	3.18	4.28	2.81	2.1	2.03
0.3500	175.947	209.18	2.03	4.48	2.75	3.58	2.41	2	2	207.9	2.03	4.49	2.88	3.61	2.46	2	2
0.4000	200.798	207.71	2.01	3.95	2.6	2.92	2.22	2	2	207.58	2.02	3.76	2.53	3.2	2.32	2	2
0.4500	225.648	210.54	2.01	3.42	2.2	2.9	2.19	2	2	210.43	2	3.38	2.23	2.8	2.16	2	2
0.5000	250.498	212.43	2	3.24	2.11	2.57	2.14	2	2	212.17	2	3.23	2.22	2.63	2.08	2	2
0.5500	275.348	212.14	2	2.89	2.07	2.37	2.09	2	2	211.37	2	2.97	2.07	2.46	2.09	2	2
0.6000	300.198	220.4	2	2.7	2.02	2.29	2.01	2	2	219.62	2	2.7	2.03	2.3	2.01	2	2
0.7000	349.899	228.58	2	2.44	2.02	2.08	2	2	2	226.89	2	2.46	2	2.12	2	2	2
0.8000	399.599	224.65	2	2.19	2	2.05	2	2	2	225.94	2	2.2	2.02	2.05	2	2	2
0.9000	449.300	239.42	2	2.11	2	2.01	2	2	2	238.72	2	2.1	2	2.01	2	2	2
0.9500	474.150	217.36	2	2.07	2	2	2	2	2	217.89	2	2.1	2	2	2	2	2
0.9900	494.030	220.93	2	2.02	2	2	2	2	2	217.78	2	2	2	2	2	2	2

Table A.3: Experimental results, the $n = 500$ case

Appendix B

Summary of Notation

$\alpha(G)$	size of a maximum cardinality independent set of graph G, p. 13
$b_T(l)$ (or $b(l)$)	branching vertex being closest to leaf l in tree T, p. 13
$b_T^-(l)$ (or $b^-(l)$)	T-neighbor of $b(l)$ being in branch of l, an abbreviation for $b(l)^{\to l}$, p. 13
$br_T(l)$ (or $br(l)$)	branch of leaf l in tree T, p. 13
C_n	cycle on n vertices, p. 13
$\mathrm{ca}(G)$	cut-asymmetry of G, p. 37
$\mathrm{comp}_G(X)$ (or $\mathrm{comp}(X)$)	number of components of $G[X]$, p. 13
$d_G(v)$ (or $d(v)$)	degree of vertex v in graph G (number of G-neighbors of v), p. 12
$d_G(X)$	total degree of vertices of the vertex set X in graph G, p. 12
$d_T(v)$	number of T-neighbors of v, p. 12
$\delta_G(v)$ (or $\delta(v)$)	set of edges incident to vertex v in G, p. 12
$\Delta(G)$ (or Δ)	maximum vertex degree in graph G, p. 12
$E(G)$ (or E)	edge set of graph G, p. 12
$e_H(X)$	number of edges in $H[X]$, for some subgraph H, p. 13
$e_H(X,Y)$	number of H-edges between disjoint subsets X and Y of $V(H)$, for some subgraph H, p. 13
$e_H(v,X)$	number of H-edges between $v \in V(H) \setminus X$ and X, for some subgraph H, p. 13
$F_{G,T}(l)$ (or $F(l)$)	set of l-leafish vertices of spanning tree T of graph G, p. 48
G	a simple undirected connected graph
$G[X]$	subgraph of G induced by its vertex set X, p. 13
$I(T)$	set of internal vertices of tree T, p. 13
K_n	complete graph on n vertices, p. 13
K_{n_1,n_2}	complete bipartite graph with color classes of size n_1 and n_2, p. 13

$\mathrm{li}(G)$	leaf independence of G, p. 41
$L(T)$	set of leaves of tree T, p. 13
$L_{\mathrm{g}}(T)$	set of long leaves of tree T, p. 48
$L_{\mathrm{p}}(G,T)$ (or $L_{\mathrm{p}}(T)$)	set of long leaves of T having no leafish vertex in their branch, for some spanning tree T of G, p. 48
$L_{\mathrm{s}}(T)$	set of short leaves of tree T, p. 48
m	number of edges, p. 12
$\mathrm{ml}(G)$	number of leaves of a minimum leaf spanning tree of G, p. 8
n	number of vertices, p. 12
$N_G(v)$ (or $N(v)$)	set of G-neighbors of vertex v , p. 12
$N_T(v)$	set of T-neighbors of vertex v , p. 12
$P_T(u,v)$	unique path between vertices u and v of tree T, p. 13
$\mathrm{sc}(G)$	scattering number of G, p. 32
$u^{\rightarrow v}$	successor of vertex u along path $P_T(u,v)$, p. 13
$V(G)$ (or V)	vertex set of graph G, p. 12
$V_i(G)$	set of i-degree vertices in graph G, p. 13
$V_{\geq i}(G)$	set of (at least i)-degree vertices in graph G, p. 13
$\lvert X\rvert$	number of elements in set X
$X-x$	abbreviation for $X\setminus\{x\}$
$X+x$	abbreviation for $X\cup\{x\}$

List of Algorithms

Cited Publications of the Author

[1] T. Cinkler, S. Győri, J. Harmatos, and G. Salamon. Dimensioning WDM-based multi-layer transport networks with grooming by genetic algorithm. In *Proc. of the* 7^{th} *European Conference on Networks and Optical Communication (NOC 2002)*, pages 44–51, June 2002.

[2] G. Salamon. Spanning tree optimization problems with degree-based objective functions. In *Proc. of the* 4^{th} *Japanese-Hungarian Symposium on Discrete Mathematics and Its Applications (JH 2005)*, pages 309–315, June 2005.

[3] G. Salamon. Approximation algorithms for the Maximum Internal Spanning Tree problem. In *Proc. of the* 32^{nd} *International Symposium on Mathematical Foundations of Computer Science (MFCS 2007)*, volume 4708 of *LNCS*, pages 90–102, August 2007.

[4] G. Salamon. Feszítőfa optimalizálási problémák a Hamilton utak általánosítására, (Spanning tree optimization problems for generalizing Hamiltonian paths, in Hungarian). In *XIII. Fiatal Műszakiak Tudományos Ülésszaka, Erdélyi Múzeum-Egyesület, (Proc.)*, pages 203–206, March 2008.

[5] G. Salamon. Approximating the Maximum Internal Spanning Tree problem. *Theoretical Computer Science, MFCS 2007 special issue*, 410:5273–5284, 2009.

[6] G. Salamon. Vulnerability bounds on the number of spanning tree leaves. *Ars Mathematica Contemporanea*, 2:77–92, 2009.

[7] G. Salamon and G. Wiener. Leaves of spanning trees and vulnerability. In *Proc. of the* 5^{th} *Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications (HJ 2007)*, pages 225–235, April 2007.

[8] G. Salamon and G. Wiener. On finding spanning trees with few leaves. *Information Processing Letters*, 105:164–169, 2008.

Further References

[9] N. Alon, D. Moshkovitz, and S. Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms*, 2(2):153–177, 2006.

[10] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer-Verlag, 1999.

[11] C. A. Barefoot, R. Entringer, and H. Swart. Vulnerability in graphs — a comparative survey. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 1:13–22, 1987.

[12] D. Bauer, H. Broersma, and E. Schmeichel. Toughness in graphs — a survey. *Graphs and Combinatorics*, 22:1–35, 2006.

[13] D. Bauer, H. Broersma, and H. J. Veldman. Not every 2-tough graph is Hamiltonian. *Discrete Applied Mathematics*, 99:317–321, 2000.

[14] D. Bauer, G. Fan, and H. J. Veldman. Hamiltonian properties of graphs with large neighborhood unions. *Discrete Mathematics*, 96:33–49, 1991.

[15] J-C. Bermond. On Hamiltonian walks. In *Proc. of the* 5^{th} *British Combinatorial Conference*, pages 41–51, 1975.

[16] T. Böhme, H. J. Broersma, F. Göbel, A. V. Kostochka, and M. Stiebitz. Spanning trees with pairwise nonadjacent endvertices. *Discrete Mathematics*, 170:219–222, 1997.

[17] O. Borůvka. O jistém problému minimálním (About a certain minimal problem, in Czech). *Práca Moravské Přirodovědecké Společnosti*, 3:37–58, 1926.

[18] S. Bregni, G. Guerra, and A. Pattavina. Optical packet switching of IP traffic. In *Proc. of the* 6^{th} *Working Conference on Optical Network Design and Modelling (ONDM 2002)*, pages 135–149, February 2002.

[19] H. Broersma and H. Tuinstra. Independence trees and Hamilton cycles. *Journal of Graph Theory*, 29:227–237, 1998.

[20] M. Brunato and R. Battiti. A multistart randomized greedy algorithm for traffic grooming on mesh logical topologies. In *Proc. of the* 6^{th} *Working Conference on Optical Network Design and Modelling (ONDM 2002)*, pages 417–430, February 2002.

[21] Y. Caro, D. B. West, and R. Yuster. Connected domination and spanning trees with many leaves. *SIAM Journal on Discrete Mathematics*, 13(2):202–211, 2000.

[22] J. Cheriyan, A. Sebő, and Z. Szigeti. An improved approximation algorithm for the minimum size 2-edge connected spanning subgraph. In *Integer Programming and Combinatorial Optimization*, volume 1412 of *LNCS*, pages 126–136, 1998.

[23] V. Chvátal. Tough graphs and on Hamiltonian circuits. *Discrete Mathematics*, 5:215–228, 1973.

[24] R. J. Faudree, R. J. Gould, M. S. Jacobson, and R. H. Schelp. Neighborhood unions and Hamiltonian properties in graphs. *Journal of Combinatorial Theory Ser. B*, 47:1–9, 1989.

[25] H. Fernau, S. Gaspers, and D. Raible. Exact and parameterized algorithms for Max Internal Spanning Tree. In *Proc. of the* 35^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2009)*, volume 5911 of *LNCS*, pages 100–111, December 2009.

[26] H. Fernau, S. Gaspers, D. Raible, and A. A. Stepanov. Exact exponential time algorithms for Max Internal Spanning Tree. Arxiv preprint arXiv:0811.1875, 2008.

[27] E. Flandrin, T. Kaiser, R. Kužel, H. Li, and Z. Ryjáček. Neighborhood unions and extremal spanning trees. *Discrete Mathematics*, 308:2343–2350, 2008.

[28] T. Fujie. The Maximum-Leaf Spanning Tree problem: Formulations and facets. *Networks*, 43(4):212–223, 2004.

[29] M. Fürer and B. Raghavachari. Approximating the minimum degree spanning tree to within one from the optimal degree. In *Proc. of the* 3^{rd} *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1992)*, pages 317–324, January 1992.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

[31] L. Gargano and M. Hammar. There are spanning spiders in dense graphs (and we know how to find them). In *Proc. of the* 30^{th} *International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *LNCS*, pages 802–816, July 2003.

[32] L. Gargano, M. Hammar, P. Hell, L. Stacho, and U. Vaccaro. Spanning spiders and light-splitting switches. *Discrete Mathematics*, 285:83–95, 2004.

[33] L. Gargano, P. Hell, L. Stacho, and U. Vaccaro. Spanning trees with bounded number of branch vertices. In *Proc. of the* 29^{th} *International Colloquium on Automata, Languages and Programming (ICALP 2002)*, volume 2380 of *LNCS*, pages 355–365, July 2002.

[34] W. Goddard. Measures of vulnerability — the integrity family. *Networks*, 24(4):207–213, 1994.

[35] W. Goddard and H. C. Swart. Integrity in graphs: bounds and basics. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 7:139–151, 1990.

[36] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. In *Proc. of the* 3^{rd} *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1992)*, pages 307–316, January 1992.

[37] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. In *Proc. of the* 4^{th} *European Symposium on Algorithms (ESA 1996)*, volume 1136 of *LNCS*, pages 179–193, September 1996.

[38] S. Guha and S. Khuller. Improved methods for approximating node weighted Steiner trees and connected dominating sets. In *Proc. of the* 18^{th} *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1998)*, volume 1530 of *LNCS*, pages 54–65, December 1998.

[39] R. Hassin and A. Tamir. On the Minimum Diameter Spanning Tree problem. *Information Processing Letters*, 53:109–111, 1995.

[40] J.-M. Ho, D.T. Lee, C.-H. Chang, and C.K. Wong. Minimum diameter spanning trees and related problems. *SIAM Journal of Computing*, 20:987–997, 1991.

[41] W. T. Huh. Finding 2-edge connected spanning subgraphs. *Operations Research Letters*, 32:212–216, 2004.

[42] K. Jain and V. V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Proc. of the* 40^{th} *Symposium on Foundations of Computer Science (FOCS 1999)*, pages 2–13, October 1999.

[43] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer System Science*, 9:256–278, 1974.

[44] H. A. Jung. On a class of posets and the corresponding comparability graphs. *Journal of Combinatorial Theory Ser. B*, 24:125–133, 1978.

[45] D. Karger, R. Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. In *Proc. of the* 3^{rd} *Workshop on Algorithms and Data Structures (WADS 1993)*, volume 709 of *LNCS*, pages 421–432, August 1993.

[46] G. Y. Katona, A. Recski, and R. Szabó. *A számítástudomány alapjai, (Introduction to computer science, in Hungarian)*. Typotex, 2002.

[47] A. Kirlangiç. Scattering number in graphs. *International Journal of Mathematics and Mathematical Sciences*, 30(1):1–8, 2002.

[48] M. Knauer and J. Spoerhase. Better approximation algorithms for the maximum internal spanning tree problem. In *Proc. of the* 11^{th} *Workshop on Algorithms and Data Structures (WADS 2009)*, volume 5664 of *LNCS*, pages 459–470, July 2009.

[49] D. Kratsch, T. Kloks, and H. Müller. Computing the toughness and the scattering number for interval and other graphs. Technical Report 2237, Institut National de Recherche en Informatique et Automatique (INRIA), March 1994.

[50] J. B. Kruskal. On the shortest spanning subtree of a graph and the Travelling Salesman problem. *Proc. of American Mathematical Society*, 7:48–50, 1956.

[51] P. Krysta and V. S. A. Kumar. Approximation algorithms for the minimum size 2-connectivity problems. In *Proc. of the* 18^{th} *Symposium on Theoretical Aspects of Computer Science (STACS 2001)*, volume 2010 of *LNCS*, pages 431–442, February 2001.

[52] V. E. Levit and E. Mandrescu. The intersection of all maximum stable sets of a tree and its pendant vertices. *Discrete Mathematics*, 308:5809–5814, 2008.

[53] H.-I. Lu and R. Ravi. The power of local optimization: approximation algorithms for maximum-leaf spanning tree. In *Proc. of* 30^{th} *Annual Allerton Conference on Communication, Control and Computing*, pages 533–542, October 1992.

[54] H.-I. Lu and R. Ravi. The power of local optimization: Approximation algorithms for maximum-leaf spanning tree (draft). Technical Report CS-96-05, Department of Computer Science, Brown University, Providence, Rhode Island, 1996.

[55] H.-I. Lu and R. Ravi. Approximation for maximum leaf spanning trees in almost linear time. *Journal of Algorithms*, 29(1):132–141, 1998.

[56] O. Ore. Note on Hamiltonian circuits. *American Mathematical Monthly*, 67:55, 1960.

[57] E. Prieto. *Kernelization in FPT Algorithm Design*. PhD thesis, The University of Newcastle, Australia, 2005.

[58] E. Prieto and C. Sloper. Either/or: Using vertex cover structure in designing FPT-algorithms — the case of k-internal spanning tree. In *Proc. of the* 8^{th} *Workshop on Algorithms and Data Structures (WADS 2003)*, volume 2748 of *LNCS*, pages 465–483, July 2003.

[59] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[60] C. Savage. Depth-first search and the Vertex Cover problem. *Information Processing Letters*, 14:233–235, 1982.

[61] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1998.

[62] A. Schrijver. *Combinatorial Optimization, Polyhedra and Efficiency. Vol. B*, chapter 50: Shortest spanning trees, pages 855–876. Springer-Verlag, 2003.

[63] K. M. Sivalingam and S. Subramaniam. *Optical WDM Networks: Principles and Practice*. Kluwer Academic Publishers, London, 2000.

[64] R. Solis-Oba. 2-approximation algorithm for finding a spanning tree with maximum number of leaves. In *Proc. of the* 6^{th} *European Symposium on Algorithms (ESA 1998)*, volume 1461 of *LNCS*, pages 441–452, August 1998.

[65] C. Swamy and A. Kumar. Primal-dual algorithms for connected facility location problems. In *Proc. of the* 5^{th} *International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2002)*, volume 2462 of *LNCS*, pages 256–270, September 2002.

[66] M Tsugaki and T. Yamashita. Spanning trees with few leaves. *Graphs and Combinatorics*, 23:585–598, 2007.

[67] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 10: Graph Algorithms. Elsevier, 1990.

[68] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[69] S. Vempala and A. Vetta. Factor 4/3 approximations for minimum 2-connected subgraphs. In *Proc. of the* 3^{rd} *International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2000)*, volume 1913 of *LNCS*, pages 262–273, September 2000.

[70] H. Whitney. Nonseparable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1934.

[71] S. Win. On a conjecture of Las Vergnas concerning certain spanning trees in graphs. *Resultate Math.*, 2:215–224, 1979.

[72] B. Y. Wu and K.-M. Chao. *Spanning Trees and Optimization Problems*. Chapman & Hall / CRC, 2004.

[73] S. Zhang, X. Li, and X. Han. Computing the scattering number of graphs. *International Journal of Computer Mathematics*, 79(2):179–187, 2002.

[74] S. Zhang and Z. Wang. Scattering number in graphs. *Networks*, 37:102–106, 2001.