Searching for Optimal Caterpillars in General and Bounded Treewidth Graphs

Masoud Khosravani

A dissertation submitted to the Department of Computer Science
University of Auckland
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Supervisor

Michael J. Dinneen

Department of Computer Science

University of Auckland

October 2011

To my mother
with respect and love

Abstract

Searching for Optimal Caterpillars in General and Bounded Treewidth Graphs

Masoud Khosravani

Department of Computer Science

Doctor of Philosophy

In this thesis we are interested in optimization problems on caterpillar trees. A caterpillar is a tree with this property that if one removes its leaves only a path is left. The majority of this thesis is devoted to studying the Minimum Spanning Caterpillar Problem (MSCP). An instance of the MSCP is a graph with dual costs over its edges. In the MSCP our goal is to find a caterpillar tree that spans the input graph with the smallest overall cost. The cost of the caterpillar is the sum of the cost of its edges, where each edge takes one of two costs based on its role as a leaf edge or an internal one.

We first show that the problem of finding a spanning caterpillar in a graph is **NP**-complete. As another result on the hardness of the MSCP, we show that there is no $f(n)$-approximation algorithm for the MSCP unless $\mathbf{P} = \mathbf{NP}$. Here $f(n)$ is any polynomial-time computable function of n, the number of nodes of a graph. Then we introduce a quadratic integer programming formulation for the MSCP. By using the Gomory cutting method iteratively, we show that one can find a near optimal solution. We then show that our integer programming formulation can be transformed to a semi-definite programming problem.

A parametrized algorithm that finds an optimal solution for the MSCP in bounded treewidth graphs is given in Chapter 4. Our algorithm is fast and practical for outer-planar, series parallel, Halin graphs and other graphs with small treewidth.

The Maximum Caterpillar Problem (MCP) is the topic of the last two chapters of this thesis. We show that there is a polynomial-time algorithm for finding a caterpillar of the largest order in directed acyclic graphs. Then we introduce a heuristic algorithm for the MCP. Finally, in Chapter 6 we prove that every interval graph has a spanning caterpillar and also we present an algorithm for the MCP in block graphs.

Acknowledgement

First of all I wish to express my debt and gratitude to my supervisor Michael J. Dinneen for his full support during the past three years.

Thanks to the Department of Computer Science who provided me a scholarship which helped me to focus on my research. I wish also to thank all people in the department specially Bakhadyr M. Khoussainov, Mark C. Wilson and Clark D. Thomborson.

Within the last three years I shared my office with Alistair Abbot, Imran Khaliq, Jiamou Liu, Alexander Melnikov, Reyhaneh Reyhani, Norsaremah Salleh, Moon Ting Su and Fu'ad Al Tabba. I would like to thank all of them to make the life easier in our crowded office with their friendly attitudes.

Thanks to Sonny Datt for spotting a few typos in a table of Chapter 4 while he was implementing one of the algorithms as his summer project.

Finally I want to thank my family for their support and love, with special thanks to my mother.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Whenever one wishes to model relations or dependencies among objects, the first model that comes to mind is a graph. That is why graphs are ubiquitous in computer science and related fields. By making such a model many questions can be addressed using the language of graph theory. For example:

- What is the largest set of objects with no mutual relationships? In graph theory term we are looking for a maximum *independent set*.
- How can one partition the objects into disjoint sets such that each set contains non-relating objects? A proper *vertex colouring* gives the answer to this question.
- How large is a set of objects in which every pair of objects are related? Here we wish to find a maximum *clique*.

In some applications one may wish to find a specific substructure within a graph. For example one may want to find a spanning tree to check the connectivity. In the Hamiltonian path problem we are looking for a path within a graph that covers all nodes in the graph. There are other applications where weights are assigned to links and/or nodes, to represent cost or traversal time. In this group of problems we look for a substructure with the optimal cost. The travelling salesman problem is a well studied example of such problems, where the goal is to find a spanning cycle with the minimum travel cost.

Many of those graph problems are computationally intractable. That is, no one knows a polynomial-time algorithm for solving any of them. So researchers

are using different techniques to find some solutions that are *as good as possible*. Here we focus on some of those hard problems about caterpillar trees.

1.1 Caterpillar Trees

In this thesis we are interested in problems concerning caterpillar trees. A *caterpillar* is a tree that is reduced to a path when one removes its leaves. We shall introduce the formal definition in the next chapter. As a subgraph we can consider a caterpillar as a path that holds its neighbours. Despite the fact that caterpillars are simple in structure, we show that almost all natural problems about them are hard to solve. Like finding a minimum spanning caterpillar in an edge-weighted graph or finding a caterpillar with the largest number of nodes in a graph.

Let us first mention a few of our motivations for studying caterpillar trees. Consider a graph with no Hamiltonian path. Then we may relinquish finding a spanning path in favour of a spanning caterpillar. In another scenario suppose that the cost of a minimum route in an instance of the travelling salesman problem is too large. In such a case a natural alternative is to consider a path and its neighbouring nodes as the desired substructure. In the following section we will introduce more real world problems whose solutions rely on finding a caterpillar.

Due to our knowledge, the term of *caterpillar* is traced back to a paper of Harary and Schwenk [40]. Though afterwards many papers were published about combinatorial properties of them, there are a few results addressing the algorithmic point of view. In recent years, there have been more interests in algorithmic studies of caterpillars. In [32] Feige and Talwar present an approximation algorithm for bandwidth of caterpillar. In [35] Gonçalves proves that every planar graph can be covered by four edge disjoint caterpillars and he also presents an algorithm to find such caterpillars.

1.2 Applications

In this section we mention a few more applications of caterpillars. In many problems where a path plays a main role, a caterpillar tree can be considered as an alternative. So their applications are not restricted to the following ones.

1. In network design, we may wish to find a cost effective linearly arranged backbone to place our communication routers. Here the backbone is the spine of a caterpillar and leaves are the sites that are connected to the backbone. Generally the cost of backbone links is different with the cost of links that connect a site to a router.

2. In a facility transportation problem, where the task of distributing facilities is divided among one global but costly distributor and some local and cheap ones. Here the global distributor follows the spine path to deliver facilities to warehouses and the local ones use the leaf edges to distribute them among end users. The goal is to find a transportation route that has the minimum overall cost.

3. In chemical graph theory caterpillars are considered as a model for benzenoid hydrocarbon molecules; see [29].

4. Caterpillars also appear in designing algorithms for solving RNA alignment and comparing evolutionary trees; see [23] and [45].

For more applications in combinatorics and mathematics (in general) we refer the reader to [56, 57].

1.3 Contributions and Outline of the Thesis

We now review the outline of the thesis, with the main results summarized in Table 1.1. In this table we present the complexity of computing the Maximum Caterpillar Problem (MCP) and the Minimum Spanning Caterpillar Problem (MSCP) in general and some special classes of graphs. We sketch the contents of each chapter with more details in the following section.

Table 1.1 Summary of results.

Graph Classes	MCP	MSCP	Remarks
general and planar graphs	NP-hard	NP-hard	Chapters 2, 3
bounded treewidth	linear	linear	Chapter 4
bounded pathwidth	linear	linear	Chapter 4
series-parallel	linear	linear	Chapter 4
outerplanar	linear	linear	Chapter 4
Halin graphs	linear	linear	Chapter 4
interval graphs	linear	NP-hard	Chapter 6
block graphs	linear	NP-hard	Chapter 6

1.3.1 Chapter 2: Preliminaries

To crack the complexity wall of the problems concerning caterpillars, we use different tools. In Chapter 2, after introducing the required definitions in graph theory and combinatorics, we review the main concepts in approximation algorithms, which is the main theme of Chapter 3. Also we give an introduction to linear and integer programmings that will be used in designing and analysing approximation algorithms. Then we introduce the notion of path and tree decomposition, which are concepts that are used in our parametrized algorithm for solving the minimum spanning caterpillar problem.

1.3.2 Chapter 3: Minimum Spanning Caterpillar Problem

Graphs with given weights on their edges are models for many real world problems. In Chapter 3 we study the minimum spanning caterpillar problem, where we have dual costs on each edge. As our first result we show that if $\mathbf{P} \neq \mathbf{NP}$ then there is no $f(n)$-approximation algorithm for the minimum spanning caterpillar problem, where $f(n)$ is any polynomial-time computable function. Then as a base for heuristic algorithm we introduce a quadratic integer programming formulation for the problem. We show that by using Gomory cuts iteratively, one can find any desirable approximation in *exponential time*.

1.3.3 Chapter 4: MSCP in Bounded Treewidth Graphs

Parametrization plays a key role when coping with the hardness of a problem. The main idea in parametrization is to restrict the input to some special classes that are accompanied by preprocessing knowledge. In Chapter 4 we give a linear time algorithm for the minimum spanning caterpillar problem when restricted to bounded treewidth graphs by assuming that a tree/path decomposition of the input graph is given.

1.3.4 Chapter 5: Maximum Caterpillar Problem

Finding a largest caterpillar in a graph is the topic of Chapter 5. We first give an integer programming formulation for this problem. Though our basic formulation is not linear, we can transform it to an equivalent linear programming problem. The main price that we pay is a linear increase in the number of variables. Then we use the solution to the relaxation of our integer linear programming problem, to present an approximation algorithm.

1.3.5 Chapter 6: Miscellaneous Algorithms

The last chapter is devoted to miscellaneous caterpillar algorithms. We present an algorithm that finds a maximum caterpillar in a block graph. We show that every interval graph has a spanning caterpillar, we also present an algorithm to extract such caterpillar. Finally, we generalize breadth first search and depth first search to k-trees.

1.3.6 Publications

This thesis is based on the results of the following papers.

1. *Hardness of Approximation and Integer Programming framework for Searching for Caterpillar Trees*, Michael J. Dinneen and Masoud Khosravani, In Alex Potanin and Taso Viglas, editors, Computing: The Australian Theory Symposium (CATS 2011), volume 119 of CRPIT, pages 145–150, Perth, Australia, 2011.

2. *A Linear Time Algorithm for the Minimum Spanning Caterpillar Problem for Bounded Treewidth Graphs*, Michael J. Dinneen and Masoud Khosravani, In Boaz Patt-Shamir and Tinaz Ekim, Structural information and Communication Complexity (SIROCCO 2010), LNCS 6058, 237-246, Springer Berlin/Heidelberg, 2010.

3. *Searching for spanning k-Caterpillar and k-trees*, Michael J. Dinneen and Masoud Khosravani, CDMTCS Report 336, University of Auckland, Auckland, 2008.

Here are my other publications that their topics are out of the scope of the current thesis.

1. *Arithmetic Progression Graphs*, Michael J. Dinneen, Nan R. Ke and Masoud Khosravani. CDMTCS Report 356, , University of Auckland, Auckland, 2009.

2. *Using OpenCL for Implementing Simple Parallel Graph Algorithms*, Michael J. Dinneen, Masoud Khosravani and Andrew Probert, PDPTA'11, 2011.

Chapter 2

Preliminaries

In this chapter we introduce basic concepts and tools that are used in the thesis. In the first section we introduce the main definitions and notations in graph theory and combinatorics. In Section 2.2 we give a short review on approximation algorithms and the notion of approximation ratio. Then in Section 2.3 we have a short review on linear programming. The topics of Section 2.4 are tree and path decompositions. We also introduce the concepts of partial k-trees and k-paths that are closely related to tree and path decompositions. Then we give formal definitions of some classes of graphs with small treewidth. Finally, in the last section we give a simple proof for the hardness of finding a spanning caterpillar in a graph.

2.1 Basic Combinatorial Structures

In this section we introduce the main definitions in graph theory and combinatorics. These concepts are used throughout this thesis. We follow the definitions and notations that are used in the standard text books of Diestel [24] and Bondy and Murty [13].

A *graph* is a pair $G = (V, E)$, where V is the set of *nodes* and E is a subset of $[V]^2 = \{\{u, v\} \mid u, v \in V\}$. In this thesis we assume all graphs have no loops or multiple edges. We refer to the elements of E as *edges* of G. A *directed graph* $D = (N, A)$ consists of a set of nodes N and a set of *arcs*, $A \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$. When dealing with more than one graph and to make discussion more clear we

denote the nodes and the edges of a graph G by $V(G)$ and $E(G)$, respectively.

A *weighted graph* is a graph that has weights over its nodes or/and edges. If the weights are assigned just to nodes or edges of a graph, then we refer to that graph as a *node-weighted graph* or an *edge-weighted graph*, respectively.

The *order* of a graph $G = (V, E)$ is the number of nodes in the graph, that is denoted by $|V|$. The *size* of a graph, $|E|$, is the number of edges. The same definitions apply to directed graphs. For every edge $e = \{u, v\}$ we say u and v are *ends* of e and we also say u is adjacent to v. We also refer to adjacent nodes as *neighbours*. The degree $d(v)$ of a node v is defined as the number of edges that are attached to v. For a node v in a directed graph D we also define *in-degree* of v, $in(v)$, as the set of arcs that are pointing to v, i.e. $in(v) = \{(u, v) \mid (u, v) \in A\}$. We also define *out-degree* of v, $out(v)$, as the number of arcs that emanate from v, i.e. $out(v) = \{(v, w) \mid (v, w) \in A\}$.

A *path* $P = v_0, e_0, v_1, \ldots, e_{n-1}, v_n$ in a graph G is an alternating sequence of distinct nodes and edges, where each edge e_i, $i = 0, \ldots, n-1$, connects two adjacent nodes v_i and v_{i+1}. The *length* of a path is the number of edges on the path. A graph is *connected* if there is a path between each pair of its nodes. A *cycle* $C = v_0, e_0, v_1, e_1, \ldots, e_{n-1}, v_0$ is an alternating sequence of nodes and edges such that all nodes on it but the first and the last one are distinct and the subsequence $v_0, e_0, \ldots, v_{n-1}$ makes a path.

A graph $H = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. We denote it by $H \subseteq G$. A *clique* is a subgraph in which every two nodes are connected by an edge.

We say H is an *induced* subgraph of a graph G if for every $u, v \in V(H)$ if $\{u, v\} \in E(G)$ then we have $\{u, v\} \in E(H)$. We say that $H \subseteq G$ is a *spanning subgraph* of G if $V(H) = V(G)$.

If $G = (V, E)$ is a graph and $U \subseteq V$ is any set of nodes, $G[U]$ denotes the induced subgraph on U. For each $H \subseteq G$ we denote its *neighbourhood* (or 1-*neighbourhood*) by

$$N(H) = \{v \mid \exists u \in V(H), (u, v) \in E(G) \text{ and } v \notin V(H)\}.$$

The *closed neighbourhood* of H is denoted as $N[H] = N(H) \cup V(H)$.

A *tree* is a graph that is connected and has no cycle. If a tree has more than

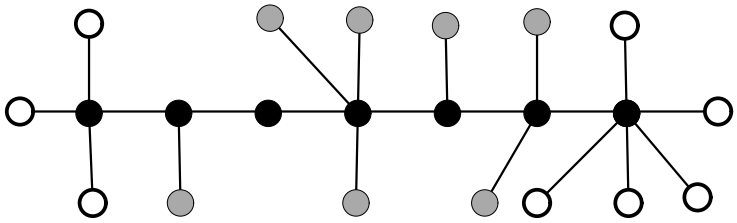

Figure 2.1 A caterpillar: spines (black), heads (white), and leaves (gray)

one node then every node of degree one is called a *leaf* of a tree. A *complete graph* K_n is a graph of order n such that every pair of its nodes are connected by an edge. A *star* S_n is a tree with $n+1$ nodes that has n nodes of degree one.

A *caterpillar* tree is a tree that is reduced to a path when all its leaves are deleted. We refer to the resulting path as the *spine* of a caterpillar. The *heads* of a caterpillar are nodes that are connected to the two ends of degree one or a single node of the path representing the spine. In Figure 2.1 we see a caterpillar tree with designated spines (black), heads (white), and leaf nodes (gray).

Let S be a set, we say $Q = \{S_1, S_2, \ldots, S_m\}$ is a *partition* of S if

1. $S_i \subset S$, for each i, $1 \leq i \leq m$, and
2. $S_i \cap S_j = \emptyset$, for each pair i, j, $1 \leq i \neq j \leq m$, and
3. $S = \bigcup_{i=1}^{m} S_i$.

The number of partitions of an n-element set is called the n-th *Bell number* and is denoted by B_n. Berend and Tassa in [8] show that the n-th Bell number B_n is restricted by the following bound.

$$B_n < \left(\frac{0.782n}{\ln(n+1)} \right)^n .$$

2.2 Approximation Algorithms

When one proves that an optimization problem is **NP**-hard, his hope for finding an exact solution for the problem fades. In such cases finding a proper approximation algorithm is another choice. For more on approximation algorithms consult the text book [59] of Vazirani.

There are two measures to express the closeness of an approximating solution to the optimal one. The first one is *approximation factor* $f(n)$, where $f(n)$ is any computable function of input. Suppose that we have a maximization problem Π. We say an approximation algorithm A is an $f(n)$ factor approximation for Π if the solution S_A of the algorithm has the following relation with the optimal solution OPT_Π.

$$f(n)OPT_\Pi \leq S_A \leq OPT_\Pi$$

note that here we have $f(n) \leq 1$. In this case we say the algorithm is an $f(n)$-approximation for the maximization problem.

If Π is a minimization problem and A is an approximation algorithm, then we say A is a $g(n)$ factor approximation if the following holds for a solution s_A with respect to the optimal solution opt_Π.

$$opt_\Pi \leq s_A \leq g(n)opt_\Pi,$$

where $1 \leq g(n)$. We refer to such an algorithm as a $g(n)$-approximation for the minimization problem.

Another measure of closeness of the solution of an approximation algorithm is *approximation ratio*. Let Π be a maximization problem and also suppose we have an approximation algorithm A for it. As before, assume that S_A is the value of algorithm and OPT_Π is the optimal value, then we define

$$\sigma_A = \frac{S_A}{OPT_\Pi}.$$

If Π is a minimization problem and and s_A is the solution of the algorithm then we have

$$\sigma_A = \frac{opt_\Pi}{s_A}.$$

Note that the value of an approximation ratio is always greater than one.

When we analyse an approximation algorithm for an **NP**-hard problem, the value of optimal solutions are not revealed to us. So we cannot compare the exact value of an optimal solution with the approximation solution. To deal with that problem we try to find upper bounds for maximization problems. Since by

having an upper bound U in a maximization problem Π we have

$$OPT_{\Pi} \leq U.$$

Now for the solution S_A if we have $f(n)U \leq S_A$ then

$$f(n)OPT_{\Pi} \leq f(n)U \leq S_A.$$

Which says that our algorithm is an $f(n)$-approximation for Π.

If Π is a minimization problem then we find a lower bound L for the optimal value of the problem and since

$$L \leq opt_{\Pi},$$

if we have $s_A \leq g(n)L$ then

$$s_A \leq g(n)L \leq g(n)opt_{\Pi}.$$

Which means the algorithm is a $g(n)$-approximation algorithm.

If we have an integer linear programming formulation for an optimization problem, then we can find an upper or lower bound for the problem. In the next section we present elements of integer programming.

2.3 Linear and Integer Programming

As mentioned in the former section, to analyse an approximation algorithm a key step is to find a proper upper or lower bound for the optimal value. Integer programming plays not only an important role in finding upper or lower bound for optimization problem, but also it is a base for designing an approximation algorithm. There are many text books on this subject. Here we follow Papadimitriou and Steiglitz [46]. As an another well known reference we have to mention the book by Bazaraa et al. [7].

There are two main obstacles on this way. First of all the problem should have an integer programming formulation with a polynomial size. Also the formulation should belong to the group of integer programming problems that

their relaxations have polynomial-time solutions or at least they have a near-optimal approximation algorithm.

Let us first define a general nonlinear programming problem. The goal is to find a value of $\mathbf{x} \in \mathbb{R}^n$ such that it maximizes (minimizes) the value of $f(\mathbf{x})$ while satisfying the following constraints.

$$g_i(\mathbf{x}) \leq 0, \quad 1 \leq i \leq m,$$
$$h_j(\mathbf{x}) = 0, \quad 1 \leq j \leq p.$$

Here g_is and h_is are functions from $\mathbb{R}^n$ to $\mathbb{R}$. A value that satisfies the constraints is called a *feasible solution*.

In a *linear programming* problem our goal is to find a vector

$$\mathbf{x_0} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$$

such that an objective function

$$f(\mathbf{x}) = \sum_{j=1}^{n} c_j x_j$$

takes the maximal value for $\mathbf{x} = \mathbf{x_0}$ subject to constraints

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \ , \quad i = 1, \ldots, m \text{ and } x_j \geq 0, j = 1, \ldots, n.$$

If we have an extra condition in which all x_j should also be integer then we say the formulation is an *integer linear programming* problem.

2.3.1 Transforming to Binary Problems

Here we show how a polynomial integer programming problem can be transformed to a formulation with binary solutions. In this and the following subsection we are using the same procedure as explained in [60].

We first need to impose this condition on the problem that the *objective function* is bounded.

By having that condition then we can transform the problem into the equivalent binary problem by substituting each integral variable by its binary representation. If $0 \leq x_j \leq d_j$, then we substitute each variable x_j by its binary representation

$$x_j = 2^0 y_1 + 2^1 y_2 + \ldots + 2^{r_j - 1} y_{r_j} ,$$

where $y_t \in \{0, 1\}$ for $t = 1, \ldots, r_j$ and also $2^{r_j - 1} \leq d_j \leq 2^{r_j} - 1$ for $j = 1, \ldots, n$.

2.3.2 Linearization

In this section we assume that we have a nonlinear binary integer programming problem. Our goal is to transform it to a linear binary problem.

To this end we first substitute each product $\Pi_{j \in N_k} x_j = x_{j_1} x_{j_2} \cdots x_{j_q}$ of variables $x_{j_1}, \ldots, x_{j_q}$ by a binary variable y_k, such that

$$y_k = \Pi_{j \in N_k} x_j = x_{j_1} x_{j_2} \cdots x_{j_q} . \tag{2.1}$$

Here we assume that there are k products in the nonlinear problem. We also assume that N_k is the index set of those variables that are involved in the kth product and $q = \mid N_k \mid$.

Then we add two linear constraints to link the values of the binary variables with the values of the products. The constraints are

$$\sum_{j \in N_k} x_j - y_k \leq |N_k| - 1, \tag{2.2}$$

$$\frac{1}{|N_k|} \sum_{j \in N_k} x_j - y_k \geq 0. \tag{2.3}$$

We show now that the suggested formulation determines exactly the same value of y_k as 2.1. If $x_j = 1$ for all $j \in N_k$, then from Inequality 2.2 we have $y_k \geq 1$ and also 2.3 gives $y_k \leq 1$. So in this case, $y_k = 1$, exactly as in 2.1. If on the other hand at least for one $j \in N_k$, $x_k = 0$, then from 2.3, $y_k < 1$, and thus $y_k = 0$, again the same as in 2.1.

2.4 k-Trees and Tree Decompositions

Having a spanning tree is considered a measure of reliability in a network and one may need to find a spanning k-tree or k-caterpillar as a more reliable substructure in a network; since a network with such substructure would be immune to $k-1$ nodes or $k-1$ links failures [4,51].

A graph G is a *k-tree* if G is a complete graph with $k+1$ nodes or G is obtained recursively from a k-tree G' by attaching a new node to an induced k-clique of G'. The first k-clique in this process is called the *base* of a k-tree. Each node of degree k in a k-tree is called a *k-leaf*. A *partial k-tree* is any subgraph of a k-tree. Informally, a k-caterpillar is built when one restricts the process of attaching a new node to k nodes of one of those *last* k-cliques that has been formed. To define a k-caterpillar formally we need a few more definitions that are given in the following paragraph.

A *simplicial node* of a graph is a node whose neighbourhood induces a clique. An ordering of the nodes $\sigma = [v_1, \ldots, v_n]$ is called a *perfect elimination scheme* if for every $1 \leq i \leq n$, v_i is a simplicial node in $G[v_i, \ldots, v_n]$. If v_i is a node in a perfect elimination scheme then we refer to each clique in $N(v_i) \cap \{v_i, \ldots, v_n\}$ as a parent of v_i. We say two k-cliques are *smooth neighbours* if the induced subgraph of their union has a perfect elimination scheme. Note that each k-tree has a perfect elimination scheme.

A $(k+1)$*-boundaried graph* is a pair (G, ∂) of a graph $G = (V, E)$ and an injective function ∂ from $\{0, \ldots, k\}$ to V. The image of ∂ is the set of *boundaried nodes* and is denoted by $Im(\partial)$. When it is clear from the context, we abuse the notation and refer to $Im(\partial)$ as ∂.

A graph is a *k-caterpillar* if it is

1. a $(k+1)$-boundaried complete graph with $(k+1)$ nodes, or
2. a $(k+1)$-boundaried graph (G, ∂) that is made by attaching a new node v to k nodes of $Im(\partial')$ from a $(k+1)$-boundaried graph (G', ∂'), such that $Im(\partial) = N[v]$.

It is easy to see that each k-caterpillar is also a k-tree. Some authors use the term *k-path* instead of k-caterpillar but since k-path also refers to a different concept in

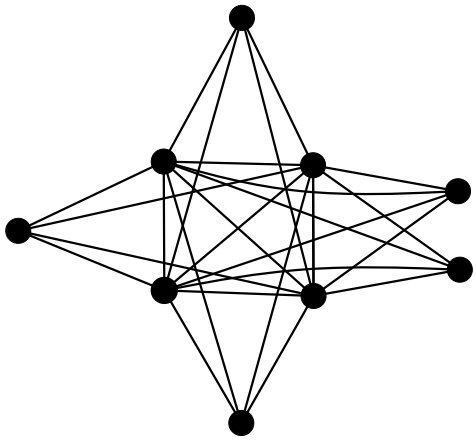

Figure 2.2 A 4-star.

the Mathematics and Computer Science literature, we use the term k-caterpillar to avoid any confusion, for example see [37, 43].

If one deletes all k-leaves of a k-caterpillar we refer to the remaining subgraph as a *k-spine*. A *k-star* is a k-caterpillar that has a k-clique as its k-spine. See Figure 2.2. If a k-caterpillar G is not a k-star, then its k-spine can be considered as an alternating sequence of distinct k-cliques and $(k+1)$-cliques $(e_0, t_1, \ldots, t_n, e_n)$, where e_is are k-cliques and t_is are $(k+1)$-cliques, and the sequence starts and ends with k-cliques, e_0 and e_n; see Proskurowski [47]. Each k-caterpillar has at most two *heads*. Each head is a subgraph induced by the union of e_j, where $j = 0, n$, and the k-leaves attached to it.

Bern [9] showed that for all $k \geq 2$ the problem of deciding whether a graph has a spanning k-tree is **NP**-complete. He also gave an approximation algorithm for finding a minimum spanning k-tree in a weighted graph, using an idea of Farley [31]. Cai and Maffraye [17] proved that the problem remains **NP**-complete even when it is restricted to split graphs, graphs with maximum degree $3k+2$, and planar graphs (for $k = 2$). Later, Cai [16] presented an approximation algorithm for finding a minimum spanning 2-tree in graphs whose edge weights satisfy the triangle inequality and graphs that are complete Euclidean graphs on a set of points in the plane.

For results concerning spanning 2-trees see Farley [31]. Recently, Tan and Zhang [57] used caterpillars to solve some problems concerning the Consec-

utive Ones Property. They also showed that the spanning caterpillar problem in graphs with maximum degree 3 is **NP**-complete. Gupta et al. [38] presented $O(n^3)$ algorithms for some related problems, such as subgraph isomorphism, topological embedding and minor containment problems in k-connected k-caterpillars.

One can define partial k-trees as graphs with treewidth at most k. A *tree (path) decomposition* [50] of a graph $G = (V, E)$ is a pair $(\{X_i , i \in I\}, T = (I, F))$, with $\{X_i , i \in I\}$ a collection of subsets of V, that are called *bags*, and $T = (I, F)$ a tree (path), such that

1. $\bigcup_{i \in I} X_i = V$.
2. For all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$.
3. For all $v \in V$, $T_v = \{i \in I \mid v \in X_i\}$ forms a connected subtree of T.

The width of a tree (path) decomposition $(\{X_i , i \in I\}, T = (I, F))$ is defined as $\max_{i \in I} |X_i| - 1$. The *treewidth* (*pathwidth*) of G, $tw(G)$ ($pw(G)$), is the minimum width over all tree (path) decompositions of G. A tree (path) decomposition of width k is a *smooth tree decomposition* if each bag contains $k + 1$ nodes and adjacent bags differ by exactly one node.

The concept of treewidth became widely known by Robertson and Seymour's work on graph minors [50]. Although, the concept had appeared under different names and definitions in the literature. As the first papers on this topic we refer the reader to Halin [39] and Rose [51]. Here we use the *constructive* definition of partial k-trees, since as we show later, it is closer to our k-parse data structure for representing graphs of bounded treewidth. For an up-to-date survey on treewidth refer to Bodlaender [12].

One natural question that arises is whether the problem of finding a spanning k-caterpillar is easier than finding a spanning k-tree. In the next section we give a negative answer to this question by showing that the problem remains **NP**-complete even when we are restricted to searching for a spanning k-caterpillar in a graph, for any $k \geq 1$. Graph structures that are expressible by Monadic Second Order Logic [21] are recognizable in linear time on partial k-trees. So does the property of having a spanning k-caterpillar in graphs. The

main disadvantage of this method is that it is very hard to implement, even for small values of k.

2.5 Graphs with Small Treewidths

In Chapter 4 we introduce a linear time algorithm for solving the minimum spanning caterpillar problem for bounded treewith graphs. In this section we give the formal definitions of three classes of graphs that have treewidths less than 4. They are: outerplanar graphs, Halin graphs and series parallel graphs; see Figure 2.3.

A graph is an *outerplanar* graph if it can be embedded on the plane with no crossing among the edges such that all of its nodes appear on the unbounded outer face. Note that with respect to the definition we can add an extra node to an outerplanar graph that is adjacent to all other nodes with no crossing. An outerplanar graph has treewidth at most two.

A *Halin graph* [55] is constructed from a tree by the following process. We first embed the tree on the plane with no crossing. Then we connect its leaves via a cycle with the same order as their appearances in clockwise order on the unbounded face. Every Halin graph has treewidth at most three.

A two terminal graph is a graph (G, s, t) with two distinguished nodes s and t, called sink and source, respectively. Every *series-parallel* graph is constructed by the following recursive process.

1. The complete graphs K_1 and K_2 are series-parallel graphs.
2. If (G_1, s_1, t_1) and (G_2, s_2, t_2) are series-parallel terminal graphs, then the graph obtained from the disjoint union of G_1 and G_2 by identifying t_1 and s_2 is also a series-parallel graph.
3. If (G_1, s_1, t_1) and (G_2, s_2, t_2) are series-parallel terminal graphs, then the graph obtained by identifying the sinks and also the sources is a series-parallel graph.

A graph has treewidth at most two if and only if it is a series-parallel graph.

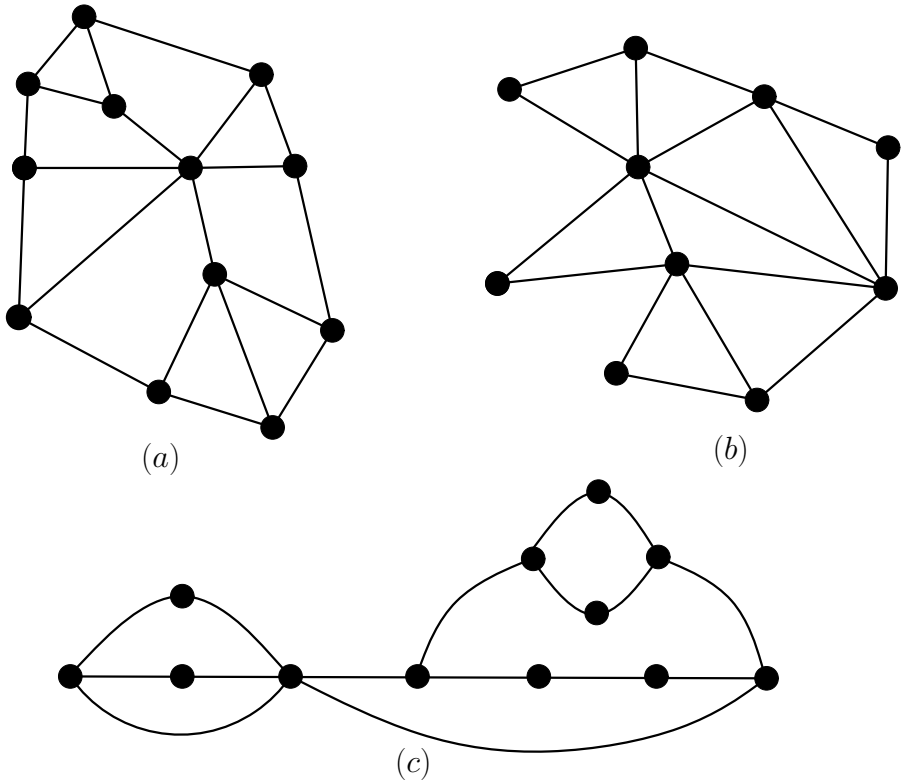

Figure 2.3 Graphs of small treewidths: (a) a Halin graph, (b) an outerplanar, and (b) a series-parallel graph

2.6 Hardness of Finding Spanning Caterpillars

As we noted earlier, Bern [9] showed that, for a fixed k, finding a spanning k-tree in a graph is **NP**-complete. In the following theorem we prove that when the problem is restricted to finding a spanning k-caterpillar, it still remains intractable. Let us state the problem formally.

Problem: SPANNING k-CATERPILLAR
Instance: A graph $G = (V, E)$,
Question: Does G contain a spanning k-caterpillar?

Theorem 1 *For each $k \geq 1$,* SPANNING k-CATERPILLAR *is* **NP***-complete.*

Proof. We prove the theorem by transforming the Hamiltonian path problem to the spanning k-caterpillar problem. Let $G = (V, E)$ be an instance of the

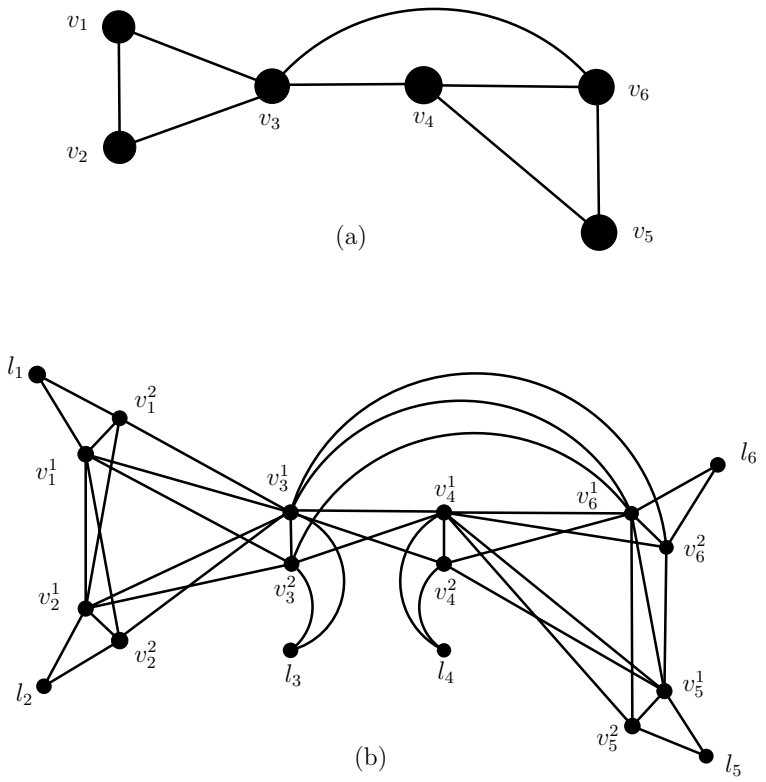

Figure 2.4 (a) A graph G and (b) its corresponding 2-caterpillar $\tilde{G}$

Hamiltonian path problem. We construct a graph $\tilde{G} = (\tilde{V}, \tilde{E})$ such that $|\tilde{V}| = (k+1)|V|$ and $|\tilde{E}| = (k/2)(k+1)(|E| + |V|)$. To each $v \in V$ we assign a k-clique $\tilde{K}_v$ that we label its nodes by $\{\tilde{v}^1, \ldots, \tilde{v}^k\}$. For each two adjacent nodes v and w in G we assign $k(k+1)/2$ edges in $\tilde{G}$ by connecting each $\tilde{v}$ to $\left\{\tilde{w}^1, \ldots, \tilde{w}^{(k-i+1)}\right\}$, $1 \leq i \leq k$. It is easy to see that by this process each $\tilde{w}^j$ is also adjacent to $\left\{\tilde{v}^1, \ldots, \tilde{v}^{(k-j+1)}\right\}$, $1 \leq j \leq k$. Then for each k-clique $\tilde{K}_v$ in $\tilde{G}$ we add a new node $\tilde{l}_v$ and we connect it to all nodes in $\tilde{K}_v$. In Figure 2.4 the process is shown for a graph G when $k = 2$.

Now let $G = (V, E)$, $|V| = n$, be a graph with a Hamiltonian path, $P = v_1, v_2, \ldots, v_n$. We show $\tilde{G}$ has a spanning k-caterpillar, too. To construct an spanning k-caterpillar for $\tilde{G}$, we first choose $\tilde{K}_{v_1}$ as the base and connect $\tilde{l}_{v_1}$ to all its nodes. Then for each fixed i, $i = 2, \ldots, n$, we choose $\tilde{v}_i^j$, $j = 1, \ldots, k$ from $\tilde{K}_{v_i}$ and we attach it to the nodes $\left\{\tilde{v}_{i-1}^1, \ldots, \tilde{v}_{i-1}^{(k-j+1)}\right\}$ in $\tilde{K}_{v_{i-1}}$. Note that by this process the set $\left\{\tilde{v}_i^1, \ldots, \tilde{v}_i^j, \tilde{v}_{i-1}^1, \ldots, \tilde{v}_{i-1}^{(k-j+1)}\right\}$ forms a $(k+1)$-clique and they are

the nodes of the boundary value set. Then we attach $\tilde{l}_{v_i}$ to $\tilde{K}_{v_i}$ and continue the process for $i+1$, $i < n$.

Now consider a graph G whose corresponding graph $\tilde{G}$ has a spanning k-caterpillar, $\tilde{H}$. The construction of $\tilde{H}$ from a base K imposes an ordering on the nodes of $\tilde{G}$, especially on the k-leaves.

Let $\left\{l_{v_{i_1}}, \ldots, l_{v_{i_n}}\right\}$ be the order of the k-leaves in the construction of $\tilde{H}$. We show that $P = v_{i_1}, \ldots, v_{i_n}$ is a Hamiltonian path of G. Let l_{v_i} and $l_{v_{i+1}}$, $i = 1, \ldots, n-1$ be two consecutive k-leaves and also let $\tilde{K}_{v_i} = N(l_{v_i})$ and $\tilde{K}_{v_{i+1}} = N(l_{v_{i+1}})$ be the attached k-cliques, respectively. Because of our method for constructing $\tilde{G}$ from G, there is at least an edge between $\tilde{K}_{v_i}$ and $\tilde{K}_{v_{i+1}}$.

Since l_{v_i} is just attached to $\tilde{K}_{v_i}$, in ordering of $\tilde{H}$ it appears exactly after the last node of $\tilde{K}_{v_i}$, also it is the same for $l_{v_{i+1}}$ and $\tilde{K}_{v_{i+1}}$. So the nodes of $\tilde{K}_{v_{i+1}}$ should be attached to $\tilde{K}_{v_i}$. □

The previous theorem is not the only bad news about the caterpillar problem. We shall show in the next chapter that it is even hard to find a proper approximation algorithm for the minimum spanning caterpillar problem. That is the reason why we concentrate on parametrized algorithm or finding a theoretical basis for heuristic algorithms such as branch and cut.

Chapter 3

Hardness of Approximation and IP Framework

3.1 Introduction

In this chapter we study the *Minimum Spanning Caterpillar* Problem (MSCP). We remind the reader that by a *caterpillar* we mean a tree that reduces to a path by deleting all its leaves. We refer to the remaining path as the *spine* of the caterpillar. The edges of a caterpillar H can be partitioned into two sets, the spine edges, $\mathcal{S}(H)$, and the leaf edges, $\mathcal{L}(H)$. An instance of the MSCP is denoted by a triple (G, s, l), where $G = (V, E)$ is an undirected graph and $s : E \to \mathbb{N}$ and $l : E \to \mathbb{N}$ are two (cost) functions. For each caterpillar H as a subgraph of G we define the cost of H by

$$c(H) := \sum_{e \in \mathcal{S}(H)} s(e) + \sum_{e' \in \mathcal{L}(H)} l(e').$$

In the MSCP one wants to find a caterpillar in a graph with the minimum cost that contains all nodes; see Figure 3.1. As shown in Chapter 2, it is not hard to prove that the problem of finding a spanning caterpillar in a graph is **NP**-hard.

In some applications, in addition to finding an optimal cost spanning caterpillar, we also need to satisfy some restrictions on the final output. For example one may wish to have a caterpillar whose total spine cost is bounded by a fixed

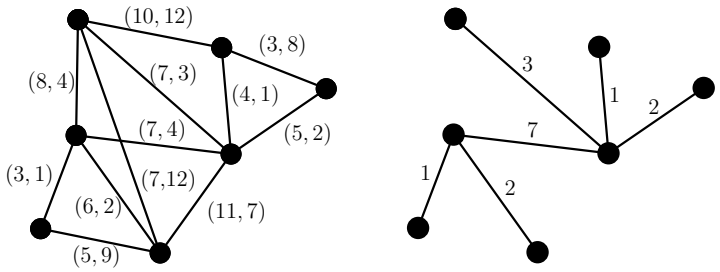

Figure 3.1 A graph and its minimum spanning caterpillar, where each (s, l) label on an edge represents the spine and leaf costs.

number, or one may wish to have an upper bound on the largest degree of a caterpillar.

Simonetti et al. [53, 54] presented a heuristic algorithm that finds a spanning caterpillar by branch and cut method. Their method is based on an Integer Linear Programming (ILP) formulation by transforming the MSCP to the Minimum Steiner Arborescence Problem. In this chapter we present an integer programming formulation for the problem that enables us to solve many restricted versions of the MSCP. In Chapter 4 we present a linear time algorithm for the special case when the input graph is restricted to the bounded treewidth graphs when the associated tree decompositions are given as part of the inputs; see also [26].

The organization of this chapter is as follows. In the next section we give our hardness result for approximating the minimum spanning caterpillar in a graph. In Section 3.3 we present our quadratic integer programming formulation for the problems and some restricted versions of them. We briefly introduce the Gomory cutting method in Section 3.4. In Section 3.5 we present a heuristic algorithm for the MSCP by using the Gomory cutting method iteratively. We also show that for every $\epsilon > 0$, our algorithm guarantees to achieve a $1/\epsilon$ factor approximation. Semidefinite programming gives another alternative for solving the problems, which is the theme of Section 3.6. The results of this chapter have been published in [27].

3.2 Hardness of Approximation

In this section we show that by assuming **P** $\neq$ **NP** it is not possible to find an approximation algorithm for the MSCP within a factor of a polynomial-time computable function of the number of nodes of the graph. Our proof is based on a reduction from the Hamiltonian Path Problem to an instance of the MSCP. We show if there is such an approximation algorithm for the MSCP then there is a polynomial time algorithm for deciding the Hamiltonian Path Problem. It is interesting to note that the same result holds for the hardness of approximation for the travelling salesman problem; see [52].

Theorem 2 *Let $f(n)$ be a polynomial time computable function. The MSCP has no approximation algorithm within a factor of $f(n)$, unless **P** = **NP**.*

Proof. Let $G = (V, E)$ be an instance of the Hamiltonian Path Problem with $n = |V| \geq 3$. We reduce it to an instance (G', s, l) of MSCP. Here G' is made from a copy of G with n extra nodes which are connected to the nodes of the copy of G by a one-to-one correspondence and one node v_{ext} which is connected to all nodes of the copy of G and n extra nodes. Then we define two cost functions s and l such that to every edge e that belongs to the copy of G, the function s assigns the value of $s(e) = 1$ and the function l assigns the value of $l(e) = f(n)(n-1)$.

To assign cost functions to the edges incident to v_{ext} we choose a fixed node v_{fix} in our copy of G. Then both functions assign the value 0 to the edge (v_{fix}, v_{ext}), and they assign the value of $f(n)(n-1)$ to the other edges incident to v_{ext}. We add v_{ext} to ensure that graph G' has at least one spanning caterpillar. The edges that connect the n extra nodes to the copy of G get the value 0 by both functions s and l.

Now we show that if one has an $f(n)$-approximation algorithm for the MSCP then it can be used to decide if G has a Hamiltonian path in polynomial time. We first run the approximation algorithm on G', then it will return a solution T that has $c(T) \leq f(n)\text{OPT}$, where OPT is the overall cost of a minimum spanning caterpillar in G'. Now if G has a Hamiltonian path then $\text{OPT} = n-1$ and $c(T) \leq f(n)(n-1)$. Note that in this case v_{ext} is attached to the Hamiltonian path in the copy of G by (v_{fix}, v_{ext}).

If G has no Hamiltonian path then either it has at least one leaf edge from the copy of G, or it uses two incident edges of v_{ext}, so we have

$$f(n)(n-1) < \text{OPT} \leq c(T).$$

□

Theorems 2 justifies our effort for the rest of this chapter to concentrate on finding heuristic algorithms for solving the MSCP. Since, assuming $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial-time approximation algorithm for the problem.

3.3 Integer Programming Formulation

In this section we introduce an integer quadratic programming formulation for the MSCP. As we shall show later, by adding proper constraints, we can use this formulation for solving some restricted versions of the problem. Though our basic formulation in not linear, it is known that any integer quadratic programming problem can be transformed to a linear one by adding extra variables and constraints in a mechanical way; see Chapter 2 and also [60]. Having an integer linear programming formulation is the first step in applying some well known heuristic algorithms for solving an optimization problem.

We first explain the formulation for the MSCP. Note that without loss of generality we can assume that all instances of the MSCP are complete graphs. Our formulation for the MSCP somehow resembles our formulation for the largest caterpillar problem; that is the subject of Chapter 5.

Let $G = (V, E)$ be a graph and let (G, s, l) be an instance of the MSCP. We convert G to a directed graph $H = (N, A)$, $N = V$, by replacing each edge $e \in E$ with a pair of anti-parallel arcs, $f, f^- \in A$ (with opposite directions). In this case, we can consider a caterpillar in H as a rooted tree that has a directed path as its spine; see Figure 3.2. Here each spine node has one incoming arc and one or more outgoing arcs and every leaf node has one incoming arc and no outgoing arc.

For each node $v \in N$, we denote by $in(v)$ and $out(v)$ the incoming and the outgoing arcs of v, respectively. Also for each $f \in A$ we denote by f^- the anti-parallel arc associated with f. The costs of anti-parallel arcs are the same as the

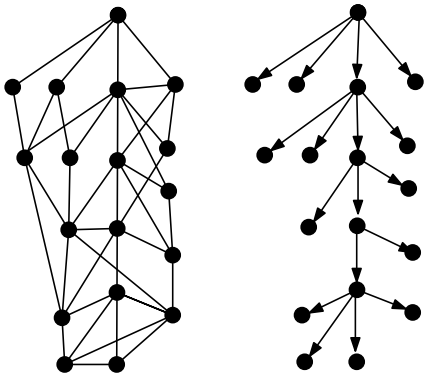

Figure 3.2 A graph and its rooted spanning caterpillar.

cost of their corresponding (undirected) edges, which also depend on their role as spine or leaf arcs. For each arc $f \in A$ we denote its spine cost $s(f)$ by s_f and its leaf cost $l(f)$ by l_f.

We now present our integer programming formulation for the MSCP with a fixed root r. To each arc $f \in A$ we assign two variables, x_f and y_f. In each solution a variable x_f is 1 if f is chosen as a leaf edge and a variable y_f is 1 if it is chosen as a spine edge, otherwise they are 0. Also to each node $v \in N$ we assign variable z_v that represents the level of v in the rooted caterpillar. In particular, we have $z_r = 0$. Our goal is to minimize the objective function

$$\left(\sum_{f \in A} l_f x_f + \sum_{f \in A} s_f y_f \right).$$

In what follows M is a large positive integer, say, a number greater than the order of the graph. The integer programming constraints are as follows.

$$x_f + y_f + x_{f^-} + y_{f^-} \leq 1, \quad f \in A \tag{3.1}$$

$$\sum_{f \in in(v)} y_f + \sum_{f \in in(v)} x_f = 1, \quad v \in N \setminus r \tag{3.2}$$

$$\sum_{f \in out(v)} y_f + \sum_{f \in in(v)} x_f = 1, \quad v \in N \tag{3.3}$$

$$\sum_{f \in out(v)} x_f - M \left(\sum_{f \in in(v)} y_f + \sum_{f \in out(v)} y_f \right) \leq 0, \quad v \in N \tag{3.4}$$

$$\sum_{f=(u,v)\in in(v)} y_f(z_v - z_u) + \sum_{f=(u,v)\in in(v)} x_f(z_v - z_u) = 1, \quad v \in N \setminus r \tag{3.5}$$

$$z_r = 0, \tag{3.6}$$

$$f \in A, \tag{3.7}$$

$$x_f, y_f \in \{0,1\}$$

$$v \in N \tag{3.8}$$

$$z_v \in \{0,\ldots,n-1\}.$$

The next theorem shows that each integral feasible solution, that satisfies these constraints, represents a spanning caterpillar.

Theorem 3 *Constraints 3.1-3.8 make a valid formulation for the MSCP.*

Proof. The first constraint shows that for any pair of anti-parallel arcs that have the same end nodes, only one may be chosen in a feasible solution. The second and the third constraints say that each node on the spine has at most one incoming spine arc and one outgoing spine arc while it has no incoming leaf arc. Constraint 3.4 says that if a node is chosen as a leaf then it has no outgoing leaf arc, but if it is chosen as a spine node then there is no restriction on the number of its outgoing leaf arcs. Constraint 3.5 alongside Constraint 3.6 guarantee that each feasible solution is a connected tree that is rooted at r. Constrains 3.7 and 3.8 enforce the integrality of a feasible solution.

On the other hand, let T be a spanning caterpillar of a graph G. Also let e be an edge of T, if e is a spine edge then we assign $y_f = 1$ and $x_f = 0$, if e is a leaf edge then we set $y_f = 0$ and $x_f = 1$. For any other edge f of the graph that does not belong to T, we have $x_f = y_f = 0$. Now it is easily seen that these values make a feasible solution for our formulation. □

When in our integer programming formulation we replace the Constraints 3.7 and 3.8 on the integrality of variables by weaker constraints that for all $f \in A$, $x_f \geq 0$ and $y_f \geq 0$ and also for all $v \in N$, $z_v \geq 0$, then we say that we have a *relaxation* of the integer programming problem. While solving an integer

linear programming problem is **NP**-hard, its relaxation (a linear programming problem) can be solved in linear time.

Now we consider more constraints to formulate the restricted versions of the problems. The first one is when we have a restriction on the number of spine edges, as an upper bound U. We can impose this by adding the following constraint

$$\sum_{f \in A} y_f \quad \leq U. \tag{3.9}$$

If the restriction is on the overall cost of the spine then we have

$$\sum_{f \in A} s_f y_f \quad \leq U. \tag{3.10}$$

Also the following set of inequalities restricts the degree of each node to be a value no more than a fixed $\delta > 0$.

$$\sum_{f \in in(v) \cup out(v)} (y_f + x_f) \quad \leq \delta, \qquad \forall v \in A. \tag{3.11}$$

In the rest of this chapter and with respect to our result on the hardness of approximation of the MSCP, we will concentrate on finding a heuristic algorithm for the problem.

3.4 Gomory Cutting Method

In this section we briefly introduce the Gomory cutting method for solving integer and mixed integer linear programs. This method was introduced by Gomory in [34]. Recently there have been more studies on the practical improvement of this technique; see [5, 18, 20]. It is proven that Gomory cuts converge to optimal solution but it requires exponential time in worst case, see [19, 48]. Our motivation to suggest Gomory cut for finding a near optimal solution for MSCP is the result of Balas et al. [5]. They show that Gomory cutting method performance improved significantly if it is combined with branch-and-cut method.

To apply the Gomory cutting method we first solve the relaxation of the

problem via the simplex method. Then we use the entries of the resulted table to add a new constraint to the problem. By adding the new constraint we exclude some elements from the feasible solutions while keeping all integer feasible solutions. We may continue this process until an integral optimal solution is found. In what follows we explain this method with more details. Our introduction to the Gomory cutting method is based on Papadimitriou and Steiglitz [46].

Suppose we are given an instance of an integer linear programming problem. Assume we have a solution for the relaxation using the simplex algorithm, that produces an optimal basic feasible solution x associated with basis $\mathcal{B}$. As an equation in the final tableau we have

$$x_{B(i)} + \sum_{j \notin B} y_{ij} x_j = y_{i0} \tag{3.12}$$

for some i, $0 \leq i \leq m$ (if z is the cost then we have $x_{B(0)} = -z$).

Since the variables x_j in Equation 3.12 are nonnegative, then

$$\sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \leq \sum_{j \notin B} y_{ij} x_j, \tag{3.13}$$

So Equation 3.12 becomes

$$x_{B(i)} + \sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \leq y_{i0} \tag{3.14}$$

Since x is constrained to be an integer in an ILP, then the left-hand side of Equation 3.14 is integer. We can replace the right-hand side of Equation 3.14 by its integer part without disturbing the relation, so we have

$$x_{B(i)} + \sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \leq \lfloor y_{i0} \rfloor, \tag{3.15}$$

subtracting Equation 3.15 from Equation 3.12 gives

$$\sum_{j \notin B} (y_{ij} - \lfloor y_{ij} \rfloor) x_j \geq y_{i0} - \lfloor y_{i0} \rfloor. \tag{3.16}$$

Let

$$f_{ij} = y_{ij} - \lfloor y_{ij} \rfloor, \tag{3.17}$$

we refer to the number f_{ij} as the fractional part of y_{ij}, which satisfies

$$0 \leq f_{ij} < 1,$$

finally, we get the following constraint

$$\sum_{j \notin B} f_{ij} x_j \geq f_{i0} \tag{3.18}$$

which is called the *Gomory cut* corresponding to row i. Finally we add the new constraint to the set of constraint and solve the relaxation of the new integer linear programming. We can follow this procedure until reaching an integral optimal solution.

3.5 An $\left(\frac{1}{\epsilon}\right)$-Approximation for the MSCP

It is known that each quadratic programming problem with bounded variables can be transformed to a linear programming (LP) formulation in a mechanical way. The only price is to introduce more variables and constraints. We refer the reader to Chapter 2 and [60]. So in this section we assume that we have an integer linear programming formulation (ILP) for the MSCP. By this assumption, we can solve the relaxation by the simplex method, and then we will apply Gomory cuts iteratively to find an approximation solution to the problem.

A Gomory cutting method adds a linear constraint to the set of constraints of an integer linear programming problem such that it does not exclude any optimal integer solution. The process is repeated until an integer optimal solution is found. It is proven that the Gomory cutting method always terminates with an integer solution that is optimal. Note that here the number of steps required may be exponential. We use the Gomory cutting method to present a heuristic algorithm for the MSCP. The outline of the algorithm is as follows. Note that in the algorithm we assume that we have access to the output of the simplex algorithm, since in each iteration we need it to construct a Gomory cut.

Algorithm: $\left(\frac{1}{\epsilon}\right)$-Approximation via Gomory cut

Input: Graph (G, s, l) **and** $\epsilon > 0$

Output: A Caterpillar T

1. Construct the integer programming formulation.
2. Find an optimal solution, OPT, for the relaxation by the simplex algorithm.
3. Find a directed path starting from a fixed root r such that the y values of the arcs of the path are not less than ϵ, where $0 < \epsilon < 1$. To this end choose an outgoing arc from r with y value at least ϵ. Then follow this process until reaching a node that has no outgoing arc with that property.
4. If all other nodes are attached to this spine with x values that are greater or equal to ϵ, then the resulting caterpillar has a cost less than $(\frac{1}{\epsilon})$OPT and stop.
   Else use the Gomory cutting plane method to add one more constraint.
5. Solve the resulting linear programming problem and go to Step 3.

The convergence of Gomory cuts guarantees that we eventually reach the desired approximation within a finite number of iterations. What follows is our formal justification.

Theorem 4 *There is an algorithm that for any* ϵ, $0 < \epsilon < 1$, *computes a* $1/\epsilon$ *factor approximation for the MSCP.*

Proof. Here we prove that by following the algorithm mentioned above, we eventually reach to the desired approximation. Let S_A be the cost of the final caterpillar that spans the graph and has arcs with costs at least ϵ.

First, note that if for a node $v \in N$ there is an arc $f \in in(v)$, such that $x_f \geq \epsilon > 0$, then we have

$$1 - x_f \leq x_f \left(\frac{1-\epsilon}{\epsilon}\right).$$

Also if there is an arc f such that $y_f \geq \epsilon > 0$ we have

$$1 - y_f \leq y_f \left(\frac{1-\epsilon}{\epsilon}\right).$$

By using these inequalities for the edges of a caterpillar we have

$$\sum_{f \in L} l_f(1 - x_f) + \sum_{f \in S} s_f(1 - y_f) \leq \left(\frac{1-\epsilon}{\epsilon}\right) \left(\sum_{f \in L} l_f x_f + \sum_{f \in S} s_f y_f\right),$$

where L is the set of leaf edges and S is the set of the spine edges of the caterpillar.

Now by using this information we show that the resulting caterpillar has a cost that is less than $\frac{1}{\epsilon}$OPT. First of all we have

$$S_A = \sum_{f \in L} l_f + \sum_{f \in S} s_f.$$

By rewriting the right-hand side summations we have

$$\begin{aligned}
\sum_{f \in L} l_f + \sum_{f \in S} s_f &= \sum_{f \in L} (x_f + (1 - x_f)) l_f + \sum_{f \in S} (y_f + (1 - y_f)) s_f \\
&= \left(\sum_{f \in L} l_f x_f + \sum_{f \in S} s_f y_f\right) + \left(\sum_{f \in L} l_f(1 - x_f) + \sum_{f \in S} s_f(1 - y_f)\right) \\
&\leq \left(1 + \frac{1-\epsilon}{\epsilon}\right) \left(\sum_{f \in L} l_f x_f + \sum_{f \in S} s_f y_f\right) \\
&\leq \frac{1}{\epsilon}\text{OPT}.
\end{aligned}$$

□

There are many software tools for solving LP problems and this is an advantage for our method. On the other side, using Gomory cuts has its own drawback, since there is no guarantee on the rate of convergence to the optimal (integral) solution. Also, converting the quadratic integer programming formulation to an LP one, introduces many new variables and constraints that increase the size of the input.

In the next section we show that the MSCP can be considered as a semidefinite programming problem. So we do not need to introduce many new variables and constraints. This point of view gives another alternative for obtaining a heuristic algorithm.

3.6 Semidefinite Programming Transformation

A *semidefinite programming* problem is the problem of optimization of a linear function of a symmetric and positive semidefinite matrix subject to linear equality constraints; for a precise definition see [1]. One can solve a semidefinite programming problem within a constant approximation factor.

Semidefinite programming has been applied to solve some other problems in combinatorial optimization. For example [33] obtained a 0.878 factor randomized approximation algorithm for the Maximum Cut Problem by using this method.

As we show in the former section the MSCP can be formulated as a quadratic integer program. Now we convert the problem to a vector program. In a *vector program* the objective function and all constraints are represented as linear combinations of inner products of vectors. It is known that each vector program is equivalent to a semidefinite program; see [33] and [59].

Theorem 5 *The quadratic formulation of the MSCP has a semidefinite equivalent formulation.*

Proof. We assign vectors $\mathbf{v}_e^x = (x_e, 0, \ldots, 0), \mathbf{v}_e^y = (y_e, 0, \ldots, 0)$, and $\mathbf{v}_u^z = (z_v, 0, \ldots, 0)$ to variables x_e, y_e, and z_e, respectively. We also have an extra vector variable $\mathbf{v}_t = (1, \ldots, 1, t)$. Now we write the objective function and all constraints in the former section by linear combinations of inner products of these

vector variables.

Finally we want to minimize

$$\left(\sum_{f \in A} l_f(\mathbf{v}_f^x \cdot \mathbf{v}_t) + \sum_{f \in A} s_f(\mathbf{v}_f^y \cdot \mathbf{v}_t) \right),$$

subject to the constraints in the following constraints.

$$\begin{aligned}
\mathbf{v}_f^x \cdot \mathbf{v}_t + \mathbf{v}_f^y \cdot \mathbf{v}_t + \mathbf{v}_{f^-}^x \cdot \mathbf{v}_t + \mathbf{v}_{f^-}^y \cdot \mathbf{v}_t &\leq 1, \quad f \in A && (3.19)\\
\sum_{f \in in(v)} \mathbf{v}_f^y \cdot \mathbf{v}_t &\leq 1, \quad v \in N && (3.20)\\
\sum_{f \in out(v)} \mathbf{v}_f^y \cdot \mathbf{v}_t &\leq 1, \quad v \in N && (3.21)\\
\sum_{f \in in(v)} \mathbf{v}_f^x \cdot \mathbf{v}_t + \sum_{f \in in(v)} \mathbf{v}_f^y \cdot \mathbf{v}_t &\leq 1, \quad v \in N && (3.22)\\
\sum_{f \in in(v)} \mathbf{v}_f^x \cdot \mathbf{v}_t + \sum_{f \in out(v)} \mathbf{v}_f^y \cdot \mathbf{v}_t &\leq 1, \quad v \in N && (3.23)\\
\sum_{f \in out(v)} \mathbf{v}_f^x \cdot \mathbf{v}_t - M \left(\sum_{f \in in(v)} \mathbf{v}_f^y \cdot \mathbf{v}_t + \sum_{f \in out(v)} \mathbf{v}_f^y \cdot \mathbf{v}_t \right) &\leq 0, \quad v \in N && (3.24)\\
\sum_{f=(u,v) \in in(v)} \mathbf{v}_f^y \cdot (\mathbf{v}_v^z - \mathbf{u}_v^z) + \sum_{f=(u,v) \in in(v)} \mathbf{v}_f^x \cdot (\mathbf{v}_u^z - \mathbf{u}_v^z) &= 1, \quad v \in N \backslash r && (3.25)\\
\mathbf{v}_r^z \cdot \mathbf{v}_t &= 0. && (3.26)
\end{aligned}$$

The validity of the assertion follows from the fact that any vector programming problem is equivalent to a semidefinite programming problem. □

3.7 Summary and Open Problems

We introduced the Minimum Spanning Caterpillar Problem. We showed that the problem has no approximation algorithm with a polynomial time computable function as an approximation factor, unless $\mathbf{P} = \mathbf{NP}$. By this result we end the search for any *proper* approximation algorithm. Then we introduced our quadratic integer programming formulation for the problem and we gave some arguments on its strength and weakness for finding heuristic algorithms, by using Gomory cuts or semidefinite programming.

There are some natural open problems left, such as:

1. We know that the Traveling Salesman Problem (TSP) also has the same property with respect to having no approximation algorithm within a polynomial time computable factor, while its metric version has a constant factor approximation. Now the question that arises is: "Can one find a constant factor approximation algorithm for a metric version of the MSCP?"

2. When compared to our method of using Gomory cuts, does a branch and cut algorithm behave better in practice?

Chapter 4

Spanning Caterpillars in Bounded Treewidth Graphs

Once more we remind the reader that by a caterpillar we mean a tree that reduces to a path by deleting all its leaves. We refer to the remaining path as the *spine* of the caterpillar. The edges of a caterpillar H can be partitioned to two sets, the spine edges, $\mathcal{B}(H)$, and the leaf edges, $\mathcal{L}(H)$. Let $G = (V, E)$ be a graph. Also let $b : E \to \mathbb{N}$ and $l : E \to \mathbb{N}$ be two (cost) functions. For each caterpillar H as a subgraph of G we define the cost of H by

$$c(H) := \sum_{e \in \mathcal{B}(H)} b(e) + \sum_{e' \in \mathcal{L}(H)} l(e').$$

As mentioned in Chapter 3 in the Minimum Spanning Caterpillar Problem (MSCP) [53] we want to find a caterpillar with the minimum cost that contains all nodes. We know from a theorem in Chapter 2 that MSCP is **NP**-complete for general graphs; see also [57]. In this chapter we consider the problem when the input is restricted to bounded treewidth graphs. That is, we assume that a tree decomposition with a fixed width of the graph is given as part of the input.

One application of this problem is to find a cost effective subnetwork within a large network. For example, one may want to find a linearly connected backbone to place routers with computers attached. Here each cost function represents a different technology that is used to connect nodes on the backbone (that consists of routers or hubs) and the leaves (i.e. computers) to the backbone.

As another application of the MSCP, Tan and Zhang [57] used it to solve some problems concerning the Consecutive Ones problem.

It is known that graph structures that are expressible by Monadic Second Order Logic (MSOL) [21,22] are recognizable in linear time on bounded treewidth graphs. The same is true for those optimization problems that can be defined by Extended Monadic Second Order Logic (EMSOL); see the survey of Hlineny *et al.* [41]. The main disadvantage of these methods is that they are very hard to implement, even for small values of k, where k is the treewidth of a graph. In the next section we introduce a formula that expresses spanning caterpillar problem in the language of monadic second order logic.

The majority of this chapter is devoted to presenting an easy-to-implement and linear-time algorithm for minimum spanning caterpillar problem in a graph with bounded treewidth. Although the hidden constant factor in the asymptotic notation of the algorithm grows very fast, but for some graphs that appear in applications it is still practical, like outerplanar, series-parallel (K_4 minor-free), and Halin graphs. Refer to Chapter 2 for definitions of those classes of graphs.

Also we explain briefly how one can tackle two other related **NP**-hard problems: the *Minimum Spanning Ring Star* Problem (MSRSP) [6,44] where the goal is to find a minimum spanning subgraph (star ring) that consists of a cycle and nodes of degree one that are connected to it, and the *Dual Cost Minimum Spanning Tree* Problem (DCMSP), where the cost of an edge incident to a leaf is different from the other edges. As far as we know it is the first time that these problems are studied from this point of view.

In Section 4.2 we shall present the required definitions formally. In particular, we introduce the k-parse data structure as an alternative to the smooth tree decomposition. In Section 4.3 we give a dynamic programming algorithm that solves the MSCP in linear time for graphs that their tree decompositions are also given as inputs. The proof of the correctness of the algorithm is presented in Section 4.4. Then we illustrate our algorithm by an example in Section 4.5. We then discuss how our algorithm can be applied to solve two related problems. The chapter ends with a conclusion and some open problems.

4.1 Monadic Second Order Logic Formula

In this section we introduce the Monadic Second Order Logic (MSO) expression of the spanning caterpillar problem. By the following theorem Courcelle [22] shows that all problems that are definable in MSO Logic can be solved in polynomial time on the class of graphs with bounded tree-width.

Theorem 6 ([22]) *Let P be a graph problem that has an MSO sentence ϕ. Also let $G = (V, E)$ be a graph of treewidth at most k where k is a fixed value. Then there is an Algorithm that solves P on input G in time $f(\|\phi\|, w) \cdot n \in O(n)$, where $\|\phi\|$ denotes the length of the MSO formula ϕ.*

Arnborg, Lagergren and Sees [3] extended the Courcelle's result by introducing Extended Monadic Second Order (EMSO) logic. They show that optimization problems which are definable as an EMSO formula have a polynomial solution on graphs of bounded treewidth.

There are properties that cannot be formulated by the first order logic, but they can be expressed in MSO logic. For example, *connectivity* of a graph can be formulated in MSO logic while it is not possible to express it as a first order sentence. So in that sense, we can say that MSO logic is more powerful than the first order logic.

Let us first define the first order formulas. For a more comprehensive introduction to logic refer to Enderton [30]. Formulas in the first order logic are made from

1. a set of variables,
2. a set of constant symbols,
3. functional symbols (with arities),
4. relational symbols (with arities),
5. two logical quantifiers $\forall$ and $\exists$, and
6. the set of connective symbols $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$.

The set of *terms* consists of expressions that are made from constants and variable symbols by applying (zero or more times) of functions. Note that this definition also consider all constants and variable symbols as terms.

Now we define atomic formulas by using terms and relations. An *atomic formula* is an expression $R(t_1, \ldots, t_n)$, where R is an n-ary relation and $t_1, \ldots, t_n$ are terms. Finally we build the set of formulas by using the connective symbols $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$ and the quantifier symbols.

The language of the first order logic consists of formulas in which quantifiers are applied to variables. In the second order logic we define two more sets of variables, function variables and relation variables.

In an MSO formula the quantifiers are applied to variables or the set of variables. Here we are interested in the MSO logic that is defined on graphs, where the set variables are defined over nodes and edges of a graph. An MSO formula in the language of graphs is made from atomic formulas $E(x, y)$ and the expression $x = y$ by using the connective and the quantifier symbols. The atomic formula $E(x, y)$ denotes adjacency of x and y. The formula $x = y$ expresses equality relation between x and y. Also the formula $X(x)$ denotes that variable x (a node in a graph) belongs to X (a subset of the set of nodes). In what follows we use lowercase letters such as x, y, z to indicate individual variables and capital letters like X, Y, Z to denote set variables.

Note that a caterpillar is a tree in which there is a path P such that each node v, either belongs to P or is attached to P. Also we define a tree as a connected graph that has no cycle, see [36]. Now we are ready to express the existence of a caterpillar in MSO logic. To reduce the length of formulas we use the following conventions for using quantifiers and nodes set variables and edge set variables (the same conventions are applied when replacing $\forall$ by $\exists$):

1. instead of $\forall x(X(x))$ we write $\forall x \in X$,
2. we denote $\forall x X(x) \wedge \forall y X(y)$ by $\forall x, y \in X$,
3. we show $\forall Y(Y \subseteq X)$ by $\forall Y \subseteq X$.
4. we write $E(x, y) \in F$ to show $E(x, y) \wedge \{x, y\} \in F$

In the following formulas we assume that X is a nodes set variable and F is an edges set variable. Also we assume that the subgraph relation is valid for of

(X, F), say $sub(X, F) = \forall E(x, y) \in F(x \in X \wedge y \in X)$ is always true.

The MSO formula that expresses that (X, F) is a connected subgraph

$$\alpha(X, F) = \forall Y \subseteq X\Big((\exists x \in Y) \wedge (\forall x, y \in X(x \in Y \wedge E(x, y) \in F) \rightarrow y \in Y) \rightarrow \forall x \in Y)\Big),$$

the subgraph (X, F) is acyclic,

$$\beta(X, F) = \neg\exists Y \subseteq X\Big(\exists x \in Y \wedge \forall x\big(x \in Y \rightarrow \exists y_1 \in X \exists y_2 \in X(y_1 \neq y_2 \wedge E(x, y_1) \in F \wedge E(x, y_2) \in F \wedge y_1 \in Y \wedge y_2 \in Y)\big)\Big),$$

node x has degree one in subgraph (X, F)

$$\gamma(X, F, x) = \exists y \in X\Big(E(x, y) \in F \wedge \big(\forall z \in X(E(x, z) \in F \rightarrow y = z)\big)\Big),$$

node x has degree two in (X, F)

$$\theta(X, F, x) = \exists y, z \in X\Big((y \neq z) \wedge E(x, y) \in F \wedge E(x, z) \in F \wedge \Big(\forall t \in X\big(E(x, t) \in F \rightarrow (t = y \vee t = z)\big)\Big)\Big),$$

subgraph (X, F) represent a path in a graph

$$\eta(X, F) = \alpha(X, F) \wedge \beta(X, F) \wedge \Big(\forall x \in X\big(\gamma(X, F, x) \vee \theta(X, F, x)\big)\Big).$$

Now we have enough materials to express that subgraph (X, F) is a caterpillar.

$$cat(X, F) = \alpha(X, F) \wedge \beta(X, F) \wedge \Big(\exists Y \subseteq X \wedge \exists H \subseteq F \wedge \eta(Y, H) \wedge \big(\forall x \in X(\exists y \in Y \wedge E(x, y) \in F)\big)\Big).$$

Finally to say that a graph has a spanning caterpillar we use the following formula

$$spanCat = \exists X \exists F \forall x\big(x \in X \wedge cat(X, F)\big).$$

Note that the same formulation is applicable for the minimum spanning caterpillar problem if the edge weights are chosen from a set with finite number of elements. Such a condition is very restrictive and that is why in the rest of this chapter we focus on a more direct approach for solving the MSCP in bounded tree width graphs.

4.2 k-Parse Representation

In this section we introduce k-parse data structure as a linear and detailed representation of a smooth tree decomposition. It enables us to follow explicitly the process of adding edges that is not clearly expressed in a smooth tree decomposition. That makes the presentation easier and the implementation more straightforward. The definition of k-parse in this section is taken from Dinneen [25].

A $(k+1)$-*boundaried graph* [28] is a pair (G, ∂) of a graph $G = (V, E)$ and an injective function ∂ from $\{0, \ldots, k\}$ to V. We refer to the image of ∂ as the set of *boundaried node*s and denote it by $Im(\partial)$ or by ∂; for the sake of brevity.

Given a path decomposition of width k of a graph, one can represent the graph by using strings of (unary) operators from the following *operator set* $\Sigma_k = V_k \cup E_k$ where

$$V_k = \{ \textcircled{0}, \ldots, \textcircled{k} \}$$

is the set of node operators. Each node operator $\textcircled{i}$, $i = 0, \ldots, k$, adds a boundary node i to the graph. We define the set of edge operators by

$$E_k = \{ \boxed{i\ j} \mid 0 \leq i < j \leq k \},$$

where each $\boxed{i\ j}$ represents the process of connecting the boundary nodes i and j by an edge.

We need one additional (binary) operator to generate a graph from a smooth tree decomposition of width k. We refer to this binary operator as *circle plus* and we denote it by $\oplus$. The operands of $\oplus$ are two boundary graphs that have the same number of boundary nodes. The $\oplus$ operator *glues* its operands via their boundary nodes that have the same labels.

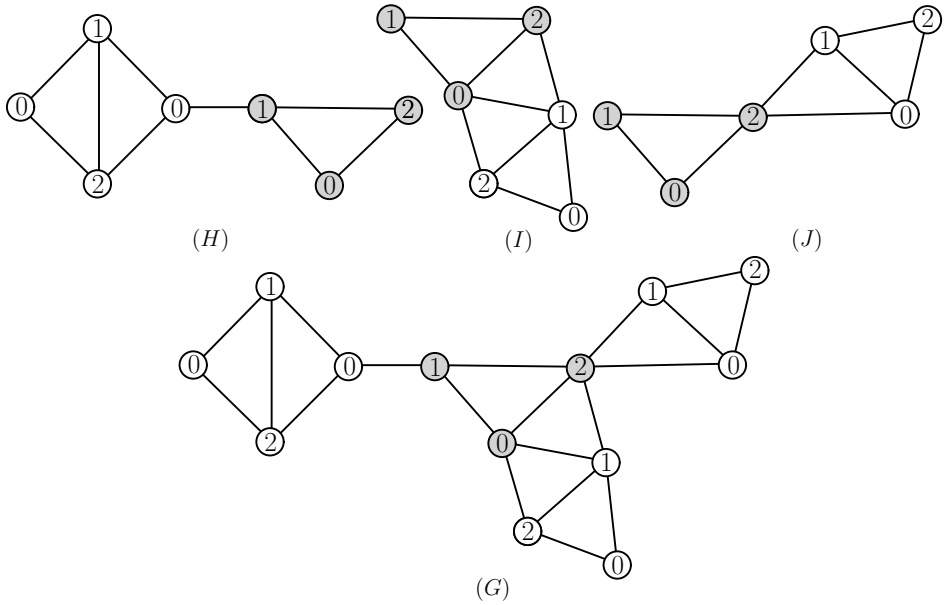

Figure 4.1 $G = (H \oplus I) \oplus J$

The semantics of these operators on $(k+1)$-boundaried graphs G and H are as follows:

$G\,\textcircled{i}$	Add an isolated node to the graph G, and label it as the new boundary node i.
$G\,\boxed{i\ j}$	Add an edge between boundaried nodes i and j of G (ignore if operation causes a multi-edge).
$G \oplus H$	Take the disjoint union of G and H except that equal-labelled boundary nodes of G and H are identified.

It is syntactically incorrect to use the operator $\boxed{i\ j}$ without being preceded by both $\textcircled{i}$ and $\textcircled{j}$, and the operator $\oplus$ must be applied to graphs with the same sized boundaries. A graph described by a string (tree, if $\oplus$ is used) of these operators is called a *k-parse*, and has an implicit labelled boundary ∂ of at most $k+1$ nodes. By convention, a k-parse begins with the *axiom operator string* $[\textcircled{0}, \textcircled{1}, \ldots, \textcircled{k}]$ which represents the edgeless graph of order $k+1$. Throughout this chapter, we refer to a k-parse and the graph it represents interchangeably.

Figure 4.1 shows three 2-paths, H, I and J that their 2-parse representations are

$H = \textcircled{0}, \textcircled{1}, \textcircled{2}, \boxed{0\ 1}, \boxed{0\ 2}, \boxed{1\ 2}, \textcircled{0}, \textcircled{1}, \boxed{0\ 1}, \textcircled{0}, \textcircled{2}, \boxed{0\ 1}, \boxed{0\ 2}, \boxed{1\ 2},$

$I = ⓪, ①, ②, \boxed{0\,1}, \boxed{0\,2}, \boxed{1\,2}, ⓪, \boxed{0\,1}, \boxed{0\,2}, ②, \boxed{1\,2}, ①, \boxed{0\,1}, \boxed{0\,2}, \boxed{1\,2}$,
$J = ⓪, ①, ②, \boxed{0\,1}, \boxed{0\,2}, \boxed{1\,2}, ②, \boxed{0\,2}, \boxed{1\,2}, ⓪, ①, \boxed{0\,2}, \boxed{1\,2}, \boxed{0\,1}$.

Graph $G = (H \oplus I) \oplus J$ is built form H, I, and J. Boundary nodes are shown by gray colour to distinguish them with other non-boundary nodes.

Let $G = (g_0, g_1, \ldots, g_n)$ be a k-parse and $Z = (z_0, z_1, \ldots, z_m)$ be any sequence of operators over Σ_k. The *concatenation* $(\cdot)$ of G and Z is defined as

$$G \cdot Z = (g_0, g_1, \ldots, g_n, z_0, z_1, \ldots, z_m).$$

(For the treewidth case, G and Z are viewed as two connected subtree factors of a parse tree $G \cdot Z$ instead of two parts of a sequence of operators.)

Bodlaender in [11] presented a linear time algorithm that from any tree decomposition of a graph makes a smooth tree decomposition and for each smooth tree decomposition one can easily construct a k-parse representation.

4.3 Algorithm for Bounded Treewidth Graphs

In this section we show that the problem of finding a minimum spanning caterpillar in a bounded treewidth graph with n nodes has an algorithm that runs in linear time, assuming the tree decomposition is given alongside the input graph G. Throughout this section we suppose that the treewidth of G is k and G is represented as a k-parse $G = (g_0, \ldots, g_m)$.

The main idea is to use a forest of at most $k + 1$ different caterpillars as a partial solution, each has at least one node in the boundary set $\partial = \{0, \ldots, k\}$. To this end we *code* the information of each partial solution in a state vector $S = (A, B)$, where A is a $(k + 1)$-tuple $(a_0, \ldots, a_k)$. Each a_i represents a label for the boundary node i from the set $\{E, S, C, I, L\}$. The labels E, S, C, I, and L are characteristics of the boundary nodes in a partial solution. They stand for *expandable, spine, center* (of a star), *isolated node*, and *leaf*, respectively. The set B is a partition set of ∂. If any two boundary nodes belong to the same element of B, then they belong to the same connected component of a partial solution that is represented by B. See Figure 4.2.

In accordance with our dynamic programming approach, we use a table $\mathcal{T}$ that has rows indexed by *state vector*s and columns indexed by k-parse operators

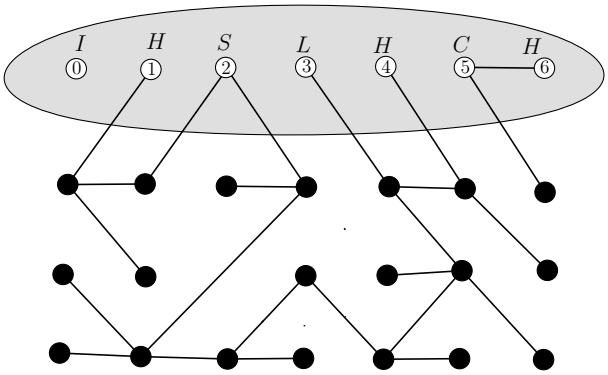

Figure 4.2 A forest of caterpillars

from g_k to g_m. Each entry in a row (A, B) and a column g_i in $\mathcal{T}$ is a pair (X, x), where $X \in \{true, false\}$ shows if the entry represents a valid forest of caterpillars and x is the minimum total cost among the partial solutions represented by the state vector (A, B) at column g_i.

We initialize all entries of $\mathcal{T}$ to the value $(false, \infty)$. Due to our convention for the first $k+1$ operators we have $g_i = \textcircled{i}$, $0 \leq i \leq k$. So at the first step we assign the value $(true, 0)$ to $\mathcal{T}((A, B), g_k)$, where $A = (I, \ldots, I)$ and $B = \{\{0\}, \{1\}, \ldots, \{k\}\}$. Suppose we have computed the entries of $\mathcal{T}$ up to the $(p-1)$th column (which is indexed by g_{p-1} operation). Then we scan the column $p-1$ and look for entries which their first coordinates are *true*. By considering each *true* entry in column $p-1$, we update the entries in column p by the following rules.

Node operator $\textcircled{i}$: Suppose that g_p is a node operator that introduces a new node as the boundary node i. Then for each $\mathcal{T}((A, B), g_{p-1}) = (true, x)$ let $B_i \subseteq \partial$ be the element of B that contains boundary value i. If $B_i - \{i\}$ is empty or if it contains boundary nodes that are labelled just as leaves, then the partial solution becomes disconnected by adding the new boundary node. In such cases we just ignore the entry and move to the next one. Otherwise, we update the value (X', x') of the entry $\mathcal{T}((A', B'), g_p)$, where $B' = (B - \{B_i\}) \cup \{\{B_i - i\}, \{i\}\}$ and A' is the same as A except it has I in its i-th coordinate, by the value of $(true, \min\{x, x'\})$.

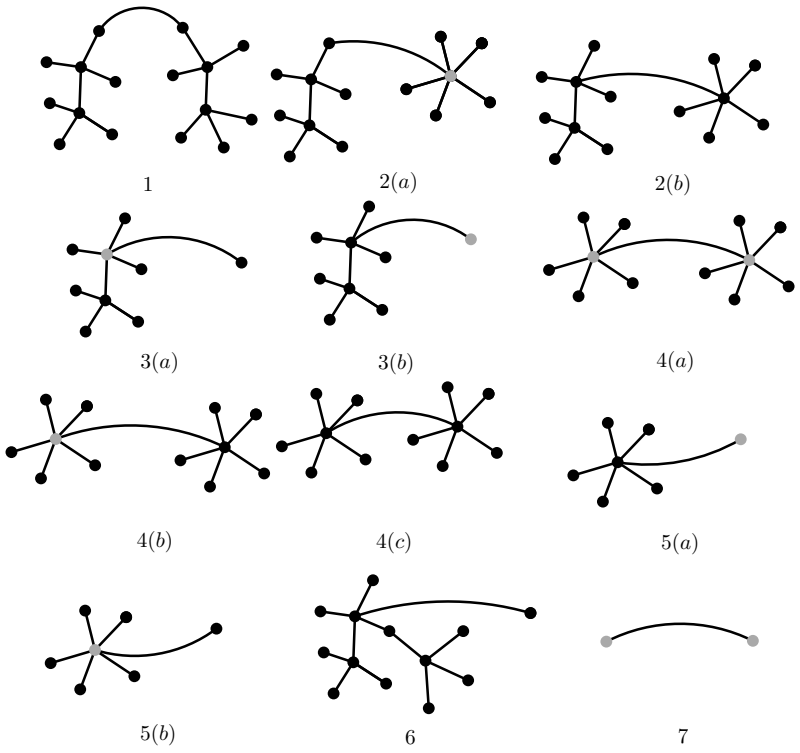

Figure 4.3 The valid combinations of labels in edge operation

Edge operator $\boxed{i\ j}$: If g_p is an edge operator that connects two boundary nodes i and j, then we need to consider two stages. In the first stage we just update the value of each entry $\mathcal{T}((A,B),g_p)$ by the same value as $\mathcal{T}((A,B),g_{p-1})$. This reflects the cases where the edge (i,j) is not used in a partial solution.

Before explaining the next stage we have to mention that only seven different combinations of the labels, when connected together by an edge, make valid partial solutions to our problem. These valid combinations are listed in Table 4.1 and are depicted in Figure 4.3.

Now let $\mathcal{T}((A,B),g_{p-1}) = (true,x)$ and $A = (a_0,\ldots,a_k)$. We find an entry $\mathcal{T}((A',B'),g_p) = (X',x')$ where A' and B' are as follows. The vector A' has a coordinate $(a'_0,\ldots,a'_k)$ where $a'_t = a_t$ if $t \neq i,j$ with the exceptions that (i) if any of i or j has an E label in A that belongs to a star, then we also change the C label of the center of the star in A to an E label in A', and (ii) if any of i or j has a C label that is changed to an E, then the E label in the corresponding star in A is replaced by an L label in A'. The values of a'_i, a'_j are determined by finding a

Table 4.1 Rules for edge operations

Rule	a_i	a_j	a'_i	a'_j	y
1	E	E	S	S	$\min\{x + b(\{i,j\}), x'\}$
2(a)	E	C	S	E	$\min\{x + b(\{i,j\}), x'\}$
2(b)	E	C	S	S	$\min\{x + b(\{i,j\}), x'\}$
3(a)	E	I	E	L	$\min\{x + l(\{i,j\}), x'\}$
3(b)	E	I	S	E	$\min\{x + b(\{i,j\}), x'\}$
4(a)	C	C	E	E	$\min\{x + b(\{i,j\}), x'\}$
4(b)	C	C	E	S	$\min\{x + b(\{i,j\}), x'\}$
4(c)	C	C	S	S	$\min\{x + b(\{i,j\}), x'\}$
5(a)	C	I	S	E	$\min\{x + b(\{i,j\}), x'\}$
5(b)	C	I	C	L	$\min\{x + l(\{i,j\}), x'\}$
6	S	I	S	L	$\min\{x + l(\{i,j\}), x'\}$
7	I	I	C	E	$\min\{x + b(\{i,j\}), x'\}$

Table 4.2 Rules for boundary join operations

Rule	a_i	a'_i	a''_i
1	S	$\{E, I, C\}$	S
2	E	$\{I, C\}$	E
3	E	E	S
4	C	$\{C, I\}$	C

proper match in a row of Table 4.1 with respect to the values of a_i and a_j.

Also $B' = (B \setminus \{B_i, B_j\}) \cup \{B_i \cup B_j\}$, where B_i and B_j are the elements of B that contain i and j, respectively. We update the value of $\mathcal{T}((A', B'), x')$ by $(true, y)$, where y is the corresponding value given in Table 4.1.

Boundary Join Operator $H \oplus H'$: Let g_p be a boundary join operator that unifies the boundary nodes of two k-parses $H = (h_0, \ldots, h_r)$ and $H' = (h'_0, \ldots, h'_s)$, where H and H' are substrings of G. For the sake of simplicity we suppose that they have two different tables, $\mathcal{T}_H$ and $\mathcal{T}_{H'}$. We save the result of the $H \oplus H'$ operation in a new table $\mathcal{T}_\oplus$, whose first-column entries are initialized by $(false, \infty)$.

Let $\mathcal{T}_H((A, B), h_r) = (true, x)$ and $\mathcal{T}_{H'}((A', B'), h'_s) = (true, x')$ be two *true*

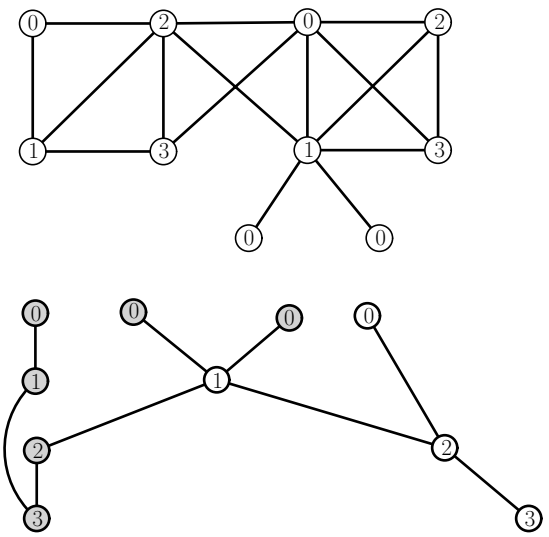

Figure 4.4 A 2-path and its spanning caterpillar

entries in the last columns of $\mathcal{T}_H$ and $\mathcal{T}_{H'}$. If there are two different boundary values i and j that they belong to the same partition (connected components) in both B and B', then this operator creates a cycle which we ignore. In such cases we move to the next possible pair of *true* entries. Otherwise, we find an entry $\mathcal{T}_{\oplus}((A'', B''), \oplus) = (X'', x'')$, where for each coordinate i in $A'' = (a''_0, \ldots, a''_k)$ there is a match for a_i, a'_i, and a''_i in a row of Table 4.2 and

$$B'' = \{B_i \cup B'_i \mid i \in \partial, B_i \in B \text{ and } B'_i \in B'\}.$$

Then we update the value of the entry by $(true, \min\{x + x', x''\})$.

We follow that procedure until we update all entries in the last column. Then among all *true* entries in the last column of $\mathcal{T}$ that has a partition set the same as $\{\partial\}$, any one with the smallest cost is a minimum spanning caterpillar of G.

Before proving the correctness of the algorithm we give a small example of the algorithm on a 4-path graph G that is depicted in Figure 4.4. Graph G has the following 2-parse representation.

$G =$ ⓪, ①, ②, ③, $\boxed{0\,1}$, $\boxed{0\,2}$, $\boxed{1\,2}$, $\boxed{1\,3}$, $\boxed{2\,3}$, ①, $\boxed{1\,2}$, ⓪, $\boxed{0\,1}$, ⓪, $\boxed{0\,1}$, ⓪, $\boxed{0\,1}$, $\boxed{0\,2}$, $\boxed{0\,3}$, ②, ③, $\boxed{0\,2}$, $\boxed{0\,3}$, $\boxed{1\,2}$, $\boxed{1\,3}$, $\boxed{2\,3}$.

We consider a spanning caterpillar of G as shown at the bottom of Figure 4.4. We show the steps that algorithm follows for some of sub-sequences of G. Note that our example corresponds to one row of the dynamic programming table.

$$
\begin{aligned}
⓪, ①, ②, ③ &\rightarrow ((I, I, I, I), \{\{0\}, \{1\}, \{2\}, \{3\}\}) \\
\boxed{0\,1}, \boxed{0\,2}, \boxed{1\,2}, \boxed{1\,3} &\rightarrow ((L, S, I, E), \{\{0, 1, 3\}, \{2\}\}) \\
\boxed{2\,3}, ①, \boxed{1\,2}, ⓪ &\rightarrow ((I, E, S, S), \{\{0\}, \{1, 2, 3\}\}) \\
\boxed{0\,1}, ⓪, \boxed{0\,1}, ⓪ &\rightarrow ((I, E, S, S), \{\{0\}, \{1, 2, 3\}\}) \\
\boxed{0\,1}, \boxed{0\,2}, \boxed{0\,3}, ② &\rightarrow ((I, E, I, S), \{\{0\}, \{1, 3\}, \{2\}\}) \\
③, \boxed{0\,2}, \boxed{0\,3}, \boxed{1\,2}, \boxed{1\,3}, \boxed{2\,3} &\rightarrow ((L, S, E, L), \{\{0, 1, 2, 3\}\}).
\end{aligned}
$$

A more elaborated example is also given is Section 4.5. Now we are ready to show the correctness of our algorithm.

4.4 Correctness of the Algorithm

In this section we justify the correctness of our minimum spanning caterpillar algorithm. We first show that if there is a *true* entry in the last column of a state table, then the graph has a spanning caterpillar. As we mentioned in Section 2.4 we denote the treewidth and the pathwidth of a graph G by $tw(G)$ and $pw(G)$, respectively.

Lemma 7 *Let $G = (g_0, \ldots, g_m)$ be a graph whose $tw(G) = k$ and also let $\mathcal{T}$ be the table produced by the algorithm. If $\mathcal{T}((A, B), g_m)$ has a true entry in the last column of $\mathcal{T}$ such that $B = \partial$, then the graph G has a spanning caterpillar.*

Proof. We show that each true entry in the column p, $p \leq m$, of $\mathcal{T}$ relates to a partial solution that has the following properties:

1. the partial solution is a forest of caterpillars,
2. the partial solution covers all nodes in $(g_0, \ldots, g_p)$.

The conditions are satisfied for the only true entry in the first column of $\mathcal{T}$. Since in the first step we set the element $\mathcal{T}((A,B),g_k)$ to *true*, where $A=(I,\ldots,I)$ and $B=\{\{0\},\{1\},\ldots,\{k\}\}$.

Now suppose that the conditions hold for each true entry in a column $p-1$, $k \leq p-1 < m$. We show that they also hold for each true entry that is produced by the next operation, g_p. The lemma is true when $\mathcal{T}((A',B'),g_p)$ is the resulting *true* entry from $\mathcal{T}((A,B),g_{p-1})$ by a node operation. When g_p is an edge operation Property 2 trivially holds, since no new node is added. Also each rule for an edge operation maintains the first property. The same argument is applicable when g_p is a boundary join operation.

Since Properties 1 and 2 hold for g_m and also since $B=\{\partial\}$, the partial solution corresponds to $\mathcal{T}((A,B),g_m)$ is a spanning caterpillar for G. □

Now we prove that if a graph has a spanning caterpillar, our algorithm can recognize it. We first prove this claim for the bounded pathwidth graphs in Lemma 8, then we give a proof for the bounded treewidth in Lemma 9.

Lemma 8 *Let $G=(g_0,\ldots,g_m)$ be a graph whose $pw(G)=k$. If G that has a spanning caterpillar then the last column of the table $\mathcal{T}$ that results from the algorithm has a* true *entry $\mathcal{T}((A,B),g_m)$ with $B=\{\partial\}$.*

Proof. Without loss of generality we suppose that C is a spanning caterpillar of G such that each leaf of it is attached to a spine node with the smallest index in the k-parse representation. We prove this stronger claim by showing that such spanning caterpillar appears as a partial solution represented by an entry *true* in the last column.

We prove the statement by an inductive argument on the number of nodes of the spine of C. If C has only one node on its spine then it is a star and the node that appears on the center of C takes a unique boundary value in the k-parse, otherwise it fails to attach to all nodes of G. So the center of the star always appears on each boundary and we attach it to a node u when u appears on a boundary by a node operation.

Now assume the lemma is valid for any k-parse that has a spanning caterpillar with less than p nodes on its spine, $p \geq 2$. Let C be a spanning caterpillar of G that has p nodes on its spine. Suppose v is the spine node that is created by

g_f, the node operation that assumes the largest index among all node operators corresponding to spine nodes of C. We delete v and all its leaves in C and if v is not a head we connect its two neighbours on the spine by adding an edge.

The resulting graph H has a caterpillar D with $p-1$ nodes on its spine. We can consider the k-parse representation of H as $(g_0, \ldots, g_{f-1})$, when v is a head, or $(g_0, \ldots, g_{f-1}).g_e$, where g_e is the edge operation corresponds to attaching the neighbours of v on the spine. Because of our inductive assumption, the last column of the table of the algorithm, when applied to H, has a true entry that its partial solution is D. Note that the table of H is the same as the table produced by the algorithm when applied to G in the column $f-1$. If v is not a head in C we just discard the edge operation g_e to allow the neighbours of v on the spine to appear as heads in the partial solution. Now as v stays as a boundary node during $g_f, \ldots, g_m$, the leaves of C can be attached to v by their appropriate edge operation. □

Lemma 9 *Let $G = H \oplus H'$, where H and H' are graphs with treewidths at most k. If G has a spanning caterpillar then the column of the table $\mathcal{T}$, that results from applying the algorithm to G, has a true entry $\mathcal{T}((A, B), H \oplus H')$ with $B = \partial$.*

Proof. If C is a spanning caterpillar in $G = H \oplus H'$, then $H \cap C$ and $H' \cap C$ are forests of caterpillars that span H and H', respectively. To connect the (spanning) forests of caterpillars in H, we first direct the edges on the spine of C from one head to the other. Then we walk along the path on the spine. Once we leave H (by entering to a non boundary node of H') and return to it (by entering to a non-boundary node of H), we add an edge between the two consecutive visited boundary nodes. By the same method we connect components of $H' \cap C$. Note that since we join the connected components via their heads, the resulting graphs are spanning caterpillars of H and H'. We consider the new edges as extensions of the k-parses of H and H'.

Now we apply the algorithm to the extended k-parses of H and H'. By Lemma 8 we know that the last column of each table has a true entry. Since the extensions are done by adding edges, there are also *true* entries on the last columns of the tables associated to k-parse representations of H and H'. In particular, there are *true* entries such that their partial solutions are associated to

$H \cap C$ and $H' \cap C$, so joining them by an $\oplus$ operator produces a *true* entry that has C as its partial solution. □

Theorem 10 *The algorithm solves the spanning caterpillar problem with the running time $O(5^{k+1}B_{k+1}n)$ for a graph of bounded pathwidth k and with the running time $O(5^{k+1}B_{k+1}^2 n)$ for a graph of bounded treewidth k; where n is the number of nodes and B_{k+1} is the $(k+1)$th Bell number (the number of partitions of a set with $k+1$ members).*

Proof. Note that each k-parse G has a representation as (a) $G = G' \cdot H$, or (b) $G = G' \oplus G''$, where G' and G'' each has treewidth at most k and H is a sequence of node and edge operators. The correctness of the algorithm follows from an inductive argument on the number of operators as in (a) and (b). The validity of the base case is the result of Lemmas 8 and 9. For the induction step one just need to use the same technique as Lemma 9 to reduce a problem to the cases with less number of operations. The optimality of the final solution follows from an inductive argument using the fact that we always choose partial solutions with smaller costs.

To solve the problem for a graph with pathwidth at most k, the algorithm uses a table that has $O(5^{k+1}B_{k+1}n)$ entries. In the case when the graph has bounded treewidth, $tw(G) = k$, the algorithm processes each boundary join operation by comparing all pairs of entries in the last two columns of the joined graphs. So it takes $O(5^{k+1}B_{k+1}^2 n)$ steps. □

4.5 Illustrating the Algorithm

In this section we present an example where the algorithm when applied to a graph reveals a spanning caterpillar tree. Here we assume that we have unit costs over edges and our goal is to find a spanning caterpillar. To make following the example easier we show a boundary node with its label together.

In Figure 4.5 we depict a graph G that is the result of the following 2-parse

$$G = (H \oplus H') \cdot (\textcircled{0}, \boxed{0\,1}, \boxed{0\,2}, \textcircled{1}, \boxed{0\,1}).$$

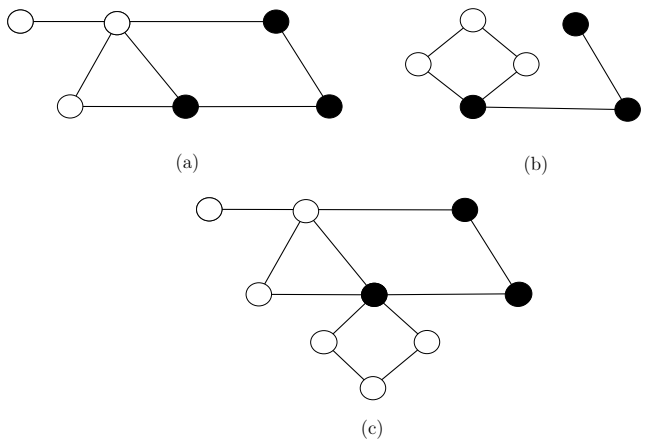

Figure 4.5 A 2-tree.

The subgraphs $H = (h_0, \ldots, h_{12})$ and $H' = (h'_0, \ldots, h'_{11})$ are partial 2-caterpillars and they are represented as 2-parses

$$H = (⓪, ①, ②, \boxed{0\,2}, \boxed{1\,2}, ⓪, \boxed{0\,1}, \boxed{0\,2}, ①, \boxed{1\,2}, ②, \boxed{0\,2}, \boxed{1\,2})$$

and

$$H' = (⓪, ①, ②, \boxed{0\,1}, \boxed{0\,2}, ⓪, \boxed{0\,1}, \boxed{0\,2}, ②, \boxed{0\,2}, ①, \boxed{1\,2}).$$

When the algorithm is applied to H the true entries of the first two columns and the last two columns of the table are given next.

column h_2: $((I, I, I), \{\{0\}, \{1\}, \{2\}\})$,

column h_3: $((E, I, C), \{\{0, 2\}, \{1\}\})$,
$((C, I, E), \{\{0, 2\}, \{1\}\})$, also true entries of the column h_2.

...

column h_{10}: $((E, I, I), \{\{1\}, \{0\}, \{2\}\})$,
$((I, E, I), \{\{0\}, \{1\}, \{2\}\})$,
$((E, E, I), \{\{0, 1\}, \{2\}\})$,
$((E, L, I), \{\{0, 1\}\{2\}\}\})$.

column h_{11}: $((E, I, E), \{\{0, 2\}, \{1\}\})$,

$((S,I,E),\{\{0,2\},\{1\}\})$,
$((E,E,C),\{\{0,2\},\{1\}\})$,
$((C,E,E),\{\{0,2\},\{1\}\})$,
$((E,E,E),\{\{0,1,2\}\})$,
$((S,E,E),\{\{0,1,2\}\})$,
$((E,L,E),\{\{0,1,2\}\})$,
$((S,L,H),\{\{0,1,2\}\})$, also true entries of the column h_{10}.

Finally the last two column for H' are given next.

column h'_9: $((C,I,I),\{\{0\},\{2\},\{1\}\})$,
$((E,I,I),\{\{0\},\{2\},\{1\}\})$,
$((E,I,I),\{\{0\},\{2\},\{1\}\})$,
$((E,I,I),\{\{0\},\{1\},\{2\}\})$,
$((L,I,S),\{\{0\},\{1\},\{2\}\})$,
$((S,I,I),\{\{0\},\{1\},\{2\}\})$.

column h'_{11}: $((H,I,H),\{\{0,2\},\{1\}\})$,
$((S,I,H),\{\{0,2\},\{1\}\})$,
$((S,I,L),\{\{0,2\},\{1\}\})$, also true entries of the column h'_9.

Now we use the rule concerning the boundary join operation to the entry

$$((E,I,I),\{\{1\},\{0\},\{2\}\}),$$

from column h_9 and to the entry

$$((E,I,E),\{\{0,2\},\{1\}\}),$$

from column h'_{11}. The resulting entry is

$$((S,I,E),\{\{0,2\},\{1\}\}).$$

We next apply the 2-parse operators from the extension to this entry (from the first column of the table of $H \oplus H'$) and we have:

1. ⓪ : $((I,I,E),\{\{0\},\{1\},\{2\}\})$,

2. $\boxed{0\,1}$: $((E,C,E),\{\{0,1\},\{2\}\}),((C,E,E),$
   $\{\{0,1\},\{2\}\}),$

3. $\boxed{0\,2}$: $((S,C,S),\{\{0,1,2\}\}),((E,E,S),$
   $\{\{0,1,2\}\}),$

4. ① : $((S,I,S),\{\{0,2\},\{1\}\}),((E,I,S),$
   $\{\{0,2\},\{1\}\}),$

5. $\boxed{0\,1}$: $((S,L,S),\{\{0,1,2\}\}),((E,E,S)\},$
   $\{\{0,1,2\}\}),((S,E,S),\{\{0,1,2\}\}).$

As it is seen from the result of the last operation, the graph G has a spanning caterpillar.

4.6 Two Related Problems

In this section we briefly explain how one can apply the idea of our algorithm to solve other related problems such as the Minimum Spanning Ring Star Problem and the Dual Cost Minimum Spanning tree.

To solve the MSRSP when the input is a graph together with its tree decomposition, we follow almost the same procedure as our algorithm for the MSCP. The main difference is that during an edge operation or a boundary join operation, we allow a cycle to appear. But afterwards, the other nodes can join the cycle just as *leaves*. So as soon as a cycle appears in a partial solution we check to see if the other connected components are isolated nodes. If so, we keep the cycle and we follow the process. Otherwise, we ignore the cycle and move to the next step. Of course here we need to have appropriate bookkeepings to distinguish between state vectors that represent forests of caterpillars and state vectors that represent cycles.

To solve the Dual Cost Minimum Spanning Tree, we ignore the role of spine nodes and we consider forests of trees as partial solutions rather than forests of caterpillars.

4.7 Summary and Open Problems

In this chapter we presented an algorithm that efficiently finds a minimum spanning caterpillar in some classes of graphs that have small treewidth, like outerplanar, series-parallel and Halin graphs. Our algorithm can be easily modified to solve other related network problems like the Minimum Ring Star Problem. Here one just need to allow cycles to appear in the process of the algorithm and keep the required information to see if it eventually spans the graph. We also can use the same idea to solve the Dual Cost Minimum Spanning Tree.

Also if the input consists of a graph and its treewidth (instead of a tree decomposition) then by using any algorithm that computes a tree decomposition, our algorithm gives another proof that MSCP is fixed-parameter tractable [28], rather than using an EMSOL expression.

It would be interesting if one can improve the constant factor in the running time of our algorithm. It is also of interest if one can improve the time complexity by reducing the size of a state table by a tradeoff with accuracy.

Chapter 5

Maximum Caterpillar Problem

As easily seen, not every graph has a spanning caterpillar. So in this chapter we concentrate on the problem of finding a caterpillar with the largest order in a graph. Here we assume that all edges have the same cost and our goal is to find a caterpillar with the largest number of nodes. Since finding a maximum caterpillar is **NP**-hard, here our focus is on finding a heuristic algorithm for this problem.

The organization of this chapter is as follows. In the next section we review the results on finding the largest path in a graph. Our goal is to see how we can use those algorithms for finding large caterpillars. In Section 5.2 we present our integer programming formulation for the Maximum Caterpillar Problem (MCP). We explain how one can solve MCP in directed acyclic graphs in polynomial-time in Section 5.3. In Section 5.4, we use the results of Section 5.2 and 5.3 to introduce our heuristic algorithm for the MCP.

5.1 Longest Paths vs. Maximum Caterpillars

The longest path problem has been the subject of many studies in recent years; see [42] and [10]. Since the set of paths in a graph is a subset of the set of caterpillars, one may wish to use the results about the longest path problem for the maximum caterpillar problem.

For example one may choose a relatively long path as a candidate for the spine of a large caterpillar. The following theorem from Karger, Motwani and

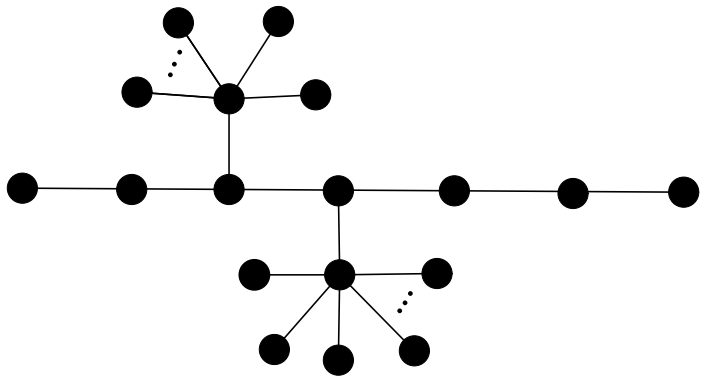

Figure 5.1 A graph with a small longest path and a large maximum caterpillar.

Ramakumar [42] gives the value of $(\lfloor m/n \rfloor + \delta)$ as a lower bound for the maximum caterpillar in a graph.

Theorem 11 ([42]) *Any graph with m edges and n nodes contains a path of length $d = \lfloor m/n \rfloor$.*

As another result on finding a long path in a graph, we mention the elegant method of colour-coding. Colour-coding was proposed by Alon, Yuster and Zwick [2] to find a small path or cycle of order k within a graph. The main idea is to colour randomly every node in a graph with one of k colours and to look for a path in which every node has a different colour.

Alon et al. simplified the colour-coding method for finding a path by giving a random orientation to the edges of a graph $G = (V, E)$. They showed that every simple path of length k has a $2/(k+1)!$ chance of becoming a directed path in the process. To find an actual path, we have to repeat the process $O((k+2)!|V|)$ expected times.

This method is practical when the length of the path is no more than $O(\log n)$. Consequently, the caterpillar constructed by joining the nodes that are adjacent to the path may be small. As seen in Figure 5.1, the longest path in a graph can be much smaller than the maximum caterpillar.

Also one may wish to use the output of any approximation algorithm for the longest path problem as the spine for a caterpillar tree. The 1-neighbourhood of such a path is a caterpillar that is at least as large as the original path.

The negative result on the hardness of approximation for the longest path problem rules out this hope that we can use algorithms for the longest path problem to deal with the maximum caterpillar problem. For example, the following theorem states that finding a path of relatively long length is hard in a Hamiltonian graph.

Theorem 12 ([42]) *For any $\epsilon < 1$, the problem of finding a path of length $n - n^{\epsilon}$ in a Hamiltonian graph is **NP**-complete.*

Also with respect to the following result, we do not expect to find a constant factor approximation algorithm for the longest path problem in general graph.

Theorem 13 ([42]) *There does not exist a constant factor approximation algorithm for the longest path problem, unless $\boldsymbol{P} = \boldsymbol{NP}$.*

The hardness of approximation results are not restricted to undirected graphs. We have the same result when dealing with a directed graph. The following theorem of Björkland, Husfeldt, and Khanna [10] says it is hard to find a constant factor approximation algorithm for Hamiltonian digraphs.

Theorem 14 ([10]) *There can be no deterministic, polynomial time approximation algorithm for Longest Path or Longest Cycle in a Hamiltonian directed graph with performance ratio $n^{1-\epsilon}$ for any fixed ϵ, unless $\boldsymbol{P} = \boldsymbol{NP}$.*

These observations are the main reasons why finding a solution to the maximum caterpillar problem require a different approach.

5.2 Integer Programming Formulation

In this section we present the MCP as an integer programming problem. Let $G = (V, E)$ be a graph. We make a directed graph $H = (N, A)$, where $N = V$ and A is a set that consists of pairs of parallel arcs, $f = (u, v), f^{-} = (v, u)$, for each edge $e = \{u, v\} \in E$.

With respect to those changes, we can consider a caterpillar in H as a rooted tree that has a directed path as its spine; see Figure 3.2, where nodes on the spine have one incoming arc and one or more outgoing arcs and the nodes on leaves

have *just* one incoming arc. For each node $v \in N$, by $in(v)$ and $out(v)$ we denote the incoming and the outgoing arcs of v, respectively. Also for each $f \in A$ we show by f^- the parallel arc with f that has an opposite direction.

Here we first present our integer programming formulation for the MCP with a fixed root r. To each directed edge $f \in A$ we assign two variables, x_f and y_f. In each solution a variable x_f is 1 if f is chosen as a leaf edge and a variable y_f is 1 if it is chosen as a spine edge, otherwise they are 0. Also to each node $v \in N$ we assign the variable z_v. In particular, we have $z_r = 0$. The objective function is

$$\max \left(\sum_{f \in A} (y_f + x_f) \right).$$

In what follows M is a large positive integer; say, a number greater than the order of the graph. The constraints are

$$x_f + y_f + x_{f^-} + y_{f^-} \leq 1, \quad f \in A \tag{5.1}$$

$$\sum_{f \in in(v)} y_f + \sum_{f \in in(v)} x_f \leq 1, \quad v \in N \setminus r \tag{5.2}$$

$$\sum_{f \in out(v)} y_f + \sum_{f \in in(v)} x_f \leq 1, \quad v \in N \tag{5.3}$$

$$\sum_{f \in out(v)} x_f - M \left(\sum_{f \in in(v)} y_f + \sum_{f \in out(v)} y_f \right) \leq 0, \quad v \in N \tag{5.4}$$

$$v \in N \setminus r \tag{5.5}$$

$$\left(\sum_{f \in out(v)} (y_f + x_f) \right) \times$$

$$\left(\left(\sum_{f=(u,v) \in in(v)} (x_f + y_f)(z_v - z_u) \right) - 1 \right) = 0,$$

$$z_r = 0, \tag{5.6}$$

$$\forall f \in A \tag{5.7}$$

$$x_f, y_f \in \{0, 1\},$$

$$\forall v \in N \tag{5.8}$$

$$z_v \in \{0, \ldots, n-1\}.$$

In the next theorem we show that each integral feasible solution, that satisfies these constraints, represents a spanning caterpillar.

Lemma 15 *Every nontrivial feasible solution for the integer programming formulation represents a caterpillar in the graph.*

Proof. Let H be a directed graph consisting of arcs $f \in A$ such that $y_f = 1$ or $x_f = 1$. Since the solution is nontrivial then $E(H) \neq \emptyset$. We show that for each node w of H, there is a directed path from r to w. Let $f = (v, w)$ be the incoming arc to w in H. Note that by Constraint 5.2 such an arc is unique. We first assume that $y_f = 1$.

By Constraint 5.5 we know that

$$\sum_{f=(u,v)\in in(v)} (x_f + y_f)(z_v - z_u) = 1,$$

so there is an arc $g = (u, v) \in in(v)$, such that $y_g = 1$ and $z_v - z_u = 1$. By following this argument we get a sequence of edges

$$f_0 = (v_1, v_0), f_1 = (v_2, v_1), \ldots, f_{k-1} = (v_k, v_{k-1})$$

Since there are just a finite number of nodes we end up in two cases, either $v_k = r$, because r is the only node in the graph that is not restricted by Constraint 5.5, or $v_k = v_0$ and we have a cycle. We show that the latter case is not possible. Let $v_0(= v), v_{k-1}, \ldots, v_1, v_0$ be a cycle. Then we have

$$\begin{aligned} z_{v_0} - z_{v_1} &= 1, \\ z_{v_1} - z_{v_2} &= 1, \\ &\vdots \\ z_{v_k} - z_{v_0} &= 1. \end{aligned}$$

Now if we sum up these equalities we get $0 = k + 1$, which is a contradiction. So there is a directed path from r to every node w of H, if $y_f = 1$ and $f = (v, w)$.

Now we consider the case where $x_f = 1$ and $f = (v, w)$. By Constraint 5.5

we have

$$\sum_{f=(u,v)\in in(v)} (x_f + y_f)(z_v - z_u) = 1,$$

and also with respect of Constraint 5.4 we have

$$1 < \sum_{f\in out(v)} x_f \leq M \left(\sum_{f\in in(v)} y_f + \sum_{f\in out(v)} y_f \right) \tag{5.9}$$

so there is an arc $f \in in(v) \cup out(v)$ such that $y_f = 1$. By using this arc and the fact that there is a directed path from r to f, we can find a directed path from r to w, too.

Now by Constraints 5.2 and 5.3 we know that every node v in H has at most one incoming arc $f = (u, v)$ such that $y_f = 1$. The same is true for outgoing arcs of v. So the set $S = \{f \mid y_f = 1\}$ is a path. As seen from 5.9, every node v that has an incoming arc f such that $x_f = 1$, is attached to a node in S. Also with respect to Constraints 5.2 and 5.3 the node v has just one incoming arc g and one outgoing arc f with $y_f = y_g = 1$. So by Constraint 5.4 we have

$$\sum_{f\in out(v)} x_f = 0.$$

Consequently, in H the node v has no outgoing arc so H is a directed caterpillar rooted at r. □

Lemma 16 *Let H be a directed caterpillar in a graph G that is rooted at r. Then H corresponds to a feasible solution of the integer programming formulation.*

Proof. We make a feasible solution from H by following process. To each arc f that belongs to the spine of H, we assign the value $y_f = 1$. Also the value $x_f = 1$ is assigned to every leaf f in H. We set all other variables to 0. It is not hard to check that all the variables satisfy Constraints 5.1–5.8. □

The next theorem follows from Lemmas 15 and 16.

Theorem 17 *Constraints 5.1-5.8 make a valid formulation for the MCP.*

Now having a valid nonlinear integer programming problem, we can convert it to a linear one; see [60]. So in the following section we assume that a solution for the relaxation of the problem is given.

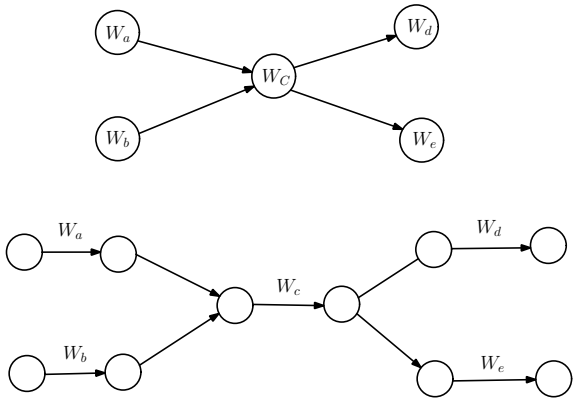

Figure 5.2 The process of transforming a node-weighted DAG to an arc-weighted one.

5.3 Maximum Caterpillar in DAGs

As part of our algorithm to find a caterpillar of large order we need to mention the process of finding a longest path in a Directed Acyclic Graph (DAG). The longest path problem has polynomial-time algorithm on an arc-weighted DAG. For finding a path of longest order in a DAG, one can first change the weights to negative ones and then follows the all pairs shortest path algorithm on the new (directed) graph.

If the DAG is node-weighted, then we need to change it to an arc-weighted one. To this end, we order the nodes by a topological ordering. As the next step we replace each node by a weighted arc. Our rules to connect the neighbours of the node to the new arc are as follows.

If a neighbour has a smaller topological numbering then we connect the end node of the arc representing the smaller node to the start node of the arc representing the larger node. Finally we assign the weight of a node to its representing arc. We also assign zero as the weight of the arcs of the main digraph. Figure 5.2 shows the process.

Finally we can use the algorithm to find a (single source) caterpillar of the largest orders in a DAG. Here we assign weights to the nodes of the digraph with respect to the number of outgoing arcs. Then we apply the algorithm for the node-weighted DAG.

5.4 A Heuristic Algorithm for the MCP

In this section we present a heuristic algorithm for the MCP. Here we assume that a solution to the relaxation of the integer programming formulation is given. We make an acyclic graph $G' = (V', E')$, where $V' = V$ and

$$E' = \{f \mid f = (u, v) \in E \text{ and } z_v > z_u\},$$

Lemma 18 *Let G' be a graph constructed by the aforementioned rule. Then G' has no directed cycle.*

To find a large caterpillar in G', we first assign to each node a weight that is equal to the number of its neighbours in G. Then we find a path P with a largest weight in the node-weighted G'. Since G' is a DAG we can find such a path in G' in polynomial time. By using the path P in G' we can find the spine of a caterpillar in G. The close 1-neighbourhood of that path is the output of our algorithm.

5.5 Summary and Open Problems

In this chapter we introduced the Maximum Caterpillar Problem. We suggested a heuristic algorithm for the problem by using an integer programming formulation. Though we believe that the MCP is as hard as the longest path problem, we are still interested in finding a formal proof for the hardness of approximation of the MCP.

Chapter 6

Miscellaneous Algorithms

One way to face the hardness of a graph problem is to restrict it to a smaller classes of graphs. In Chapter 4 we studied caterpillar problems on graphs of bounded treewidth. In this chapter we are going to study it on two other classes of graphs, block and interval graphs. We also briefly explain our idea of how one can generalize the notions of depth first search and breadth first search in graphs for searching for k-trees.

Here we first give a linear time algorithm that finds the *heaviest* caterpillar in a node-weighted tree. Then we use this result to find a largest caterpillar in a block graph. Then we prove that every interval graph has a spanning caterpillar. In the last section we introduce depth first and breadth first search of k-trees.

6.1 The Largest Caterpillar in a Tree

In this section we suppose that we have a node-weighted tree. Our goal is to find a subtree as a caterpillar such that the sum of the weights of its nodes is as large as possible. It is easy to see that there is an $O(n^3)$ algorithm for this problem, we first find the weights of all caterpillars that have a pair of leaves as ends and then we choose the one with the largest weight. But here we are interested in finding a linear time algorithm. Since we are going to use it in another algorithms, we prefer that the running times of our algorithms are not dominated by such a subroutine.

We can use Dijkstra's algorithm for finding the longest path in a tree to find

a largest caterpillar in a tree. In Dijkstra's algorithm we suppose each edge of the tree is a string of the physical length the same as its weight. Then we pick up a node and let the other nodes hang down. Then we choose one of the deepest node a and pick it up. Then the distance between node a and one of the deepest node, say b, is the length of the longest path in the tree. See Bulterman et al. [15] for a formal specification and a proof.

To apply this idea for finding the largest caterpillar in tree, we first give weights to the nodes with respect to the number of their adjacent nodes. Then we follow the same process for our node-weighted tree. That the algorithm finds the largest caterpillar in a tree follows the same argument as the proof of Dijkstra algorithm [15]. We also refer the reader to a paper of Uehara and Uno [58] for finding a longest path in a tree and some other classes of graphs.

6.2 Block Graphs

A graph G is a *block graph* if G is connected and every maximal 2-connected subgraph of G is complete, see [14]. We can consider a block graph as a graph that is made by replacing each edge in a tree by a clique of arbitrary size, where each pair of cliques are joined by at most one node, see Figure 6.1. Here we want to find a largest caterpillar in a block graph.

Given a block graph G, we use the result of the former section about trees to find the largest caterpillar in G. We first make a node-weighted tree T_G where each vertex in T_G corresponds to a clique in G and each pair of nodes in T_G are connected by an edge if their corresponding cliques have a node in common. Then we assign weights to nodes in such a way that the weight of a node $v \in V(T_G)$ is equal to the order of the corresponding clique of v, Figure 6.2 shows the node-weighted tree of the block graph depicted in Figure 6.1.

As the next step we find a caterpillar with the largest weight in T_G. To find the caterpillar within G we first order the nodes on the spine of C_T head to head. Let $v_1, v_2, \ldots, v_k$ be the ordered spinal nodes of T_G. Assume $Q_i \subseteq G$ is the associated clique of v_i, where $1 \leq i \leq k$. Let Q_{i-1}, Q_i and Q_{i+1} be three consecutive cliques that are associated to v_{i-1}, v_i and v_{i+1}, respectively.

Then we find the sequence of nodes $s_2, s_3, \ldots, s_k$ in G where s_i, $2 \leq i \leq k$, is

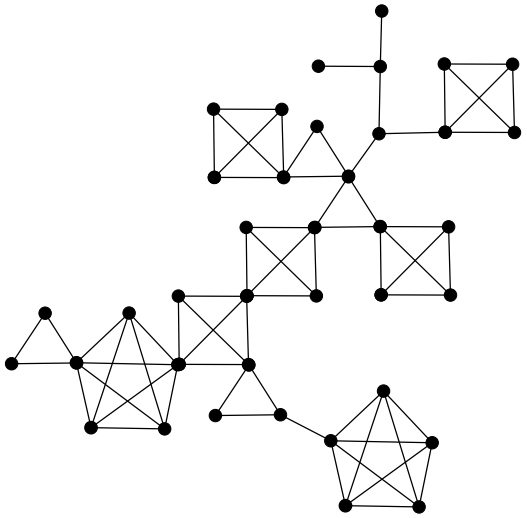

Figure 6.1 A block graph.

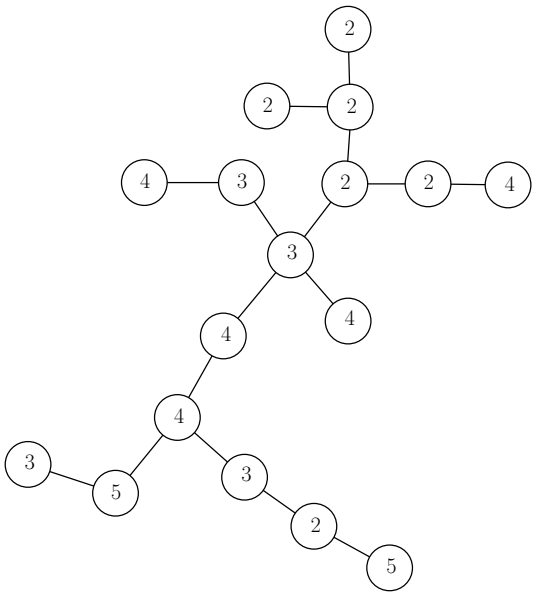

Figure 6.2 A node-weighted tree representation.

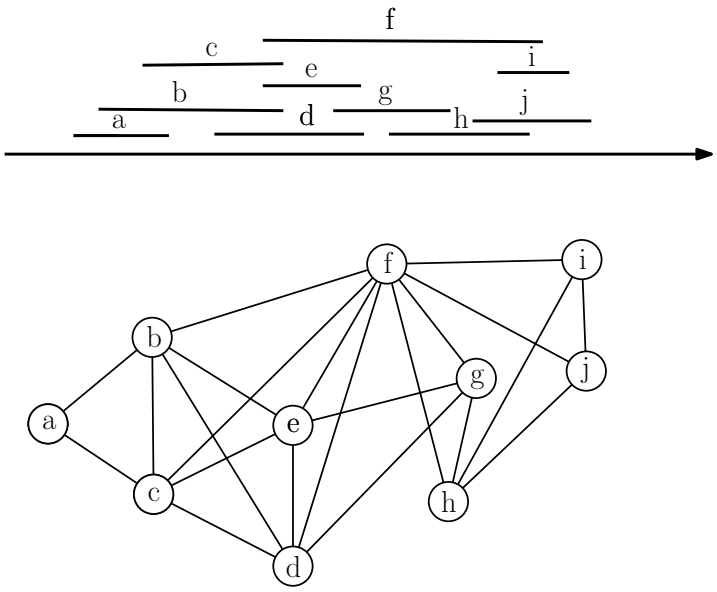

Figure 6.3 A graph and its interval representation.

the joining node between Q_{i-1} and Q_i. We extend this list by adding $s_1 \in Q_1$ where $s_1 \neq s_2$ and also by adding $s_{k+1} \in Q_k$ where $s_{k+1} \neq s_k$. Since each Q_i is a clique, there is a path P_i in Q_i between s_i and s_{i+1} that spans Q_i.

In order to make the spine of the desired caterpillar in G, we join P_i to P_{i+1}, $1 \leq i \leq k$. It is trivially seen that the nodes of the cliques that have common nodes with Q_is, appear as leaves in this caterpillar.

6.3 Interval Graphs

In this section we show that each interval graph has a spanning caterpillar. Informally an interval graph is a graph that its nodes correspond to intervals on the real line.

To define interval graphs formally we first need to give the definition of an intersection graph. Let $\{S_1, S_2, \ldots, S_n\}$ be a set of non empty subsets of a set S. We assign a node v_i to a set S_i, and we connect nodes v_i to v_j if $S_i \cap S_j \neq \emptyset$, where $1 \leq i, j \leq n$ and $i \neq j$. We call such a graph the *intersection graph* of $S_1, \ldots, S_n$.

Now let $T = \{I_1, I_2, \ldots, I_n\}$ be a finite collection of intervals in the real line

where G_I is its intersection graph. A graph G is an *interval graph* if G is the intersection graph G_I of a finite set of interval on the real line; see [14]. In Figure 6.3 an interval graph and its interval representation is shown.

It is known that every interval graph can be recognized in linear time, also there is an algorithm that for a given interval graph G produces its interval model G_I in linear time; see [49].

In what follows we assume that we have an interval model of an interval graph. We show that each interval graph has a spanning caterpillar.

Theorem 19 *Let G be an interval graph. Then there is a caterpillar as a subgraph of G that spans G.*

Proof. Let T be a caterpillar in G with the largest order. Also assume there is a node $v \in G(V)$ that does not belong to $V(T)$. We assume that v has a neighbour $u \in V(T)$, otherwise we find a path from v to T and we consider the last node on this path that does not belong to T as v. Let (a, b) and (c, d) be the corresponding intervals of v and u in G_I, respectively.

Since v is connected to u, the intervals (a, b) and (c, d) overlap. So we assume that $a < c < b < d$. Since $v \notin V(T)$ there is no spinal node of T that its interval representation overlaps with (a, b).

Then we find the spine node s for which in its interval representation (i, j), the value of i is the smallest among all other spine nodes. It is easy to see that (c, d) overlaps (i, j). Then we connect u as the new spine node to s and then we can add v to the extended spine and we find a caterpillar with the order one more than T. Which contradicts the assumption that T has the largest order.

□

We close this section by reminding the reader that since a complete graph belongs to the class of interval graphs, then the minimum spanning caterpillar problem still remains **NP**-hard for interval graphs.

6.4 Depth-First and Breadth-First Search for k-Trees

There is an efficient algorithm for recognizing a k-tree. One can choose consecutively k-leaves to remove nodes from a graph in any order. If the process fails

to find a k-leaf at a step then the graph is not a k-tree. This algorithm imposes an ordering on the nodes that is the same as a perfect elimination scheme. But as simply seen, this process is not applicable for finding a spanning k-tree that is *hidden* as a subgraph in a graph.

The depth-first search and the breadth-first search algorithms are used as subroutines in many graph algorithms. Since k-trees are considered as generalization of trees, one may ask how we can generalize these algorithms for finding a hidden k-tree in a graph? We try to answer this question by introducing a heuristic algorithm that, if it is properly implemented, resembles BFS and DFS algorithms. We refer to this algorithm as the k-tree search algorithm (KTS).

Note that in a graph that has a spanning k-tree, each node is attached to at least one k-clique. We use this trivial fact for designing a heuristic algorithm to extract k-trees of a graph G. To save information during the execution of our algorithm we use a list L to save the order of nodes in a reverse order of their appearance in a perfect elimination scheme. We also use a set S (with a priority structure) to save k-cliques that appear during the search process.

We first choose an arbitrary node v of the graph. Then we find a k-clique K in $G[N(v)]$ and save K in S. We also add the nodes in K as the first k nodes to L and assign the empty set as the parent of all nodes.

We repeat the following steps until S becomes empty. We remove a k-clique K from S and add each node $u \in K$ to L if it is not added yet. For each $w \in \bigcap_{x \in K} N(x)$ if w is not in L, we update its parent by considering K as the new parent, otherwise we do not change the parent of w. We then save all k-cliques comprising w and each set of $k-1$ nodes of K in S.

Note that if one implements S (as a dynamic set) by a queue the resulting k-trees extend in a breadth first search fashion, and if S is implemented by a stack the resulting k-trees extend in a depth first search fashion. Sometimes, due to improper choices of nodes or k-cliques, an output of the algorithm is a forest of k-trees and it does not cover all nodes of a graph. For example in Figure 6.4 if we first choose the node v and then the edge (u, w) as a 2-clique, the first resulting 2-tree is $G[\{v, u, w\}]$. This cause the output of the algorithm consists of at least four connected components.

As a solution to such problems we introduce the concept of the neighbourhood density to guide the search in a proper way. In a graph G the *neighbourhood-*

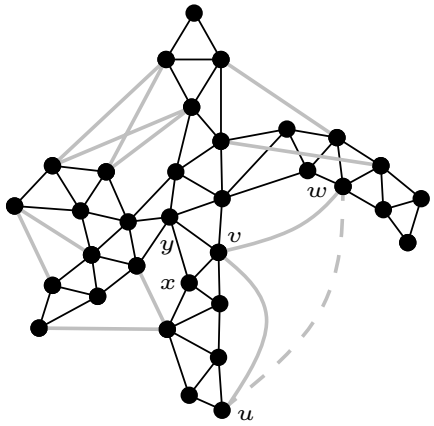

Figure 6.4 A graph with a spanning 2-tree.

density of two subgraphs H and H' of G is defined as

$$d(H, H') = |N(H) \cap N(H')|.$$

This concept can be used in the algorithm by this way: In the first step choose the k-clique K such that $d(v, K)$ gets the maximum value among all k-cliques that are neighbours of v. Also in the iterative steps of the algorithm, when a k-clique is chosen from the set S, allow the next k-clique to be chosen if it is built from a node $w \in \bigcap_{x \in K} N(x)$ and a set of $k-1$ nodes of K such that

$$d(w, K) = max \left\{ d(u, K) \mid u \in \bigcap_{x \in K} N(x) \right\}.$$

Now if one uses the guided version of the algorithm for the graph in Figure 6.4, the first 2-clique is (x, y) rather than (u, v). Choosing the next 2-cliques by the same way, produces a larger k-tree in comparison with the unguided version of the KTS algorithm.

We use the KTS algorithm as a subroutine to find a large k-caterpillar. We refer to this as the k-caterpillar search algorithm (KCS). To this end we use a

function ω that is defined, on a path P, as

$$\omega(P) = \left| \bigcup_{v \in V(P)} N[v] \right|.$$

The function ω computes the number of nodes in a k-path comprising the number of k-leaves that are attached to it. The KCS algorithm first uses the k-tree search algorithm to find a forest of k-trees in a graph. Then for each pair of k-leaves, it finds a k-caterpillar that has the largest number of nodes. This is done by computing the weight function ω for each k-caterpillar between a pair of k-leaves.

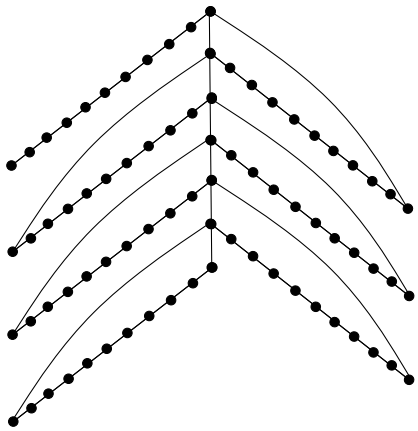

Figure 6.5 A somewhat difficult graph to find a spanning 1-caterpillar.

It is worth mentioning that the result of this algorithm depends on the result of the k-tree search algorithm. For example see Figure 6.5. The graph is shown as though the final result of applying the depth-first search algorithm; with back-edges that are not part of the tree. As one can easily check by choosing the path with the largest *weight*, a small portion of the total number of nodes is covered.

6.5 Summary and Open Problems

In this chapter we introduces a few results on finding caterpillars in three classes of graphs: trees, block graphs and interval graphs. We also extended the no-

tions of depth first search and breadth first search to k-trees. It is interesting to know for which other classes of graphs the maximum caterpillar problem has a polynomial-time algorithm.

Index

Bibliography

[1] Farid Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal of Optimization*, 5(1):13–51, 1995.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42:844–856, July 1995.

[3] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

[4] Stefan Arnborg and Andrzej Proskurowski. Characterization and recognition of partial k-trees. In *Proceedings of the sixteenth Southeastern international conference on combinatorics, graph theory and computing (Boca Raton), Florida, 1985*, volume 47, pages 69–75, 1985.

[5] Egon Balas, Sebasián. Ceria, Gerard. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996.

[6] Roberto Baldacci, Mauro Dell'Amico, and J. Salazar González. The capacitated-ring-star problem. *Operations Research*, 55(6):1147–1162, 2007.

[7] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear programming and network flows*. John Wiley & Sons Inc., Hoboken, NJ, fourth edition, 2010.

[8] D. Berend and T. Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2), 2010.

[9] Marshall Bern. *Network design problems: Steiner trees and spanning k-trees*. PhD thesis, University of California, Berkely, 1987.

[10] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 222–233. Springer, 2004.

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

[12] Hans L. Bodlaender. Treewidth: structure and algorithms. In *Structural information and communication complexity*, volume 4474 of *Lecture Notes in Computer Science*, pages 11–25, Berlin, 2007. Springer.

[13] J. A. Bondy and U. S. R. Murty. *Graph theory*, volume 244 of *Graduate Texts in Mathematics*. Springer, New York, 2008.

[14] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: a survey*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1999.

[15] R. W. Bulterman, F. W. van der Sommen, Gerard Zwaan, Tom Verhoeff, A. J. M. van Gasteren, and W. H. J. Feijen. On computing a longest path in a tree. *Information Processing Letters*, 81(2):93–96, 2002.

[16] Leizhen Cai. On spanning 2-trees in a graph. *Discrete Applied Mathematics*, 74(3):203–216, 1997.

[17] Leizhen Cai and Frédéric Maffray. On the spanning k-tree problem. *Discrete Applied Mathematics*, 44(1-3):139–156, 1993.

[18] Sebastián Ceria, Gérard Cornuéjols, and Milind Dawande. Combining and strengthening Gomory cuts. In *Integer programming and combinatorial optimization (Copenhagen, 1995)*, volume 920 of *Lecture Notes in Computer Science.*, pages 438–451. Springer, Berlin, 1995.

[19] W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Appl. Math.*, 18:25–38, November 1987.

[20] Gérard Cornuéjols, Yanjun Li, and Dieter Vandenbussche. k-cuts: a variation of Gomory mixed integer cuts from the LP tableau. *INFORMS Journal on Computing*, 15(4):385–396, 2003.

[21] Bruno Courcelle. Graph rewriting: an algebraic and logic approach. In *Handbook of theoretical computer science, Vol. B*, pages 193–242. Elsevier, Amsterdam, 1990.

[22] Bruno Courcelle. The monadic second-order logic of graphs. I. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[23] Miklos Csuros, J. Andrew Holey, and Igor B. Rogozin. In search of lost introns. *Bioinformatics*, 23(13):i87–96, 2007.

[24] Reinhard Diestel. *Graph theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, third edition, 2005.

[25] Michael J. Dinneen. Practical enumeration methods for graphs of bounded pathwidth and treewidth. Technical Report CDMTCS–055, Center for Discrete Mathematics and Theoretical Computer Science, Auckland, 1997.

[26] Michael J. Dinneen and Masoud Khosravani. A linear time algorithm for the minimum spanning caterpillar problem for bounded treewidth graphs. In Boaz Patt-Shamir and Tinaz Ekim, editors, *Structural Information and Communication Complexity*, volume 6058 of *Lecture Notes in Computer Science*, pages 237–246. Springer Berlin / Heidelberg, 2010.

[27] Michael J. Dinneen and Masoud Khosravani. Hardness of approximation and integer programming frameworks for searching for caterpillar trees. In Alex Potanin and Taso Viglas, editors, *Computing: The Australasian Theory Symposium (CATS 2011)*, volume 119 of *CRPIT*, pages 145–150, Perth, Australia, 2011. ACS.

[28] Rodney G. Downey and Michael R. Fellows. *Parameterized complexity*. Springer-Verlag, New York, 1999.

[29] Sherif El-Basil. Caterpillar (Gutman) trees in chemical graph theory. In Ivan Gutman and Sven Cyvin, editors, *Advances in the Theory of Benzenoid Hydrocarbons*, volume 153 of *Topics in Current Chemistry*, pages 273–289. Springer Berlin / Heidelberg, 1990.

[30] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[31] Arthur M. Farley. Network immune to isolated failures. *Networks*, 11:255–268, 1981.

[32] Uriel Feige and Kunal Talwar. Approximating the bandwidth of caterpillars. *Algorithmica*, 55(1):190–204, 2009.

[33] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Association for Computing Machinery*, 42(6):1115–1145, 1995.

[34] Ralph E. Gomory. An algorithm for integer solutions to linear programs. In *Recent advances in mathematical programming*, pages 269–302. McGraw-Hill, New York, 1963.

[35] D. Gonçalves. Caterpillar arboricity of planar graphs. *Discrete Mathematics*, 307(16):2112–2121, 2007.

[36] Martin Grohe. Algorithmic meta theorems. In Hajo Broersma, Thomas Erlebach, Tom Friedetzky, and Daniël Paulusma, editors, *WG*, volume 5344 of *Lecture Notes in Computer Science*, page 30, 2008.

[37] Arvind Gupta and Naomi Nishimura. Characterizing the complexity of subgraph isomorphism for graphs of bounded path-width. In *STACS 96 (Grenoble, 1996)*, volume 1046 of *Lecture Notes in Computer Science*, pages 453–464, Berlin, 1996. Springer.

[38] Arvind Gupta, Naomi Nishimura, Andrzej Proskurowski, and Prabhakar Ragde. Embeddings of k-connected graphs of pathwidth k. *Discrete Applied Mathematics*, 145(2):242–265, 2005.

[39] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976.

[40] Frank Harary and Allen J. Schwenk. The number of caterpillars. *Discrete Mathematics*, 6:359–365, 1973.

[41] Petr Hlinený, Sang il Oum, Detlef Seese, and Georg Gottlob. Width parameters beyond tree-width and their applications. *The Computer Journal*, 51(3):326–362, 2008.

[42] David R. Karger, Rajeev Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[43] Ken-ichi Kawarabayashi and Bojan Mohar. Some recent progress and applications in graph minor theory. *Graphs and Combinatorics*, 23(1):1–46, 2007.

[44] Martine Labbé, Gilbert Laporte, Inmaculada Rodríguez Martín, and Juan José Salazar González. The ring star problem: polyhedral analysis and exact algorithm. *Networks*, 43(3):177–189, 2004.

[45] Antoni Lozano, Ron Y. Pinter, Oleg Rokhlenko, Gabriel Valiente, and Michal Ziv-Ukelson. Seeded tree alignment. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 5(4):503–513, 2008.

[46] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1982.

[47] Andrzej Proskurowski. Separating subgraphs in k-trees: cables and caterpillars. *Discrete Mathematics*, 49(3):275–285, 1984.

[48] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997.

[49] G. Ramalingam and C. Pandu Rangan. A unified approach to domination problems on interval graphs. *Inf. Process. Lett.*, 27(5):271–274, 1988.

[50] Neil Robertson and Paul D. Seymour. Graph minors II, algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[51] Donald J. Rose. On simple characterizations of k-trees. *Discrete Mathematics*, 7:317–322, 1974.

[52] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the Association for Computing Machinery*, 23(3):555–565, 1976.

[53] L. Simonetti, Yuri Frota, and Cid C. de Souza. An exact method for the minimum caterpillar spanning problem. In Sonia Cafieri, Antonio Mucherino, Giacomo Nannicini, Fabien Tarissan, and Leo Liberti, editors, *CTW*, pages 48–51, 2009.

[54] L. Simonetti, Yuri Frota, and Cid C. de Souza. Upper and lower bounding procedures for the minimum caterpillar spanning problem. *Electronic Notes in Discrete Mathematics*, 35:83–88, 2009.

[55] Maciej Syslo and Andrzej Proskurowski. On halin graphs. In M. Borowiecki, John Kennedy, and Maciej Syslo, editors, *Graph Theory*, volume 1018 of *Lecture Notes in Mathematics*, pages 248–256. Springer Berlin / Heidelberg, 1983.

[56] László A. Székely and Hua Wang. On subtrees of trees. *Advances in Applied Mathematics*, 34(1):138 –155, 2005.

[57] Jinsong Tan and Louxin Zhang. The consecutive ones submatrix problem for sparse matrices. *Algorithmica*, 48(3):287–299, 2007.

[58] Ryuhei Uehara and Yushi Uno. Efficient algorithms for the longest path problem. In *Algorithms and computation*, volume 3341 of *Lecture Notes in Computer Science*, pages 871–883. Springer, Berlin, 2004.

[59] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, 2001.

[60] Stanisław Walukiewicz. *Integer programming*, volume 46 of *Mathematics and its Applications (East European Series)*. Kluwer Academic Publishers Group, Dordrecht, 1991.