TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Master Thesis

Complexity and Algorithms for Weighted Subgraphs

Author:

Felix Hoffmann

29th of March, 2019

Department of Mathematics
Technische Universität Kaiserslautern

Supervisor:
Prof. Dr. Sven O. Krumke

Co - Supervisors:
M.Sc. Manuel Streicher
M. Sc. Eva Schmidt

Contents

Introduction

In some applications (for example as a subproblem of Pricing in Robust Optimization) the problem of finding a connected subgraph with maximum or minimum weight arises. Given an undirected vertex- and edge-weighted graph, the maximum weighted subgraph problem asks to identify a connected subgraph with maximal sum of weights. We are interested in solving this problem efficiently. The complexity and approximability of the weighted subgraph problem on different graph classes are examined. We show that in general, the weighted subgraph problem is $\mathcal{NP}$-complete and even $\mathcal{NP}$-complete to approximate within a constant factor.

We present several algorithms and heuristics for this problem, including a polynomial time dynamic programming algorithm on decomposable graphs and several integer linear programming formulations. In order to test the practical relevance, some of the algorithms presented in this thesis were also implemented. Among those is the dynamic programming algorithm for paths, trees, and series-parallel graphs as well as two heuristic methods. We will see that one of the proposed heuristic procedures, as well as the dynamic program, is a lot faster for solving weighted subgraph problems than a commercial IP solver.

Related Work

A related problem is the so-called densest k-subgraph problem. Here, one seeks a subgraph induced on k nodes, whose weight is maximized. This problem can be constantly approximated for weights fulfilling the triangle inequality [RRT94; Tam91]. For general weights, there are also approximations [KP93].

Furthermore, weighted subgraph problems were previously discussed in a paper by El-Kebir and Klau [EK14]. This thesis uses some of the ideas and results of this work.

Outlook

In the following, we will give an overview of the chapters of this thesis.

Chapter 1 is dedicated to the recapitulation of definitions from the field of graph theory, complexity theory and mathematical optimization that are necessary in order to formulate comprehensive mathematical statements in the further chapters. The reader familiar with these preliminaries may skip this chapter and only come back to it when needed. Subsequently, in Chapter 2, we introduce the notion of treewidth and the class of decomposable graphs and present some basic results. The remainder of this thesis is organized as follows.

Chapter 3 formulates the weighted subgraph problem among with some variations considered here. In Chapter 4, we present a polynomial time dynamic programming algorithm for the weighted subgraph problem on special graph classes – namely paths, trees, series-parallel graphs, and decomposable graphs. Afterward, in Chapter 5 the densest k-subgraph problem is defined and the applicability of techniques for approximating the weighted subgraph problem is examined. We show the hardness of approximating the weighted subgraph problem within a constant factor and describe some heuristic procedures in Section 6. Integer programming formulations exactly solving weighted subgraph problems are given in Chapter 1.2. In Chapter 8 we study the performance and practical use of our implementations on graphs of different sizes compared to commercial IP solvers. Finally, Chapter 9 lists our results and hints to further possible studies.

Acknowledgments

I would like to express my gratitude to all those who made it possible for me to complete this thesis. First of all, I would like to thank my supervisor Prof. Dr. Sven O. Krumke as my co-supervisors M. Sc. Eva Schmidt and M. Sc. Manuel Streicher for their professional support, assistance, and for their time.

Their office doors were always open when I had questions about my research, they were interested in my recent results, and they gave me helpful feedback and suggestions for improvements.

Special thanks go to Michelle Schulz, Oliver Bachtler and Matthias Bätjer who examined the final version of the thesis closely for correctness, English style and grammar.

Chapter 1

Basic Definitions

In this chapter, we go over some well-established concepts in the field of graph theory and mathematical optimization that are used in this thesis.

In Section 1.1 we introduce the basic definitions and notations of graph theory. After this, we define optimization problems and introduce the basics of integer programming in Section 1.2. We complete this chapter by establishing some concepts of complexity theory in Section 1.3 where we define decision problems, the $\mathcal{O}$-Notation and conclude with the introduction of the complexity classes $\mathcal{P}$ and $\mathcal{NP}$.

1.1 Graph Theory

Let us begin with the fundamentals and give several basic definitions from graph theory and present some notation that is used throughout the thesis. All of the definitions stated here can be found in [KN12]. We begin with the definition of a graph itself.

Definition 1.1 (Directed Graph)**.** A *directed graph* is a quadruple $G = (V, R, \alpha, \omega)$ where

(i) V is a nonempty set, the set of *nodes* or *vertices* of G.

(ii) R is a set, the set of *arcs* of G.

(iii) It holds that $V \cap R = \emptyset$.

(iv) $\alpha\colon R \to V$ and $\omega\colon R \to V$ are functions ($\alpha(r)$ is the start node, $\omega(r)$ the end node of the arc r).

For a graph G, we also refer to its set of vertices by $V(G)$ and its set of arcs by $R(G)$. We call a graph G finite if both the V and the R are finite. In that case, we let $n \coloneqq |V|$ denote the number of nodes and $m \coloneqq |R|$ denote the number of arcs in G.

Definition 1.2 (Loops and Parallel Edges)**.** Let $G = (V, R, \alpha, \omega)$ be a graph. An arc $r \in R$ is called *loop* if $\alpha(r) = \omega(r)$. The graph G is called *loop-free* if it contains no loops.
Two arcs $r, r' \in R$ with $r \neq r'$ are called *parallel* if $\alpha(r) = \alpha(r')$ and $\omega(r) = \omega(r')$. If G does not contain parallel edges or loops, it is called *simple*. In this case, every arc r is uniquely characterized by the pair $(\alpha(r), \omega(r))$ and we write $G = (V, R)$ for the directed graph G, where $R \subseteq V \times V$.

Definition 1.3 (Adjacency, Incidence, Degree)**.** Let $G = (V, R, \alpha, \omega)$ be a graph. A vertex $v \in V$ and an arc $r \in R$ are called *incident* if v is either the start or the end vertex of r, that is $v \in \{\alpha(r), \omega(r)\}$. Two arcs $r, r' \in R$ are called *incident* if there is a vertex v that is incident to both r and r'. Two vertices $v, v' \in V$ are called *adjacent* if there is an arc $r \in R$, such that r is incident to v and v'. For a vertex $v \in V$ we write:

$\delta_G^+(v) := \{r \in R \colon \alpha(r) = v\}$	for the set of *outgoing arcs* of v,
$\delta_G^-(v) := \{r \in R \colon \omega(r) = v\}$	for the set of *incoming arcs* of v,
$N_G^+(v) := \{\omega(r) \colon r \in \delta_G^+(v)\}$	for the set of *successors* of v,
$N_G^-(v) := \{\alpha(r) \colon r \in \delta_G^-(v)\}$	for the set of *predecessors* of v,
$g_G^+(v) := \lvert\delta_G^+(v)\rvert$	for the *outdegree* of v,
$g_G^-(v) := \lvert\delta_G^-(v)\rvert$	for the *indegree* of v,
$g_G(v) := g^+(v) + g^-(v)$	for the *degree* of v.

Oftentimes we only want to regard a part of a graph. Therefore we need the following definition.

Definition 1.4 (Subgraph, Induced Subgraph)**.** Let $G = (V, R, \alpha, \omega)$ be a graph. A graph $G' = (V', R', \alpha', \omega')$ is called *subgraph* of G (we write $G' \leq G$) if

(i) $V' \subset V$ and $R' \subset R$, and

(ii) $\alpha|_{R'} = \alpha'$ and $\omega|_{R'} = \omega'$.

For a subset $V' \subset V$ of the nodes we call

$$G[V'] := (V', R', \alpha|_{R'}, \omega|_{R'})$$

the *subgraph* of G *vertex-induced* by V' with $R' = \{r \in R \colon \alpha(r) \in V' \text{ and } \omega(r) \in V'\}$. For a subset $R' \subseteq R$ of the arcs we call

$$G_{R'} := (V, R', \alpha|_{R'}, \omega|_{R'})$$

the *subgraph* of G *arc-induced* by R'. For $v \in V$ we write $G - v$ for the vertex-induced subgraph $G[V \setminus v]$. Analogously, for $r \in R$ we denote by $G - r$ the arc-induced graph $G_{R \setminus r}$.

For some of the problems considered in this thesis the orientation of the arcs in the graph does not matter. Thus we would like to introduce the notion of an *undirected graph.*

Definition 1.5 (Undirected Graph)**.** An *undirected graph* is a triple $G = (V, E, \gamma)$ where

(i) V is a nonempty set, the set of *nodes* or *vertices* of G.

(ii) E is a set, the set of *edges* of G.

(iii) It holds that $V \cap E = \emptyset$.

(iv) $\gamma \colon E \to \{X \colon X \subseteq V \text{ with } 1 \leq |X| \leq 2\}$ is a function that maps each edge to its two endpoints (possibly the same).

Terms like *incidence*, *adjacency*, *degree*, *subgraph* etc. are defined analogously to the directed graphs. If for an edge e in an undirected graph $G = (V, E, \gamma)$ the set $\gamma(e)$ contains a single element, that is $\gamma(e) = \{v\}$, then e is referred to as a *loop* on v. We use the following notations for undirected graphs:

$\delta_G(v) := \{e \in E \colon v \in \gamma(e)\}$	for the set of *incident edges* to v,
$N_G(v) := \{u \in V \colon \gamma(e) = \{u, v\} \text{ for any } e \in E\}$	for the set of *neighbors* of v,
$g_G(v) := \sum_{e \in E \colon v \in \gamma(e)} (3 - \lvert\gamma(e)\rvert)$	for the *degree* of v.

The undirected graph G is called *simple* if it does not contain loops or parallels. For simple graphs, more generally for undirected graphs without parallels, one can consider each edge $e \in E$ as a set $e = \{u, v\} \subseteq V$ with at most two members. In that case, we also write $e = (u, v)$ and $G = (V, E)$ for the undirected graph G.

Definition 1.6 (Density)**.** The density d_G of an undirected graph $G = (V, E, \gamma)$ is its average degree. That is, $d_G = 2\frac{|E|}{|V|}$. When G is clear from the context, we denote the density by d.

Definition 1.7 (Associated Undirected Graph, Orientation)**.** Let $G = (V, R, \alpha, \omega)$ be a directed graph and define the undirected graph $H := (V, E, \gamma)$ with $E := R$ and $\gamma(e) := \{\alpha(e), \omega(e)\}$ for $e \in E$. Then the graph H is called the *undirected graph associated* with G. Vice versa, we call G an *orientation* of H.

Now that we have established some basic concepts, we define terms for structures within a graph.

Definition 1.8 (Paths and Cycles)**.** Let $G = (V, R, \alpha, \omega)$ be a graph. For some $k \geq 0$ we call a finite sequence $P = (v_0, r_1, v_1, r_2, \ldots, r_k, v_k)$ a *path* from v_0 to v_k in G if $v_0, \ldots v_k \in V$, $r_1, \ldots, r_k \in R$ with $\alpha(r_i) = v_{i-1}$ and $\omega(r_i) = v_i$ for all $i = 1, \ldots k$. Analogously we speak of a path $P = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k)$ in an undirected graph if $v_0, \ldots v_k$ are vertices and $e_1, \ldots, e_k$ are edges where e_i connects vertex v_{i-1} and v_i, that is, $\gamma(e_i) = \{v_{i-1}, v_i\}$ for all $i = 1, \ldots, k$. We call $|P| := k$ the *length* of the path. A path is called *simple* if $r_i \neq r_j$ (or $e_i \neq e_j$ in the undirected case) for $i \neq j$, that is if it does not use an arc (an edge) more than once. A path is called *elementary* if it is simple and – except for the case that start and end vertex coincide – no vertex is touched more than once. If for some path C with $k \geq 1$ we have $v_0 = v_k$, we call C a *cycle*. The notions of simple and elementary cycles transfer from paths. Whenever a graph G is simple, we omit the edges in the finite sequences for all paths since in this case the edges of the paths are uniquely determined by the vertices on the path.

Definition 1.9 (Connected Component)**.** Let $G = (V, E, \gamma)$ be an undirected graph. For a node $v \in V$ we denote by

$$C(v) := \{u \in V \colon \text{ there is a path } P = (v, \ldots, u) \text{ in } G\}$$

the set of nodes that are *connected* to v. We call such a set $C(v)$ a *connected component* of G. The graph G is called connected if $C(v) = V$ for any $v \in V$.

Definition 1.10 (Cut)**.** A cut (A, B) in a directed or undirected graph G is a partition of the set of vertices into two nonempty subsets, that is, $V(G) = A \cup B$ with $A \cap B = \emptyset$, $A \neq \emptyset$, and $B \neq \emptyset$.
In extension of the notations $\delta^+(v)$ and $\delta^-(v)$ we define for a subset $V' \subseteq V$ of the vertex set of a graph G:

$$\delta^+(V') := \{r \in R \colon \alpha(r) \in V' \text{ and } \omega(r) \in V \setminus V'\}$$

$$\delta^-(V') := \{r \in R \colon \omega(r) \in V' \text{ and } \alpha(r) \in V \setminus V'\}$$

If G is undirected, then we define $\delta(U)$ as the set of edges with exactly one end in V':

$$\delta(V') := \{e \in E \colon \gamma(e) = (u, v) \text{ with } u \in V' \text{ and } v \in V \setminus V'\}.$$

We will now define trees.

Definition 1.11 (Tree, Forest)**.** A simple undirected graph $F = (V, E, \gamma)$ is called a *forest* if it does not contain any cycles. Furthermore, if a graph $T = (V, E, \gamma)$ is a forest and connected, we call T a *tree.* A subgraph T' of T is called a *subtree* if it is connected.

Definition 1.12 (Root, Rooted Tree, Binary Tree)**.** Let $T = (V, E, \gamma)$ be a tree. We can choose any $r \in V$ and call T a *rooted tree* with *root* r. In a rooted tree T with root r, we define the *depth* d_v of a node $v \in V$ as the length of a simple path from r to v. Furthermore, all nodes of depth i are referred to as the nodes on the ith *level* of the tree. Let $v \in V$ be some node in the tree. Then a node u is called a *child* of v if $d_u = d_v + 1$ and v is adjacent to u. If v is not the root we call the unique node u with $d_u = d_v - 1$ that is adjacent to v the *parent* of v. If u is the parent of v and w is the parent of u the node w is called the *grandparent* of v. Furthermore, u is a *descendant* of v if there is a path $(v = v_0, v_1, \dots, v_k = u)$ in the tree T such that v_i is a child of v_{i-1} for $i = 1, \dots, k$. We call the subtree with root v that contains all descendants of v the subtree rooted at v. A node u is said to be *below* v in the tree T if u is in the subtree rooted at v. Analogously u is *above* v in T if v is in the subtree rooted at u. The nodes in T without children are called leaves. A rooted tree T is called *binary tree* if no node in T has more than two children. We call a binary tree *full* if no node has exactly one child.

1.2 Integer Programs and Relaxations

We continue by defining optimization problems. An in-depth description of this subject is given in [Wol98].

Definition 1.13 (Optimization Problem)**.** Let $n > 0$ be a positive integer, $f \colon \mathbb{R}^n \to \mathbb{R}$ a function, and $X \subset \mathbb{R}^n$. We define

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in X \end{aligned} \tag{OP}$$

to be an *Optimization Problem* (OP). A solution $x \in \mathbb{R}^n$ is *feasible* for (OP) if $x \in X$. A feasible solution x^* is *optimal*, if $f(x^*) = \min\{f(x) \colon x \in X\}$.

Definition 1.14 (LP, IP, Linear Relaxation)**.** Let $\min\{f(x) \colon x \in X\}$ be an optimization problem. If $f(x) = c^T x$ is a linear function and $X = \{x \in \mathbb{R}^n \colon Ax \leq b\}$ for $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, it is referred to as a *Linear Problem* or *Linear Program* (LP) and is written as:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b, \\ & x \in \mathbb{R}^n. \end{aligned} \tag{LP}$$

If all variables are integer, meaning $X = \{x \in \mathbb{Z}^n \colon Ax \leq b\}$, we call the problem *(Linear) Integer Problem* or *(Linear) Integer Program* (IP) and write:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b, \\ & x \in \mathbb{Z}^n. \end{aligned} \tag{IP}$$

We call (LP) the *Linear Relaxation* of (IP).

Definition 1.15 (Integrality Gap)**.** Given an instance I of an integer program (IP) with its optimal value being $M_{\text{IP}}(I)$ and the optimal value of its relaxation LP being $M_{\text{LP}}(I)$. Then, the *Integrality Gap* for instance I is defined by:

$$IG(I) = \frac{M_{\text{IP}}(I)}{M_{\text{LP}}(I)}.$$

1.3 Complexity Theory

In this section, we give a brief introduction to complexity theory, as far as it is relevant for this thesis. Details can be found in [Pap94]. In order to be able to classify our results into complexity classes, we first need an exact definition of a decision problem. Informally, a decision problem is a problem that we can answer with “yes” or “no”.

Definition 1.16 (Decision Problem)**.** A *decision problem* Π consists of a set of instances $\mathcal{I}_\Pi$ and a set of yes-instances $\mathcal{Y}_\Pi \subset \mathcal{I}_\Pi$. Deciding Π for an instance $I \in \mathcal{I}_\Pi$ is to check, if $I \in \mathcal{Y}_\Pi$.

Every optimization problem can be reformulated as a decision problem. Given the representation of (OP) from Definition 1.13, the decision problem can be formulated as:

Is there a feasible solution $x \in X$ to (OP) with value $f(x) \leq k$?

In this case an instance of consists of k and a representation of f and X. For the linear program (LP), we would have the instance $I = (k, c, A, b)$.

Definition 1.17 (Complexity Classes $\mathcal{P}$ and $\mathcal{NP}$)**.** The class $\mathcal{P}$ consists of all decision problems, which can be solved in polynomial time by a deterministic algorithm. The class $\mathcal{NP}$ consists of all decision problems Π with the following property: Given an instance $I \in \mathcal{Y}_\Pi$ and a witness y with $L(y) = \mathcal{O}(L(I)^p)$ for a $p \in \mathbb{N}$, there is a polynomial algorithm using I and y as its input that is able to verify $I \in \mathcal{Y}_\Pi$.

From Definition 1.17 it follows immediately that $\mathcal{P}$ is a subset of $\mathcal{NP}$.

Definition 1.18 (Input Length)**.** Let I be an instance of a decision problem Π. The input length $L(I)$ is defined as the length of the binary encoding of the representation of the instance I.

Definition 1.19 (Polynomial Time Reduction)**.** We say that a decision problem Π can be reduced in polynomial time to a decision problem Π' and we write $\Pi \leq_p \Pi'$ if there is a function $f\colon \mathcal{I}_\Pi \to \mathcal{I}_{\Pi'}$ which can be computed in polynomial time with

$$x \in \mathcal{Y}_\Pi \iff f(x) \in \mathcal{Y}_{\Pi'}.$$

Definition 1.20 ($\mathcal{NP}$-complete Problem)**.** A problem $\Pi \in \mathcal{NP}$ is called $\mathcal{NP}$-complete if, for all problems $\Pi' \in \mathcal{NP}$, $\Pi' \leq_p \Pi$.

For an integer program $\min\{c^{\mathrm{T}}x | Ax \leq b, x \in \mathbb{Z}^n\}$ an instance is given by $I = (c, A, b)$. Given all data is integer, its input length sums up to

$$L(I) = \sum_{i=1}^{n} \lceil \log c_i \rceil + \sum_{j=1}^{m} \lceil \log b_j \rceil + \sum_{i=1}^{n} \sum_{j=1}^{m} \lceil \log A_{ij} \rceil.$$

In the following, we will introduce the $\mathcal{O}$-notation in order to later study and categorize running times of algorithms.

Definition 1.21 ($\mathcal{O}$-Notation)**.** Consider the two functions $f, g\colon \mathbb{N} \to \mathbb{N}$. We write $f(n) \in \mathcal{O}(g(n))$ if there exist $c, n_0 \in N$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Now, we are ready to formally define the running time of an algorithm.

Definition 1.22. Given a decision problem Π. Let A be an algorithm for deciding Π and I be an instance of Π. We define $f_A(I)$ to be the number of elementary calculations that A needs to run on I. Also, let $f_A^*(l) = \sup_{I \in \mathcal{I}_\Pi} \{f_A(I)\colon L(I) = l\}$ be the running time of algorithm A. We call the algorithm A polynomial for the problem Π, if $f_A^*(l) = \mathcal{O}(l^p)$ for some $p \in N$.

Chapter 2

Decomposable Graphs

In this chapter, we introduce the notion of treewidth and decomposable graphs. Section 2.1 defines a tree decomposition and recapitulates some well-known properties and results. After this, we define the class of decomposable graphs in Section 2.2.

2.1 Treewidth

Informally speaking, the treewidth of a graph is a measure of how “similar" the graph is to a tree. We refer to [Bod96] for a comprehensive introduction to treewidth. Since the introduction of the concept of treewidth, bounding the treewidth has been widely used to obtain polynomial time algorithms for problems that are otherwise $\mathcal{NP}$-complete. In order to formally define the treewidth of a graph, we need to introduce the structure of a tree decomposition.

Definition 2.1 (Tree Decomposition)**.** Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair $D = (S, T)$, where $S = \{X_i : i \in I\}$ is a family of subsets of V and $T = (I, F)$ is a tree with the following properties:

1. *Node coverage.* $\bigcup_{i \in I} X_i = V$.
2. *Edge coverage.* For every edge $e = (v, w) \in E$, there is a subset $X_i, i \in I$, with $v \in X_i$ and $w \in X_i$.
3. *Coherence.* For all $i, j, k \in I$ it holds that if j lies on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

We call the X_i the *bags* of the tree decomposition.

Definition 2.2 (Treewidth)**.** The *width* of a tree decomposition $D = (S, T)$, $S = \{X_i : i \in I\}$ for a graph G is defined as

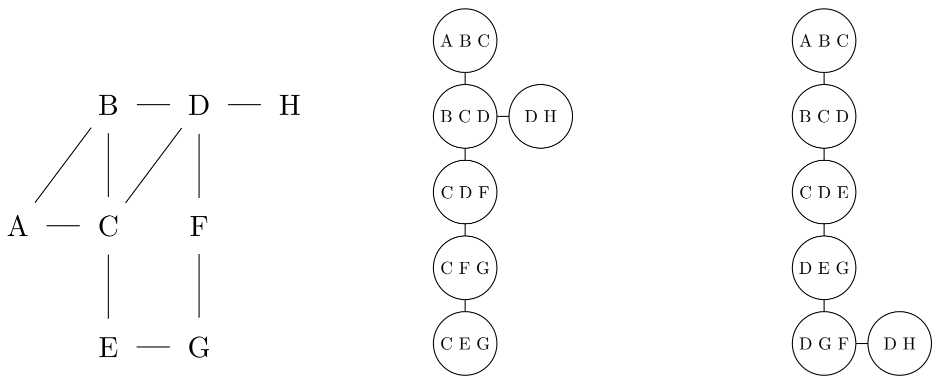
$$\text{width}(D) \coloneqq \max_{i \in I} |X_i| - 1.$$

The *treewidth* of G is denoted by

$$\text{tw}(G) \coloneqq \min_{D \text{ tree decomp.}} \text{width}(D)$$

and is the minimum width of all tree decompositions for G.

Example 2.3. Figure 2.1 gives an example of a graph G and two possible corresponding tree decompositions. G has eight vertices and both decompositions D_1 and D_2 are trees with six bags. Each bag lists at most three vertices, so the width of these decompositions is two: $\text{width}(D_1) = \text{width}(D_2) = 2$.



(a) Input graph G. (b) Tree-decomposition D_1. (c) Tree-decomposition D_2.

Figure 2.1: Input graph G with two possible valid tree decompositions D_1 and D_2, both with a treewidth of 2.

A few observations can be made that can aid in the understanding of the concept of tree decompositions.

- The tree decomposition must obviously satisfy the three properties given in Definition 2.1: node coverage, edge coverage, and coherence. All vertices are present, all edges can be associated with one or more bags, and the bags containing a certain vertex induce a subtree.
- The trivial tree decomposition consists of a single bag containing all the vertices. This satisfies all the constraints but is not useful in practice.
- Any non-leaf bag splits the graph into multiple disconnected components, making it a separator for the graph.
- Each clique is fully contained in a bag.
- Some bags can be connected to the tree at multiple locations, possibly reducing the degree of a bag.

While determining the exact treewidth of a graph is $\mathcal{NP}$-complete, there are efficient algorithms that can determine whether the treewidth is at most k.

Theorem 2.4. *Let $k \in N$ be a fixed constant. There exists a linear time algorithm, which decides for a graph G, whether $\operatorname{tw}(G) \leq k$ and if so, outputs a tree decomposition $D = (S, T)$ of G with $\operatorname{width}(D) = \operatorname{tw}(G)$ and $|V(T)| \in \mathcal{O}(n)$.*

For the proof of Theorem 2.4, we refer to [Bod96].

Observation 2.5. As a consequence of Theorem 2.4, given a class of graphs $\mathcal{G}$ and constant $k \in N$ such that $\operatorname{tw}(G) \leq k$ for all $G \in \mathcal{G}$, we can compute a tree decomposition of minimum width and size $\mathcal{O}(n)$ in linear time.

Up to this point, we have no information about the structure of a tree decomposition. The next definition introduces a special type of tree decomposition which is the analog of a rooted binary tree. It limits the structure to a small set of possible transitions between bags. This allows us to formulate procedures using these structural benefits.

Definition 2.6 (Nice Tree Decomposition)**.** Let $G = (V, E)$ be a simple undirected graph and $D = (S, T)$ a tree decomposition of G. D is called *nice* if the tree $T = (I, F)$ is rooted, binary and the following holds for every node $i \in I$:

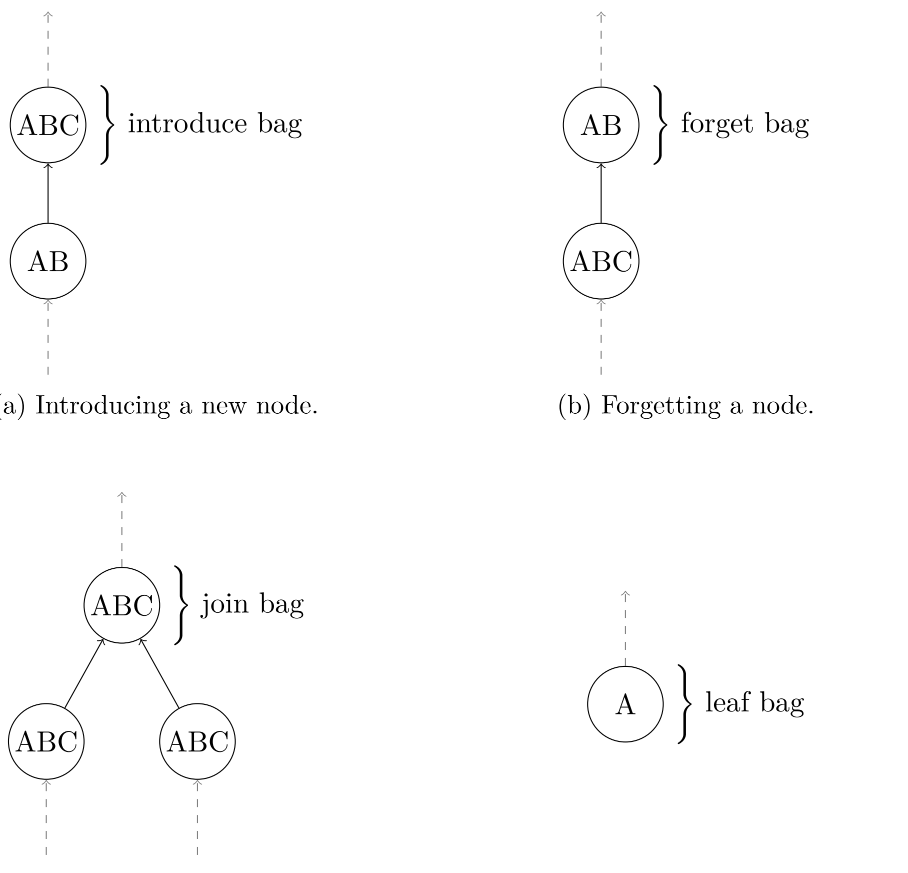
(i) If i is a leaf, then $|X_i| = 1$ and we call i *leaf bag*.

(ii) If i has one child j, then either

- $X_i = X_j \cup \{v\}$ for some $v \notin X_j$ (in this case we call i *introduce bag*) or
- $X_i = X_j \setminus \{v\}$ for some $v \in X_j$ (in this case we call i *forget bag*).

(iii) If i has two children j_1, j_2, then $X_i = X_{j_1} = X_{j_2}$ and we call i a *join bag*.

The nice tree decomposition remains a valid tree decomposition itself, but only allows for three different transitions between bags. Figure 2.2 illustrates the bags of a nice tree decomposition.



(a) Introducing a new node.

(b) Forgetting a node.

(c) Joining two bags.

(d) Leaf of the tree decomposition.

Figure 2.2: Bags in a nice tree decomposition.

It is not difficult to transform a given tree decomposition into a nice tree decomposition. More precisely, the following result holds (see [Klo94]).

Lemma 2.7. *Given a tree-decomposition $D = (S, T)$ of a graph G, we can compute a nice tree-decomposition $D_0 = (S_0, T_0)$ of G with*

$$\text{width}(D_0) \leq \text{width}(D)$$

and

$$|V(T_0)| \leq |V(T)| \cdot \text{width}(D)$$

in linear time.

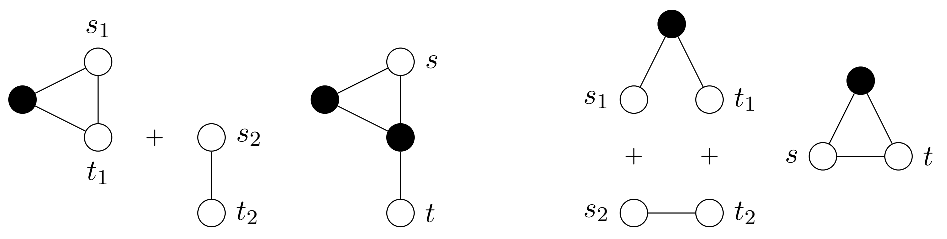
2.2 Decomposable Graphs

In this section, we turn to another class of graphs, the decomposable graphs. We begin by defining series-parallel graphs, which turn out to be a form of decomposable graphs. The definition of both graph classes can, for example, be found in [BLW87].

Definition 2.8 (Series-Parallel Graph)**.** The class of directed *series-parallel graphs* is defined recursively as follows: A graph consisting of a single edge (s, t) is *series-parallel* with *terminals* s and t, called *source* and *sink*, respectively. If two graphs G_1 and G_2 with sources s_1 and s_2 and sinks t_1 and t_2 are series-parallel, so is their *series* or *parallel composition*, which are defined as follows:

(i) The *series composition* is obtained by taking the disjoint union of G_1 and G_2 and identifying t_1 and s_2. The new source is s_1 and the new sink is t_2.

(ii) The *parallel composition* is obtained by taking the disjoint union of G_1 and G_2 and identifying s_1 and s_2 and also t_1 and t_2. The new source is s_1 respectively s_2 and the new sink is t_1 respectively t_2.

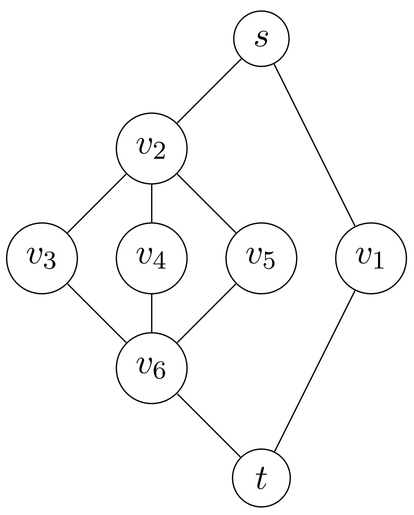


(a) Series composition.

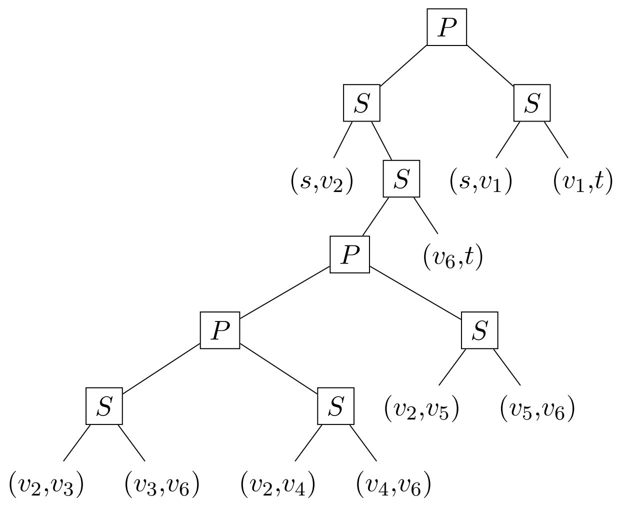
(b) Parallel composition.

Figure 2.3: Operations on series-parallel graphs.

It follows immediately from the definition that every series-parallel graph is connected. A "*decomposition-tree*" for a series-parallel graph displays its recursive construction according to the definition: Each tree node is a series-parallel graph and the children of each node are the subgraphs from which that graph was constructed by series or parallel composition. Figure 2.4 shows a series-parallel graph and one of its decomposition trees. Here, the leaf (u, v) denotes the series-parallel graph which consists only of the edge (u, v) and the label $P(S)$ on a node stands for the series-parallel graph obtained by parallel (series) composition of its children.



(a) A series-parallel graph G.



(b) A decomposition tree of G.

Figure 2.4: A series-parallel graph and one of its decomposition trees.

Lemma 2.9. *If G is a series-parallel graph, a decomposition tree of G can be constructed in time $\mathcal{O}(n)$.*

For the proof of Lemma 2.9, we refer to [VTL79]

Definition 2.10 (Decomposable Graphs)**.** Let Γ be a class of graphs. We call the graphs in Γ *decomposable* if they are given by a set of recursive rules which satisfy the following conditions.

(i) There is a finite number of *primitive* graphs in Γ, meaning only a finite number of graphs in Γ cannot be created with the recursive rules.

(ii) Each graph in Γ has an ordered (possibly empty) set of special vertices called *terminals.* The number of terminals is bounded by a constant $l \in N$.

(iii) There are a finite number of binary composition operations $\Gamma \times \Gamma \to \Gamma$ that act only at terminals, joining terminals either by identifying two terminals or by adding an edge (called an *attachment edge*) between two terminals. A composition operation also determines the terminals of the resulting graph, which must be a subset of the terminals of the composing graphs. Note that the operation does not necessarily act on all terminals.

An important cautionary note: Given a graph G and a set of rules defining a class of decomposable graphs Γ, it may be a difficult problem to recognize whether or not G belongs to Γ. For the remainder of this thesis, we shall simply assume that when we are given a graph G of Γ, we are also given a decomposition tree for G whose size is linear in the size of G. In such a decomposition tree, each leaf represents a primitive graph of Γ and each non-leaf node represents a composition operation (as well as a subgraph of G).

Remark 2.11*.* Series-parallel graphs form a class of decomposable graphs.

- The only primitive graph of the class is the graph consisting of a single edge connecting two different vertices.

- Each graph in the class has two terminals (called source and sink in Definition 2.8).
- It can be seen easily that the binary operations *series composition* and *parallel composition* are of the desired kind.

Other classes of decomposable graphs are trees, outerplanar graphs, and bandwidth-k graphs ([BLW87]).

It can be shown that every class of decomposable graphs has a fixed upper bound on the treewidth of the graphs in the class ([Wim87]).

Chapter 3

Weighted Subgraph Problems

This chapter is dedicated to describing the main problem of this thesis. In Section 3.1 we define several variants of the WEIGHTED SUBGRAPH PROBLEM, all seeking for a weighted subgraph of a given graph fulfilling certain properties. We continue by restricting the problem in Section 3.2, where we consider a special case when all node weights are negative and all edge weights are positive. After that, in Section 3.3 we develop some properties for the WEIGHTED SUBGRAPH PROBLEM. We finish the chapter by looking at the complexity of weighted subgraph problems in Section 3.4.

3.1 Problem Description

We start by defining the WEIGHTED SUBGRAPH PROBLEM and some variations. The problems discussed here can be formally described as follows: Let $G = (V, E, \gamma)$ be a graph. We indicate the *weight* of an edge $e \in E$ with $c(e)$ and the *weight* of a vertex $v \in V$ with $d(v)$.

To define the WEIGHTED SUBGRAPH PROBLEM we need the definition of a weight function:

Definition 3.1. Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d\colon V \to \mathbb{R}$ and edge weights $c\colon E \to \mathbb{R}$. A *weight function* w is defined as:

(i) $w(e) := c(e)$ for all $e \in E$.

(ii) $w(E') := \sum_{e\in E'} c(e)$ for all $E' \subseteq E$.

(iii) $w(v) := d(v)$ for all $v \in V$.

(iv) $w(V') := \sum_{v\in V'} d(v)$ for all $V' \subseteq V$.

(v) $w(G) := \sum_{e\in E} c(e) + \sum_{v\in V} d(v)$.

Definition 3.2 (Weighted Subgraph Problem)**.** Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d\colon V \to \mathbb{R}$ and edge weights $c\colon E \to \mathbb{R}$. Further, let w be a weight function according to Definition 3.1. Then:

(i) The WEIGHTED SUBGRAPH PROBLEM (WSP) asks for a connected subgraph H of G where $w(H)$ is minimized or maximized.

(ii) The WEIGHTED INDUCED SUBGRAPH PROBLEM (WISP) asks for a connected, induced subgraph H of G where $w(H)$ is minimized or maximized.

Now we define a rooted variant of the problem with one of the vertices forced to be in any solution.

Definition 3.3 (Rooted Weighted Subgraph Problem)**.** Given an undirected graph $G = (V, E, \gamma)$ with node weights $d\colon V \to \mathbb{R}$ and edge weights $c\colon E \to \mathbb{R}$, a weight function w according to Definition 3.1 and a root node $r \in V$.

(i) The Rooted Weighted Subgraph Problem (rwsp) is the problem of finding a connected subgraph H of G such that $r \in V(H)$ and $w(H)$ is minimized or maximized.

(ii) The Rooted Weighted Induced Subgraph Problem (rwisp) is the problem of finding a connected, induced subgraph H of G such that $r \in V(H)$ and $w(H)$ is minimized or maximized.

We can look at variations of the problem by specifying the number of nodes of the desired subgraph. Let $k \in \mathbb{N}$ be a natural number. Then:

Definition 3.4 (Weighted k-Subgraph Problem)**.** Given an undirected graph $G = (V, E, \gamma)$ with node weights $d\colon V \to \mathbb{R}$ and edge weights $c\colon E \to \mathbb{R}$, a weight function w according to Definition 3.1 and an integer $k \in \mathbb{N}$.

(i) The Weighted k-Subgraph Problem (wksp) asks for a connected subgraph H of G containing exactly k nodes where $w(H)$ is minimized or maximized.

(ii) The Weighted Induced k-Subgraph Problem (wiksp) asks for a connected, induced subgraph H of G containing exactly k nodes where $w(H)$ is minimized or maximized.

In general, the input for these problems is assumed to consist of a description of the graph (with node and edge weights). The problem can be modified by restricting the graph. For an undirected graph $G = (V, E, \gamma)$ with node and edge weights, we denote by max-wsp(G) a solution for the Weighted Subgraph Problem which has maximum weight, that is a subgraph of G. We use similar notations to denote solutions for all other weighted subgraph problems.

Since we are asking for connected subgraphs we assume input graphs to be connected. If a graph is not connected, solving any of these problems can be done by solving the problem on every connected component of the graph. Moreover for all problem except the rooted ones, the empty subgraph is a valid solution with weight zero.

For the remainder of this thesis we need the following lemma to recursively calculate weight function values:

Lemma 3.5. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d\colon V \to \mathbb{R}_{\geq 0}$ and edge weights $c\colon E \to \mathbb{R}_{\geq 0}$ and let $H \leq G$ be a subgraph of G. For a weight function w according to Definition 3.1 it holds, that:*

(i) When adding a vertex $v \in V \setminus V(H)$ to H then:

$$w(H + v) = w(H) + w(v) = w(H) + d(v).$$

(ii) When adding an edge (u, v) with $u \in V(H)$ and $v \in V \setminus V(H)$ to H then:

$$w(H + (u, v)) = w(H) + w(v) + w((u, v)) = w(H) + d(v) + c(e).$$

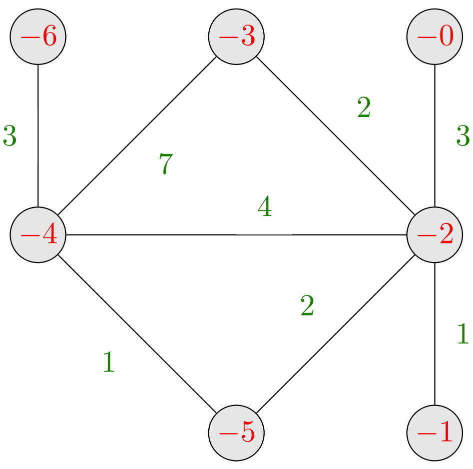
(iii) Suppose H is decomposable and was constructed from two graphs H_1 and H_2. Call the set of identified vertices I and the set of added edges A. Then:

$$\begin{aligned} w(H) &= w(H_1 + H_2) \\ &= w(H_1) + w(H_2) - w(I) + w(A) \\ &= w(H_1) + w(H_2) - \sum_{v \in I} d(i) + \sum_{e \in A} c(e). \end{aligned}$$

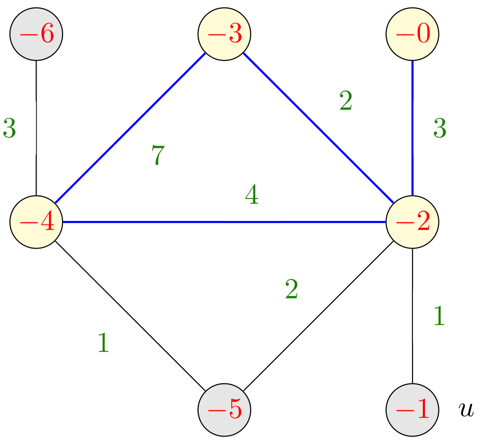
3.2 Restricting the Problem

Before we start looking at the general Weighted Subgraph Problem, we want to focus on a restricted version of the problem where all node weights are non-positive and all edge weights are non-negative. This restriction allows us to gain structural benefits. Later we also generalize some of the results developed here.

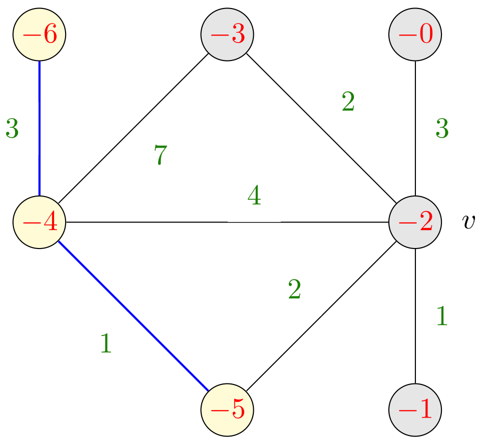
Assumption 3.6. Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d \colon V \to \mathbb{R}$ and edge weights $c \colon E \to \mathbb{R}$. We assume that $c(e) \geq 0$ for all $e \in E$ and $d(v) \leq 0$ for all $v \in V$.



(a) Input graph G.



(b) A maximum weighted subgraph of G.



(c) A minimum weighted subgraph of G.

Figure 3.1: A graph G satisfying Assumption 3.6 and two weighted subgraphs of G.

Example 3.7. Figure 3.1 shows a graph G and two weighted subgraphs of G. On the left side of the figure a MAX-WSP subgraph of G is shown. It has weight

$$\begin{aligned} w(H) &= \sum_{e \in E(H)} c(e) + \sum_{v \in V(H)} d(v) \\ &= (7+4+2+3) + (-4-3-2-0) = 7. \end{aligned}$$

The nodes of the subgraph are highlighted in yellow and the chosen edges are highlighted in blue. Notice that if we include the node u on the bottom right of G and the corresponding incident edge to be in the subgraph H, the weight $w(H)$ does not change. We can conclude that the maximum weighted subgraph is generally not unique.

On the right side of Figure 3.1 a MIN-WSP subgraph of G is shown. Its weight is

$$\begin{aligned} w(H) &= \sum_{e \in E(H)} c(e) + \sum_{v \in V(H)} d(v) \\ &= (3+1) + (-6-4-5) = -11. \end{aligned}$$

In this case, if the node v and the corresponding incident edge is added to the subgraph H, the weight $w(H)$ does not change.

The following lemma shows that in the restricted case there is no need to differentiate between MAX-WSP and MAX-WISP.

Lemma 3.8. *Let $G = (V, E, \gamma)$ be an undirected graph according to Assumption 3.6. Then the problems* MAX-WSP *and* MAX-WISP *are equivalent on G.*

Proof. Since edge weights always contribute non-negative values to $w(G)$ any optimal solution for MAX-WSP can be extended to be an induced subgraph of G. Suppose $H =$ MAX-WSP(G) is not an induced subgraph, adding the missing edges will not worsen the value of the weight function w, thus making H an optimal solution for MAX-WISP. Conversely, every optimal solution for MAX-WISP is already optimal for MAX-WSP. □

This implies that the MAX-WSP subgraph on the right side of Figure 3.1 is a solution for MAX-WISP on G, too. Consequently, due to Lemma 3.8, in the restricted version of the problem one can solve MAX-WSP by solving the more restrictive problem MAX-WISP. Note, that we do need to differentiate between MIN-WSP and MIN-WISP since in that case removing edges from an induced subgraph improves the value of the weight function. This yields the following lemma.

Lemma 3.9. *Let $G = (V, E, \gamma)$ be an undirected graph according to Assumption 3.6. Then a solution H for* MIN-WSP *on G is a spanning tree on a set $V' \subseteq V$.*

Proof. Suppose H is an optimal solution for MIN-WSP but is not a spanning tree. Then removing the edge e with the biggest cost $c(e)$ in H such that H will stay connected and repeating that process as long as possible will not make the value of the weight function w worse, but yield a spanning tree on a set $V' \subseteq V$. □

From 3.9 we get an immediate consequence for a relation between solutions of MIN-WSP and MIN-WISP.

Corollary 3.10. *Let $G = (V, E, \gamma)$ be an undirected graph according to Assumption 3.6. Then it holds that* MIN-WSP$(G) \leq$ MIN-WISP(G).

Proof. This follows from the fact that all edge weights are non-negative and every solution for MIN-WISP is feasible for MIN-WSP, but not vice-versa. □

3.3 General Problem

In the last Section 3.2, we have considered a restricted version of the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem. Now, we want to look at a general version of these problems where the node and edge weights are not restricted and ask ourselves whether the results obtained can be can be applied here, too.

Lemma 3.11. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $c\colon V \to \mathbb{R}$ and edge weights $d\colon E \to \mathbb{R}$. Define the weight-negated graph $G^-(:= V, E, \delta)$ with node weights $c'(v) = -c(v)$ for all $v \in V$ and edge weights $d'(e) = -d(e)$ for all $e \in E$. Then for two solutions $H = \text{MAX-WSP}(G)$ and $H' = \text{MIN-WSP}(G^-)$ it holds that $w(H) = -w(H')$.*

Proof. This follows directly from the construction of G^-. □

Although Lemma 3.11 seems very trivial it shows us that every instance of max-wsp can be transformed to an instance of min-wsp and there is no need to differentiate between maximizing or minimizing the weight function. For the remainder of this chapter, we refer to both problems simply as wsp.

The following lemma shows that under certain conditions solutions for the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem coincide.

Lemma 3.12. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d\colon V \to \mathbb{R}$ and edge weights $c\colon E \to \mathbb{R}$. Then the* Weighted Subgraph Problem *and the* Weighted Induced Subgraph Problem *are equivalent on G if any one of the following conditions hold:*

(i) G does not contain cycles or parallels, that is G is a tree.

(ii) In the case of maximizing all edge weights are non-negative and in case of minimizing all edge weights are non-positive.

Proof. If G does not contain cycles and parallel edges, every connected subgraph of G is induced. The other case follows from Lemma 3.8 and Lemma 3.11. □

Lemma 3.13. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $c\colon V \to \mathbb{R}$ and edge weights $d\colon E \to \mathbb{R}$. Then $\text{MAX-WSP}(G) \geq \text{MAX-WKSP}(G, k)$ for all $0 \leq k \leq |V|$. More precisely*

$$\text{MAX-WSP}(G) = \max_{0 \leq k \leq |V|} \text{MAX-WKSP}(G, k).$$

Analogously it holds that $\text{MIN-WSP}(G) \leq \text{MIN-WKSP}(G, k)$ for all $0 \leq k \leq |V|$ and

$$\text{MIN-WSP}(G) = \min_{0 \leq k \leq |V|} \text{MIN-WKSP}(G, k).$$

Proof. This follows from the definitions of wsp and wksp because solving wksp for all $0 \leq k \leq |V|$ yields an optimal solution for wsp. □

Similarly to the previous lemma, there is a direct relation between solutions of the Weighted Subgraph Problem and Rooted Weighted Subgraph Problem.

Lemma 3.14. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $c\colon V \to \mathbb{R}$ and edge weights $d\colon E \to \mathbb{R}$. Then* $\textsc{max-wsp}(G) \geq \textsc{max-rwsp}(G, v)$ *for all $v \in V$. More precisely*

$$\textsc{max-wsp}(G) = \max_{v \in V} \textsc{max-rwsp}(G, v).$$

Analogously, it holds that $\textsc{min-wsp}(G) \leq \textsc{min-rwsp}(G, v)$ *for all $v \in V$ and*

$$\textsc{min-wsp}(G) = \min_{v \in V} \textsc{min-rwsp}(G, v).$$

Proof. Solving RWSP for all $v \in V$ yields an optimal solution for WSP. □

3.4 Complexity Results

In this section, we regard the complexity of the Weighted Subgraph Problem and show that the problem is $\mathcal{NP}$-complete. In the progress, we obtain a stronger result and show that both problems are $\mathcal{NP}$-complete even under Assumption 3.6. We distinguish two cases: Maximizing and minimizing the weight function w.

Theorem 3.15. *Under Assumption 3.6 the* Weighted Subgraph Problem *and the* Weighted Induced Subgraph Problem *on undirected graphs are $\mathcal{NP}$-complete.*

Before we are able to prove Theorem 3.15 we need to formulate WSP and the WISP as decision problems. If the decision problem is hard, then, by definition, so is the optimization version.

For MAX-WSP we can formulate the corresponding decision problem as follows: Given an undirected graph G and a positive integer W, does G have a connected subgraph H with weight $w(H)$ at least W? Analogously the MIN-WSP decision problem: Given an undirected graph G and a positive integer W, does G have a connected subgraph H with weight $w(H)$ at most W?

We propose reductions from several well-known $\mathcal{NP}$-complete problems to prove Theorem 3.15.

The first $\mathcal{NP}$-complete problem we define is called Min-Steiner Tree.

Min-Steiner Tree

Given:	An undirected graph $G = (V, E, \gamma)$ and a subset of vertices $R \subseteq V$, usually referred to as *terminals*.
Problem:	Find a subtree $D \leq G$ using the least amount of edges that contains all terminals, that is $R \subseteq V(D)$.

The Min-Steiner Tree problem in graphs is defined in decisional form as follows: Is there a subtree D of G that includes all the vertices of R (i.e. a spanning tree for R) and contains at most k edges? For more details on Steiner trees and Steiner tree problems, we refer to [KN12].

Exact Cover by 3-Sets	
Given:	A ground set X with $\lvert X\rvert = 3q$ and $q \in \mathbb{N}$ (so, the size of X is a multiple of 3), and a collection C of 3-element subsets of X.
Problem:	Find a subset C' of C where every element of X occurs in exactly one member of C' (so, C' is an *exact cover* of X). In other words, is there a subset $C' \subseteq C$ such that $\bigcup_{S \in C'} S = X$.

The Exact Cover by 3-Sets problem is already in decisional form.

We are now ready to formally prove the $\mathcal{NP}$-completeness of the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem under Assumption 3.6.

Proof of Theorem 3.15. Suppose we are given a hypothetical solution H for any of the problems max-wsp, min-wsp, max-wisp or min-wisp. We can check in polynomial time that:

- The subgraph H of G has the desired properties, that is it is connected and for wisp additionally induced.
- In the case of maximizing, H has at least weight $w(H) \geq W$ and in the case of minimizing H has at most weight $w(H) \leq W$.

Thus, we can verify the solution in polynomial time which implies the problem is in $\mathcal{NP}$.

We propose reductions starting from generic instances of the above mentioned problems which are executable in polynomial time.

Let us start with min-wsp. Let (G, R, k) be an arbitrary instance of the Min-Steiner Tree problem. We can construct an instance of min-wsp in polynomial time as follows:

Define the set of vertices V' as:

$$V' = V \text{ with } d(v) = \begin{cases} -M, & \text{if } v \in R, \\ 0, & \text{if } v \notin R, \end{cases}$$

where $M > |E|$ is an integer. Basically, we put weight $-M$ on all vertices of the terminal set R and weight 0 on all other vertices in the graph G'.

Now define the set of edges:

$$E' = E \text{ with } c(e) = 1.$$

All edges are given weight 1 in $G' = (V', E')$. Now, if there exists a Steiner tree D in G with at most k edges, there also exists a subgraph $H = D$ of G' with weight $w(H) \leq k - |R|M$. Since D has minimum weight it uses the least amount of edge while containing all terminals. Having additional edges in H would only increase the weight $w(H)$. Having an additional vertex $v \notin R$ does not improve the total weight, since it has weight $w(v) = 0$, but worsens it since an additional edge would have to be used to connect v to the remainder of H. Conversely, suppose a solution $H =$ min-wsp(G) with weight $w(H) = k - |R|M$ exists, but does not contain all terminals. This is not possible since only the terminal vertices have weight $-M$ and

$M > |E|$. Additionally, MIN-WSP(G) does not contain more than k edges otherwise its weight would be greater than $k - |R|M$.

With $W = k - |R|M$ we constructed an instance (G, w, W) of MIN-WSP. Thus, the instance of MIN-WSP has a solution if and only if the initial instance of the MIN-STEINER TREE problem has a solution.

Now, assume we are maximizing and consider MAX-WSP.

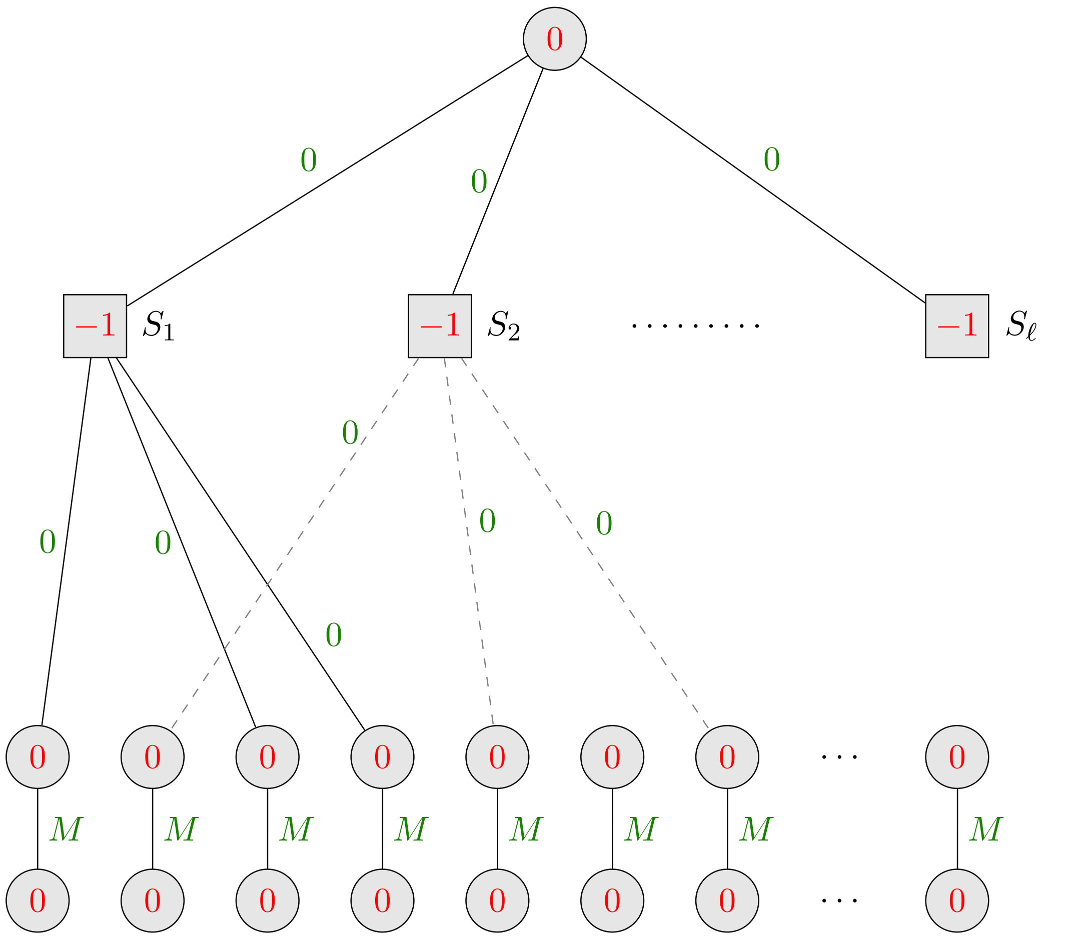


Figure 3.2: The basic $\mathcal{NP}$-hardness reduction from EXACT COVER BY 3-SETS to MAX-WSP

Let (X, C) an arbitrary instance of the EXACT COVER BY 3-SETS problem. We want to obtain an instance of MAX-WSP by constructing a graph. For every set $S \in C$ we create a vertex v_S and call it a set-vertex. For every element $x \in X$ we create two vertices v_x and w_x and call them element-vertices. The set-vertices are given weight 1, all other vertices have weight 0. More formally, we define the set of vertices V as

$$V = \{r\} \cup \{v_S \colon S \in C\} \cup \{v_x \colon x \in X\} \cup \{w_x \colon x \in X\} \text{ with}$$

$$d(v) = \begin{cases} -1, & \text{if } v \text{ is a set-vertex } v_S \text{ belonging to a set } S \in C, \\ 0, & \text{otherwise.} \end{cases}$$

We create edges from the root vertex r to all set-vertices, from each set-vertex v_S to the 3 vertices v_x with $x \in S$ and between v_x and w_x for all $x \in X$. Those edges between v_x and w_x are given a weight $M > k$, all other edges have weight 0.

$$E = \{(r, v_S) \colon S \in C\} \cup \{(v_S, v_x) \colon S \in C \text{ and } x \in S\} \cup \{(v_x, w_x) \colon x \in X\} \text{ with}$$

$$c(e) = \begin{cases} M, & \text{if } e = (v_x, w_x) \text{ for any } x \in X, \\ 0, & \text{otherwise.} \end{cases}$$

The graph obtained is illustrated in Figure 3.2.

Assume the instance (X, C) of Exact Cover by 3-Sets has a solution, that is there exists a subset $C' \subseteq C$ which covers X exactly. Suppose C' contains q 3-sets, then this solution corresponds to a solution for max-wsp, that is a connected subgraph H of G with weight $w(H) = |X|M - q$. All edges (v_x, w_x), their adjacent vertices, the vertices v_S with $S \in C'$, the edges (v_x, v_S) with $S \in C'$ and $x \in S$, the root vertex and the edges (r, v_S) with $S \in C'$ are selected. Since all edges with positive edge weights are selected and at least q of the sets $S \in C$ have to be selected, this is an optimal solution for the instance (G, w, W) of max-wsp with weight $w(H) = |X|M - q$. Conversely, suppose the instance (G, w, W) has a solution H with weight $w(H) = |X|M - q$. In this case it has to contain all edges (v_x, w_x) due to those being the only possibility to gain positive weight $|X|M$. The subgraph has to contain set-vertices to be connected. Since set-vertices decrease the value of the weight function, H chooses as little as possible which cover all vertices associated with elements of X. Additionally, H contains the root vertex and the edges to the q selected set-vertices. Selecting additional set-vertices would decrease the value of the weight function, thus making it lower than W. Therefore this corresponds to a solution for the instance (X, C) of Exact Cover by 3-Sets.

With $W = |X|M - q$ we constructed an instance (G, w, W) of max-wsp. Thus, the instance of max-wsp has a solution if and only if the initial instance of the Exact Cover by 3-Sets problem has a solution.

A similar reduction is possible for min-wisp (see Figure 3.3). Let $M > 3q$ be an integer. We slightly change the set of vertices and the weights:

$$V = r \cup \{v_S \colon S \in C\} \cup \{v_x \colon x \in X\} \text{ with}$$

$$d(v) = \begin{cases} -M, & \text{if } v \text{ is an element-vertex } v_x \text{ belonging to an element } x \in X, \\ 0, & \text{otherwise.} \end{cases}$$

And the set of edges:

$$E = \{(r, v_S) \colon S \in C\} \cup \{(v_S, v_x) \colon S \in C \text{ and } x \in S\} \text{ with}$$

$$c(e) = \begin{cases} 1, & \text{if } e = (v_S, v_x) \text{ for any } S \in C \text{ and } x \in S, \\ 0, & \text{otherwise.} \end{cases}$$

With $W = 3q - |X|M$ we constructed an instance (G, w, W) of min-wisp. Using the same arguments as before, min-wisp has a solution if and only if the initial instance of the Exact Cover by 3-Sets problem has a solution.

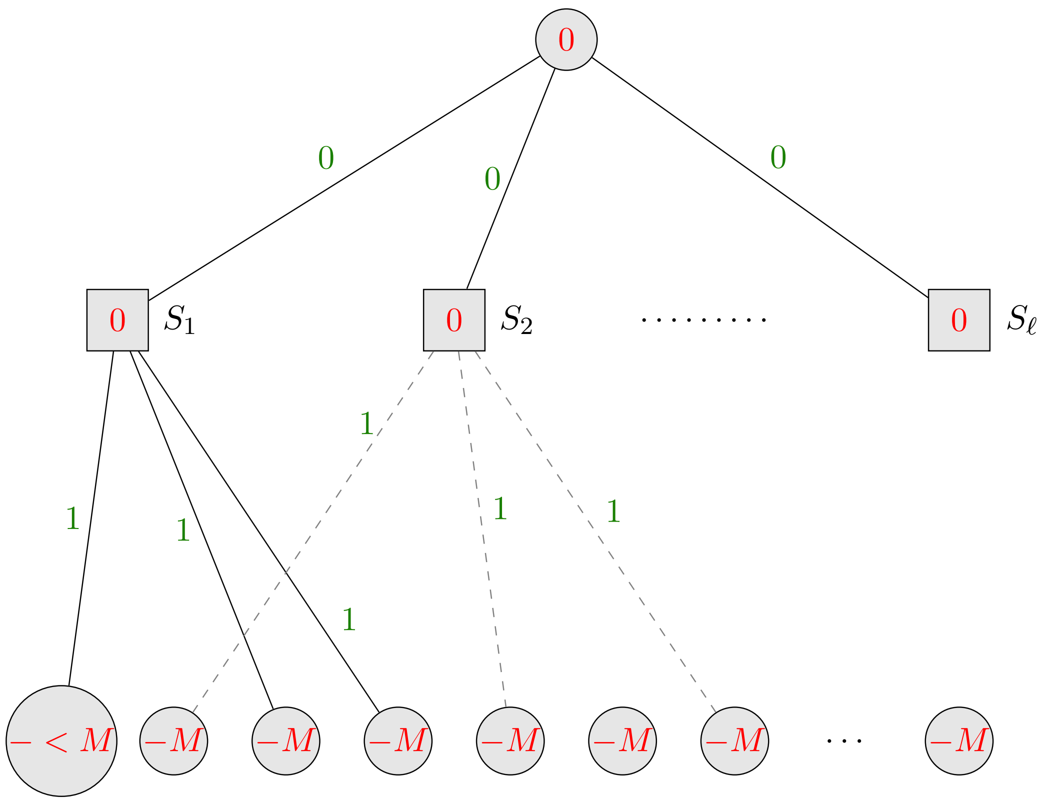


Figure 3.3: The basic $\mathcal{NP}$-hardness reduction from Exact Cover by 3-Sets to MIN-WISP

It is easy to see that all reductions can be done in polynomial time. Therefore, MIN-WISP, MAX-WISP, MIN-WSP and, due to Lemma 3.8, MAX-WSP are $\mathcal{NP}$-complete. □

Corollary 3.16. *The* Weighted Subgraph Problem *on undirected graphs with node weights $c\colon V \to \mathbb{R}$ and edge weights $d\colon E \to \mathbb{R}$ is $\mathcal{NP}$-complete.*

Proof. We have shown that the WSP is $\mathcal{NP}$-complete even under Assumption 3.6 in Theorem 3.15. Since this is a restriction of WSP, the problem WSP is $\mathcal{NP}$-complete, too. Otherwise one could solve instances of the restricted problem by solving them as instances of the general problem. □

Corollary 3.17. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d\colon V \to \mathbb{R}_{\geq 0}$ and edge weights $c\colon E \to \mathbb{R}_{\geq 0}$. Then the* Rooted Weighted Subgraph Problem *and the* Rooted Weighted Induced Subgraph Problem *are $\mathcal{NP}$-complete.*

Proof. Suppose RWSP is not $\mathcal{NP}$-complete. Then any instance of WSP can be solved by solving RWSP for each node $v \in V$ as a root (possibly including the empty graph). □

Corollary 3.18. *Let $G = (V, E, \gamma)$ be an undirected graph with node weights $d\colon V \to \mathbb{R}_{\geq 0}$ and edge weights $c\colon E \to \mathbb{R}_{\geq 0}$. Then the* Weighted k-Subgraph Problem *and the* Weighted Induced k-Subgraph Problem *are $\mathcal{NP}$-complete.*

Proof. Suppose WKSP is not $\mathcal{NP}$-complete. Then one can solve any instance of WSP by solving WKSP for each $0 \leq k \leq |V|$. $\square$

Chapter 4

Dynamic Programming Algorithm

In the last chapter we have seen that the problem is generally $\mathcal{NP}$-complete. In this chapter, we want to ask ourselves: Are there classes of graphs on which the Weighted Subgraph Problem or the Weighted Induced Subgraph Problem are polynomially or even linearly solvable?

Throughout this chapter a dynamic program will be presented which is executable in polynomial time for certain graph classes. In each section we focus on a specific class. From section to section the procedure is gradually enhanced.

We start by looking at paths in Section 4.1. This approach will be extended to trees and series-parallel graphs in Sections 4.2 and 4.3, respectively. In Section 4.4 we proceed with decomposable graphs and in the last Section 4.5 we have a brief glimpse on how to alter the dynamic program for solving the Rooted Weighted Subgraph Problem and the Rooted Weighted Induced Subgraph Problem.

4.1 Basic Idea and Dynamic Program on Paths

Let us first start with the Weighted Subgraph Problem on paths. If G is a path, every subgraph of G is a sub-path, since it has to be connected. Also note that Lemma 3.12 makes a differentiation between the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem unnecessary. In order to solve the Weighted Subgraph Problem one simply needs to look at the values of all sub-paths.

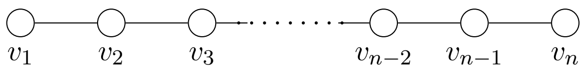


Figure 4.1: A path graph.

A path of length n has $(n-k)$ sub-paths of length k and $\binom{n}{2} = \frac{n(n-1)}{2}$ sub-paths in total, which is polynomial in n. Calculating the weight of a sub-path of length k takes $(k+1)+k$ time, since it has $k+1$ vertices and k edges. In total, this yields a running time of:

$$\sum_{k=1}^{n-1}(n-k)\cdot(2k+1) = \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n.$$

Therefore, the simple algorithm of going through all sub-paths yields the correct solution in polynomial time $\mathcal{O}(n^3)$.

We can achieve a better runtime by using a dynamic programming approach. Let $G = (v_1, v_2, \dots, v_n)$ be a path. Instead of looking at all sub-paths of G, we start with the sub-path consisting of only the node v_1 and start increasing the length of the sub-path. In step i we calculate $\textsc{wsp}(G_i)$ for the sub-path $G_i = (v_1, v_2, \dots, v_i)$.

We have to calculate and save two solutions to recursively reuse when increasing the length of the sub-path. One containing the current node v_i and one not containing it. By $H_{v_i}^{\text{in}}$ we denote the solution containing the current node v_i and by $H_{v_i}^{\text{out}}$ the one not containing it.

Our dynamic programming algorithm starts with $H_{v_1}^{\text{in}} = G_1 = v_1$ and $H_{v_1}^{\text{out}} = \emptyset$. When increasing the length of the sub-path from $i-1$ to i there are two possible solution candidates for $H_{v_i}^{\text{in}}$. One is the optimal solution $H_{v_{i-1}}^{\text{in}}$ containing v_{i-1} which we extend by adding v_i and the edge (v_{i-1}, v_i) to it. The other one is the graph just containing v_i. The algorithm compares the weight of both graphs and takes the maximum. Analogously, there are two possibilities for $H_{v_i}^{\text{out}}$, which are the two solutions for G_{i-1}, namely $H_{v_{i-1}}^{\text{in}}$ and $H_{v_{i-1}}^{\text{out}}$. Again, our algorithm compares their weights and takes the maximum.

Note, that we can compute the argmax in our algorithm in linear time because of Lemma 3.5.

Algorithm 4.1 MAX-WSP on paths

```
1: procedure WSP($G$)
2:     $H_{v_1}^{\text{in}} = v_1$
3:     $H_{v_1}^{\text{out}} = \emptyset$
4:     $i = 2$
5:     while $i \leq n$ do
6:         $H_{v_i}^{\text{in}} = \text{argmax}\left\{w(H_{v_{i-1}}^{\text{in}} + (v_{i-1}, v_i)), w(v_i)\right\}$
7:         $H_{v_i}^{\text{out}} = \text{argmax}\left\{w(H_{v_{i-1}}^{\text{in}}), w(H_{v_{i-1}}^{\text{out}})\right\}$
8:     return argmax $\left\{w(H_{v_n}^{\text{in}}), w(H_{v_n}^{\text{out}})\right\}$
```

Algorithm 4.1 can be used to solve MIN-WSP by computing the argmin instead of argmax. The argumentation is the same as before.

Theorem 4.1. *Algorithm 4.1 calculates a solution for the* WEIGHTED SUBGRAPH PROBLEM *and the* WEIGHTED INDUCED SUBGRAPH PROBLEM *on paths correctly in $\mathcal{O}(n)$ time.*

Proof. Let $G = (v_1, v_2, \ldots, v_n)$ be a path. We prove that when computing $H_{v_i}^{\text{in}}$ and $H_{v_i}^{\text{out}}$ as described in Algorithm 4.1 the argmax or argmin of $\{w(H_{v_i}^{\text{in}}), w(H_{v_i}^{\text{out}})\}$ is the optimal solution for WSP on the subgraph $G_i = (v_1, v_2, \ldots, v_i)$ of G for all $1 \leq i \leq n$ and, consequently, the correctness of the algorithm, that is $H_{v_i}^{\text{in}}$ is the optimal for G_i which contains v_i and $H_{v_i}^{\text{out}}$ is the optimal solution for G_{i-1}.

For the first vertex v_1, $H_{v_1}^{\text{in}}$ is initialized as the graph just containing v_1 and $H_{v_1}^{\text{out}}$ is the empty set, which are both optimal for the weight function w by construction. Suppose the solutions $H_{v_{i-1}}^{\text{in}}$ and $H_{v_{i-1}}^{\text{out}}$ are correctly computed and argmax or argmin of $\{w(H_{v_{i-1}}^{\text{in}}), w(H_{v_{i-1}}^{\text{out}})\}$ is the optimal solution for WSP on the subgraph G_{i-1}. Then H_i^{out} is the argmax or argmin of $\{w(H_{v_{i-1}}^{\text{in}}), w(H_{v_{i-1}}^{\text{out}})\}$, since it is the optimal solution for WSP on G_{i-1} and H_i^{out} is not allowed to contain the new vertex v_i.

Suppose $H_{v_i}^{\text{in}}$ is not the optimal solution for G_i, which contains v_i. Then neither the graph just containing v_i nor $H_{v_{i-1}}^{\text{in}}$ extended by adding the vertex v_i and the edge (v_{i-1}, v_i) are optimal. But those are the only possible subgraphs, since by induction

$H^{\text{in}}_{v_{i-1}}$ is the optimal solution for G_{i-1} containing v_{i-1} and a connected subgraph of G_i containing v_i either is v_i itself or has to contain v_{i-1}.

The algorithm loops through each vertex exactly once. In each step the length of the sub-path G_i increases by one and the algorithm has to calculate the weight of four sub-graphs. If we save the weights of the solutions for G_{i-1} we can calculate the weights for G_i in constant time (see Lemma 3.5). This yields a running time of $\mathcal{O}(n)$. □

4.2 Continuing with Trees

Now, that we can solve the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem on paths efficiently, we want to continue with a bigger class of graphs, namely trees. If G is a tree, every connected subgraph of G is also a tree. Again, theres no need to differentiate between connected and induced subgraphs. The dynamic programming approach we used to solve the problem on paths can be applied to trees, too.

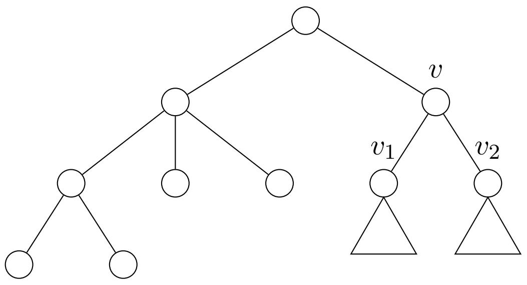


Figure 4.2: A rooted tree.

Let $G = (V, E, \gamma)$ be any tree. If G is not rooted choose any node as the root of G (see Definition 1.12) and continue. For each node $v \in V$ we call the corresponding subtree G_v rooted at v.

Given the wsp-subgraphs of all children $(v_1, v_2, \dots, v_k)$, we can calculate the solution for the parent node v. We save two solutions per node. By H^{in}_v we denote the wsp-subgraph of G_v including v itself and by H^{out}_v we denote the wsp-subgraph of G_v not including v. We start computing solutions for the leaves of the tree and go up level by level recursively reusing previous solutions until we reach the root and therefore obtain a solution for the complete tree.

Our dynamic programming algorithm starts by computing $H^{\text{in}}_v = v$ and $H^{\text{out}}_v = \emptyset$ for all leaves $v \in V$. In the next step the algorithm goes through the next level of the tree. It is sufficient to consider all children which would benefit the objective value $w(H^{\text{in}}_v)$ when adding them, i.e. all children with $w(H^{\text{in}}_v + (v, v_i)) > 0$, $i \in \{1, \dots, k\}$. Those children are then concatenated (see Lemma 3.5) to obtain H^{in}_v. The solution H^{out}_v is computed as follows: There are solutions for each child v_i, $i \in \{1, \dots, k\}$, namely $H^{\text{in}}_{v_i}$ and $H^{\text{out}}_{v_i}$. The algorithm compares their weights and takes the maximum.
As in the case for paths, Algorithm 4.2 can be used to solve min-wsp by computing the argmin instead of argmax. The same arguments hold.

Theorem 4.2. *Algorithm 4.2 computes a solution for the* Weighted Subgraph Problem *and* Weighted Induced Subgraph Problem *on trees correctly in* $\mathcal{O}(n)$ *time.*

Algorithm 4.2 MAX-WSP on trees

```
 1: procedure WSP($G$)
 2:     Let $Q = \emptyset$ be a queue
 3:     Add all leaves of $G$ to $Q$
 4:     while $Q \neq \emptyset$ do
 5:         Choose $v \in Q$
 6:         if $v$ is a leaf then
 7:             $H_v^{\text{in}} = v$
 8:             $H_v^{\text{out}} = \emptyset$
 9:         else
10:             Let $(v_1, v_2, \dots, v_k)$ be the children of $v$ in $G$
11:             $H_v^{\text{in}} = v + \sum_{\substack{i=1 \\ w(H_{v_i}^{\text{in}} + (v_i, v)) > 0}}^{k} H_{v_i}^{\text{in}} + (v_i, v)$
12:             $H_v^{\text{out}} = \operatorname{argmax} \Big\{ w(H_{v_i}^{\text{in}}), w(H_{v_i}^{\text{out}}) | i \in \{1, \dots, k\} \Big\}$
13:         Mark $v$ and remove it from $Q$
14:         if all siblings of $v$ are marked then
15:             Add parent of $v$ to $Q$
16:     return argmax $\Big\{ w(H_{v_n}^{\text{in}}), w(H_{v_n}^{\text{out}}) \Big\}$
```

Proof. Let G be a tree with root $r \in V$. We prove that when computing H_v^{in} and H_v^{out} as described in Algorithm 4.2 argmax or argmin of $\{w(H_v^{\text{in}}), w(H_v^{\text{out}})\}$ is the optimal solution for WSP on the subtree G_v of G for all $v \in V$ and, consequently, the correctness of the algorithm. In particular, for every $v \in V$ the graph H_v^{in} is the optimal solution for G_v which contains v and H_v^{out} is the optimal solution for G_v not including v.

For a leaf v of the tree, H_v^{in} is initialized as the graph just containing v and H_v^{out} is the empty set, which are both optimal for the weight function w. Now, suppose v is a parent node with children $(v_1, v_2, \dots, v_k)$, the solutions $H_{v_i}^{\text{in}}$ and $H_{v_i}^{\text{out}}$ are correctly computed and argmax or argmin of $\{w(H_{v_{i-1}}^{\text{in}}), w(H_{v_{i-1}}^{\text{out}})\}$ is the optimal solution for WSP on the subgraph G_{v_i} for all $i \in I = \{1, \dots, k\}$.

Then H_v^{out} is argmax or argmin of $\Big\{w(H_{v_i}^{\text{in}}), w(H_{v_i}^{\text{out}}) \colon i \in \{1, \dots, k\}\Big\}$, since by induction those are the optimal solutions for WSP on G_{v_i}, respectively and H_v^{out} is not allowed to contain the new vertex v.

Suppose H_v^{in} computed in the algorithm is not the optimal solution for G_v which contains v. Since we only consider the WSP-subgraphs whose weight $w(H_{v_i}^{\text{in}} + (v_i, v)) > 0$ is greater than zero and, therefore, benefits the value of the weight function, this implies that either the graph just containing v or at least one of the graphs $H_{v_i}^{\text{in}}$ extended by adding the vertex v and the edge (v_i, v) are not optimal. But this is a contradiction, since by induction $H_{v_i}^{\text{in}}$ is the optimal solution for G_{v_i} containing v_i for all $i \in \{1, \dots, k\}$ and a connected subgraph of G_v containing v either is v itself or has to contain at least one of its children v_i, $i \in \{1, \dots, k\}$.

The algorithm loops through each vertex exactly once. In each step the algorithm has to calculate the weight of three subgraphs for each child of the current node.

Using Lemma 3.5 this can be done in constant time and yields a total running time of $\mathcal{O}(n)$. □

4.3 Expanding to Series-Parallel Graphs

Another interesting class of graphs is the class of series-parallel graphs. We want to transfer our results from trees to solve the Weighted Subgraph Problem on series-parallel graphs. The idea is the same as for paths and trees. This time though, there is a difference between connected and induced subgraphs, since SP-graphs can have parallel edges and cycles. In this section we want to focus on the Weighted Subgraph Problem of finding connected subgraphs. We explain how to modify the dynamic program for solving the Weighted Induced Subgraph Problem, too. Note, that there is no need for all sub-solutions to be connected as long as the complete solution is connected. We focus on maximizing the weight function. The procedure can be slightly changed to solve the minimization variant of the problem.

Let G be any series-parallel graph with decomposition tree D. The idea is to work from the bottom to the top of the decomposition tree recursively reusing previous solutions until we reach the root and, as a result, obtain a solution for the graph G. For each node d in the decomposition tree we call the corresponding series-parallel graph G_d with source $s_d \in G_d$ and sink $t_d \in G_d$.

This time, we need to save 5 partial solutions per node: One only containing the source $s \in G_d$ denoted by H_d^s and one only containing the sink $t \in G_d$ called H_d^t. Another one containing none of the terminals, which we call $H_d^{\emptyset}$. All of those solutions need to be connected. Additionally, we need two subgraphs which contain both the source s_d and the sink t_d: $H_d^{s,t,C}$ which is already connected and $H_d^{s,t,N}$ which is not (yet) connected.

We begin by computing the solution for all leaves in the decomposition tree. Any leaf $d \in D$ corresponds to a graph G_d consisting of a single edge (s_d, t_d). This graph has no further nodes except the sink s_d and the source t_d. All five solutions are computed as follows:

$$\begin{aligned} H_d^s &= s_d, \\ H_d^t &= t_d, \\ H_d^{\emptyset} &= \emptyset, \\ H_d^{s,t,C} &= (s_d, t_d), \\ H_d^{s,t,N} &= \{s_d, t_d\}. \end{aligned} \tag{4.1}$$

Regard a node d of D with sons $d_1, d_2 \in D$. We explain how to compute the max-wsp solution for G_d given the solutions for G_{d_1} and G_{d_2}. We distinguish two cases.

Assume the node d specifies a parallel composition. Then G_d is the union of G_{d_1} and G_{d_2} where the sources s_{d_1}, s_{d_2} and the sinks t_{d_1}, t_{d_2} are identified. The new terminals are $s_d = s_{d_1} = s_{d_2}$ and $t_d = t_{d_1} = t_{d_2}$. Before we start, note that it holds that:

$$\begin{aligned} w(H_{d_1}^s), w(H_{d_2}^s) &\geq w(s_d), \\ w(H_{d_1}^t), w(H_{d_2}^t) &\geq w(t_d). \end{aligned} \tag{4.2}$$

The dynamic program works as follows:

$$
\begin{aligned}
H_d^s &= \operatorname{argmax}\Big\{w(H_{d_1}^s), w(H_{d_2}^s), w(H_{d_1}^s + H_{d_2}^s)\Big\},\\
H_d^t &= \operatorname{argmax}\Big\{w(H_{d_1}^t), w(H_{d_2}^t), w(H_{d_1}^t + H_{d_2}^t)\Big\},\\
H_d^\emptyset &= \operatorname{argmax}\Big\{w(H_{d_1}^\emptyset), w(H_{d_2}^\emptyset)\Big\},\\
H_d^{s,t,C} &= \operatorname{argmax}\Big\{w(H_{d_1}^{s,t,C}), w(H_{d_2}^{s,t,C}), w(H_{d_1}^{s,t,C} + H_{d_1}^{s,t,C}),\\
&\qquad w(H_{d_1}^{s,t,C} + H_{d_2}^s), w(H_{d_1}^{s,t,C} + H_{d_2}^t),\\
&\qquad w(H_{d_1}^s + H_{d_2}^{s,t,C}), w(H_{d_1}^t + H_{d_2}^{s,t,C}),\\
&\qquad w(H_{d_1}^{s,t,N} + H_{d_2}^{s,t,C}), w(H_{d_1}^{s,t,C} + H_{d_2}^{s,t,N})\Big\},\\
H_d^{s,t,N} &= \operatorname{argmax}\Big\{w(H_{d_1}^{s,t,N}), w(H_{d_2}^{s,t,N}), w(H_{d_1}^{s,t,N} + H_{d_2}^{s,t,N}),\\
&\qquad w(H_{d_1}^{s,t,N} + H_{d_2}^s), w(H_{d_1}^{s,t,N} + H_{d_2}^t), w(H_{d_1}^s + H_{d_2}^{s,t,N}),\\
&\qquad w(H_{d_1}^t + H_{d_2}^{s,t,N})\Big\}.
\end{aligned}
\tag{4.3}
$$

Assume now the node d specifies a series composition. Then G_d is the disjoint union of G_{d_1} and G_{d_2}. Further, suppose the terminals t_{d_1} and s_{d_2} are identified. The new terminals are $s_d = s_{d_1}$ and $t_d = t_{d_2}$. The following holds:

$$
\begin{aligned}
w(H_{d_1}^{s,t,C} + H_{d_2}^s) &\geq w(H_{d_1}^{s,t,C}),\\
w(H_{d_1}^t + H_{d_2}^{s,t,C}) &\geq w(H_{d_2}^{s,t,C}),\\
w(H_{d_1}^s + H_{d_2}^t)\} &\geq w(H_{d_1}^t), w(H_{d_2}^s).
\end{aligned}
\tag{4.4}
$$

In this case, the solutions can be extended as follows:

$$
\begin{aligned}
H_d^s &= \operatorname{argmax}\Big\{w(H_{d_1}^s), w(H_{d_1}^{s,t,C} + H_{d_2}^s)\Big\},\\
H_d^t &= \operatorname{argmax}\Big\{w(H_{d_2}^t), w(H_{d_1}^t + H_{d_2}^{s,t,C})\Big\},\\
H_d^\emptyset &= \operatorname{argmax}\Big\{w(H_{d_1}^t + H_{d_2}^s), w(H_{d_1}^\emptyset), w(H_{d_2}^\emptyset)\Big\},\\
H_d^{s,t,C} &= H_{d_1}^{s,t,C} + H_{d_2}^{s,t,C}\\
H_d^{s,t,N} &= \operatorname{argmax}\Big\{w(H_{d_1}^{s,t,N} + H_{d_2}^{s,t,C}), w(H_{d_2}^{s,t,C} + H_{d_2}^{s,t,N}), w(H_{d_1}^s + H_{d_2}^t),\\
&\qquad w(H_{d_1}^{s,t,C} + H_{d_2}^t), w(H_{d_1}^s + H_{d_2}^{s,t,C})\Big\}.
\end{aligned}
\tag{4.5}
$$

Changing the procedure to minimize the weight function, again, one simply needs to compute argmin instead of argmax. The dynamic program works as in the maximization case.

Theorem 4.3. *Let G be a series-parallel graph. Given the decomposition tree D of G we can calculate a solution for the* Weighted Subgraph Problem *on G correctly in $\mathcal{O}(|D|) = \mathcal{O}(m)$ time, as the decomposition tree has $2m - 1$ nodes.*

Proof. Let G be a series-parallel graph with decomposition tree D. We regard the procedure from above. We need to show that for each node $d \in D$ when computing the solution sets $H_d^s, H_d^t, H_d^\emptyset, H_d^{s,t,C}$, and $H_d^{s,t,N}$ as described in the algorithm argmin or argmax of $\{w(H_d^s), w(H_d^t), w(H_d^\emptyset), w(H_d^{s,t,C}), w(H_d^{s,t,N})\}$ is the optimal solution for WSP on the subgraph G_d of G and therefore the correctness of the algorithm. In particular, for every $d \in D$ the graphs H_d^s, H_d^t, $H_d^\emptyset$, $H_d^{s,t,C}$ and $H_d^{s,t,N}$ are optimal solutions for G_d while containing or not containing the specified terminals.

For the leaves this is obvious. So regard a node d with sons d_1 and d_2, suppose the solutions $H_{d_i}^s$, $H_{d_i}^t$, $H_{d_i}^\emptyset$, $H_{d_i}^{s,t,C}$ and $H_{d_i}^{s,t,N}$ are correctly computed and the argmin or argmax of $\{w(H_{d_i}^s), w(H_{d_i}^t), w(H_{d_i}^\emptyset), w(H_{d_i}^{s,t,C}), w(H_{d_i}^{s,t,N})\}$ is the optimal solution for WSP on the subgraph G_{d_i} for all $i \in \{1, 2\}$. We distinguish two cases.

Assume d specifies a parallel composition.

- H_d^s has to contain the source s. Therefore the candidates for a solution have to contain s, as well and H_d^s is a combination of those candidates. Suppose H_d^s computed in the algorithm is not the optimal solution for G_d which contains s. But due to (4.2) this means either $H_{d_1}^s$, $H_{d_2}^s$ or $H_{d_1}^s + H_{d_2}^s$ which are the only candidates are not optimal for their subgraphs which is a contradiction. The same argumentation holds for H_d^t.

- $H_d^\emptyset$ is not allowed to contain s or t. Since d specifies a parallel composition it has to be either $H_{d_1}^\emptyset$ or $H_{d_2}^\emptyset$. A combination is not possible, since $H_d^\emptyset$ has to be connected.

- Both $H_{d_i}^{s,t,C}$ and $H_{d_i}^{s,t,N}$ have to be combinations of previous solutions. Suppose they are not optimal. Then they would contain nodes or edges not included in any subsolutions. But then the corresponding subsolution could be improved which is a contradiction to its optimality.

Now, assume d specifies a series composition. The new terminals are $s_d = s_{d_1}$ and $t_d = t_{d_2}$.

- H_d^s has to contain the new source s_d. Therefore the candidates for a solution have to contain s_{d_1} and H_d^s is a combination of those candidates. Suppose H_d^s computed in the algorithm is not the optimal solution for G_d which contains s. But due to (4.4) this means either $H_{d_1}^s$ or $H_{d_1}^{s,t,C} + H_{d_2}^s$ which are the only candidates are not optimal for their subgraphs which is a contradiction. The same argumentation holds for H_d^t.

- $H_d^\emptyset$ is not allowed to contain s or t. Since d specifies a series composition it has to be either $H_{d_1}^\emptyset$, $H_{d_2}^\emptyset$ or $H_{d_1}^t + H_{d_2}^s$.

- The same arguments as for the parallel composition hold.

We now regard the complexity of the procedure. The decomposition tree D has $2m-1$ vertices. The algorithm loops through each vertex exactly once. In each step the algorithm has to calculate the weight of at most 14 subgraphs. Using Lemma 3.5 this can be done in constant time and yields a total running time of $\mathcal{O}(|D|) = \mathcal{O}(m)$. □

We describe how to modify the dynamic program for solving the WEIGHTED INDUCED SUBGRAPH PROBLEM on series-parallel graphs. Initializing the sets $H_d^s, H_d^t, H_d^\emptyset$, and $H_d^{s,t,C}$ for primitive graphs works as in (4.1). $H_d^{s,t,N}$ has to initialized as the empty set for all primitive graphs.

$$H_d^{s,t,N} = \emptyset. \tag{4.6}$$

In case of a series composition the sets can be extended as in (4.5). Otherwise, in a parallel composition, we need to adjust the update rule for $H_d^{s,t,C}$ and $H_d^{s,t,N}$ in (4.3). If there are edges from s_{d_1} to t_{d_1} in G_{d_1} we need to include those edges to $H_{d_2}^{s,t,C}$, since they are induced edges between s_d and t_d in the new graph G_d. Vice versa, if there are edges from s_{d_2} to t_{d_2} in G_{d_2} they have to be added to $H_{d_1}^{s,t,C}$.

By E_{d_1} we denote the set of edges between s_{d_1} to t_{d_1} (analogous: E_{d_2}). The new rules for the Weighted Induced Subgraph Problem are:

$$\begin{aligned} H_d^{s,t,C} = \operatorname{argmax} \Big\{ & w(H_{d_1}^{s,t,C} + E_{d_2}), w(H_{d_2}^{s,t,C} + E_{d_1}), w(H_{d_1}^{s,t,C} + H_{d_1}^{s,t,C}), \\ & w(H_{d_1}^{s,t,C} + E_{d_2} + H_{d_2}^{s}), w(H_{d_1}^{s,t,C} + E_{d_2} + H_{d_2}^{t}), \\ & w(H_{d_1}^{s} + H_{d_2}^{s,t,C} + E_{d_1}), w(H_{d_1}^{t} + H_{d_2}^{s,t,C} + E_{d_1}), \\ & w(H_{d_1}^{s,t,N} + H_{d_2}^{s,t,C} + E_{d_1}), w(H_{d_1}^{s,t,C} + E_{d_2} + H_{d_2}^{s,t,N}) \Big\}. \end{aligned} \tag{4.7}$$

If either $E_{d_1} \neq \emptyset$ or $E_{d_2} \neq \emptyset$ this means there is a direct edge between s_d and t_d. So it is not possible to obtain an induced subgraph containing both these vertices which is not connected. In that case we set:

$$H_d^{s,t,N} = \emptyset. \tag{4.8}$$

Lemma 4.4. *Using rules* (4.6), (4.7) *and* (4.8) *for updating* $H_d^{s,t,C}$ *the procedure from above returns an induced subgraph.*

Proof. Suppose the procedure returns a subgraph H of G which is not an induced subgraph of G, that is there exists an edge $e = (u, v) \in G$ with $u, v \in V(H)$, but $e \notin E(H)$. Then at some point in the procedure u was the source and v was the sink of a graph G_{d_1} or vice versa. When combing this graph with another graph G_{d_2}, rule (4.7) ensures, that the edge e has been added to all connected subgraphs containing both u and v. Additionally, due to (4.8) we "delete" subgraphs which would no longer be induced without adding edge e. Adding edge e would make them connected which is not intended for that subgraph. □

Corollary 4.5. *Let G be a series-parallel graph. Given the decomposition tree D of G we can calculate a solution for the* Weighted Induced Subgraph Problem *on G correctly in $\mathcal{O}(|D|) = \mathcal{O}(m)$ time.*

Proof. This follows from Theorem 4.3 and Lemma 4.4. □

4.4 Generalizing to Decomposable Graphs

In a next step we want to generalize the procedure to decomposable graphs.

Let Γ be a class of decomposable graphs, let l be the maximum number of terminals associated with any graph in Γ. For simplicity of notation we assume that all graphs in Γ have exactly l terminals.

The idea of the procedure is the same as in the procedure for series-parallel graphs. For a given graph $G \in \Gamma$ we work on the decomposition tree D_G from the bottom to the top.

For each node $d \in D_G$ we call the corresponding partial graph G_d with terminals $T_d \subset V(G_d)$.

For every subset K of $L := \{1, \dots, l\}$, we have to consider the terminals $T_d^K := \{t_k^d \colon k \in K\} \subseteq T_d$. In addition, we need to calculate solutions for each possible combination of connected components containing these terminals. This can be achieved by looking at all partitions $P_1^K, \dots, P_{B_{|K|}}^K$ of the set K. The corresponding solutions are denoted by H_d^{K,P_i^K}, for all $i \in \{1, \dots, B_{|K|}\}$.

We begin by computing the solution for all leaves in the decomposition tree. Any leaf d corresponds to a primitive graph G_d. Since the number of primitive graphs is finite, the size of the primitive graphs is bounded by a constant. We can calculate all induced subgraphs of each primitive graph and thus compute all needed sets for the leaves in constant time.

Regard a node d of D with sons $d_1, d_2 \in D$. We explain how to compute the MAX-WSP solution for G_d given the solutions for G_{d_1} and G_{d_2}. The terminals T_d of G_d are a subset of the terminals of the composing graphs: $T_d \subset (T_{d_1} \cup T_{d_2})$. We need to compute $H_d^{K,P}$ for all partitions $P \in \{P_1^K, \dots, P_{B_{|K|}}^K\}$ of K and for all sets $K \subseteq L$. Using the composition rule specified by d, the terminals are either identified with each other, a single edge is added between them or they are left untouched.

Fix $K \subseteq L$ and $P \in \{P_1^K, \dots, P_{B_{|K|}}^K\}$. We describe how to compute $H_d^{K,P}$ containing the terminals T_d^K and the connected components specified by P. Assume the partition P has r sets: $P = \{X_1, X_2, \dots, X_r\}$. For each set X_i we have terminals that were identified X_i^{id}, that an edge was added to X_i^e and which were untouched X_i^u. We have to construct the connected components of $H_d^{K,P}$ by selecting the corresponding $H_{d_1}^{K_1,P_1}$ and $H_{d_2}^{K_2,P_2}$ and possibly selecting the new edges for our new solution. Terminals in X_i^{id} are in the same connected component now, for the ones in X_i^e we can choose to select the edge to connect their components and the ones in X_i^u stay in their component if no other node or selected edge connects their components. We need to look at all $H_{d_1}^{K_1,P_1}$ and $H_{d_1}^{K_2,P_2}$ with $K = \bar{K_1} \cup \bar{K_2}$ to avoid selecting terminals form $L \setminus T_d^K$. In addition, the union $\bar{P_1} \cup \bar{P_2}$ either has to equal P or has to be a refinement P, where $\bar{P_1} = \{X \cap T_d^K | X \in P_1\}$ and $\bar{P_2} = \{X \cap T_d^K | X \in P_2\}$, $\bar{K_1} = K_1 \cap T_d^K$ and $\bar{K_2} = K_2 \cap T_d^K$ are restrictions of P_1, P_2, K_1 and K_2, respectively. Write

$$\bar{P_1} \cup \bar{P_2} = \{Y_{11}, Y_{12}, \dots, Y_{1a_1}, Y_{21}, Y_{22}, \dots, Y_{2a_2}, Y_{31}, \dots Y_{ra_r}\},$$

where $Y_{ij}, j \in \{1, \dots a_i\}$ specifies the refinement of $X_i \in P$ for all $i \in \{1, \dots, r\}$. Let $\mathcal{H}_d^{K,P}$ be the set of all possible candidates for the solution $H_d^{K,P}$. Add all $H_{d_1}^{K_1,P_1}$ with $K_1 \cap T_d^K = K$ and $\{X \cap T_d^K | X \in P_1\} = P$ to $\mathcal{H}_d^{K,P}$. Analogously for d_2. Combined solutions for $H_{d_1}^{K_1,P_1}$ and $H_{d_2}^{K_2,P_2}$ are valid if by identifying terminals or adding edges according to the composition rule the components $Y_{ij}, j \in \{1, \dots a_i\}$ become the component X_i for all $i \in \{1, \dots, r\}$. If edges were added to connect the components, call the set of added edges $\tilde{E}$, append these, and add the solution candidate $H_{d_1}^{K_1,P_1} + H_{d_2}^{K_2,P_2} + \tilde{E}$. Otherwise, add $H_{d_1}^{K_1,P_1} + H_{d_2}^{K_2,P_2}$ to $\mathcal{H}_d^{K,P}$. Then calculate the solution with

$$H_d^{K,P} = \operatorname{argmax}\{w(H) \colon H \in \mathcal{H}_d^{K,P}\}.$$

Theorem 4.6. *Let Γ be a class of decomposable graphs and let $G \in \Gamma$ be a graph of that class. Given the decomposition tree D of G we can calculate* WEIGHTED SUBGRAPH PROBLEM *on G correctly in polynomial time.*

Proof. We regard the procedure from above. Let

$$\mathcal{H}_d := \{H_d^{K,P}, K \subseteq L \text{ where } P \text{ is a permutation of } K\}.$$

We need to show that for each node $d \in D$ when computing the solution set $H_d^{K,P}$ for all $K \subseteq L$ and $i \in \{1, \dots, B_{|K|}\}$ as described in the procedure argmin or argmax of $\{w(H) \colon H \in \mathcal{H}_d\}$ is the optimal solution for WSP on the partial graph G_d of G and consequently, the correctness of the algorithm. In particular, for every $d \in D$, the graph $H_d^{K,P}$ is an optimal solution for G_d containing the terminals T_d^K and having the connected components specified in P.

For the leaves which are primitive graphs this is obvious. Any leaf d corresponds to a primitive graph G_d. Since the number of primitive graphs is finite, the size of the primitive graphs is bounded by a constant, all subgraphs of each primitive graph and thus the sets H_d^{K,P_i^K} with $K \subseteq L$ and $i \in \{1, \dots, B_{|K|}\}$ can be computed in constant time.

So regard a node d with sons d_1 and d_2, suppose the solutions $H_{d_j}^{K,P_i^K}$ for all $K \subseteq L$ and $i \in \{1, \dots, B_{|K|}\}$ are correctly computed and argmin or argmax of $\{w(H) \colon H \in \mathcal{H}_{d_j}\}$ is the optimal solution for WSP on the subgraph G_{d_j} for $j \in \{1, 2\}$. Assume $H_d^{K,P}$ is not is not optimal for G_d containing the terminals T_d^K and having the connected components specified in P. We distinguish multiple cases.

First, assume $H_d^{K,P}$ does not contain all terminals T_d^K or has additional terminals. This is not possible since we only looked at $H_{d_1}^{K_1,P_1}$ and $H_{d_1}^{K_2,P_2}$ with $K = \bar{K}_1 \cup \bar{K}_2$ with $\bar{K}_1 = K_1 \cap T_d^K$ and $\bar{K}_2 = K_2 \cap T_d^K$. None of these subgraphs contain additional terminals and their composition has all terminals T_d^K.

Now, suppose $H_d^{K,P}$ does not have the connected components specified in P. Similar to before, this does not happen due to the construction of $H_d^{K,P}$.

Finally, assume there is a subgraph $\tilde{H}$ containing the terminals T_d^K, having the connected components specified in P and $w(\tilde{H} > w(H_d^{K,P}))$, that is $\tilde{H}$ is an optimal solution. Then $\tilde{H}$ is not in $\mathcal{H}_d$. Since $\tilde{H}$ is an optimal solution on G_d, it has to be either a subgraph of G_{d_1}, G_{d_2}, or a combination of two subgraphs.

In the former case, it would be an optimal solution on that subgraph meaning $\tilde{H} = H_{d_j}^{K_j,P_j}$ for $j = 1$ or $j = 2$. But then since $\tilde{H}$ contains the terminals T_d^K and has the connected components specified in P it would have been added to $\mathcal{H}_d$. In the latter case each of its subgraphs is an optimal solution on G_{d_j} containing the terminals $T_{d_j}^K$ meaning it would have been added to $\mathcal{H}_{d_j}$ for $j = 1$ or $j = 2$, respectively. But since the combination of both these subgraphs fulfills all criteria, it would have been added to $\mathcal{H}_d$ and thus $w(\tilde{H} \leq w(H_d^{K,P}))$.

We now regard the complexity of the procedure. Let $|D|$ denote the size of the decomposition tree D. The algorithm loops through each vertex exactly once. There are $\binom{l}{k}$ subsets of size k of the set of all terminals T with $|T| = l$ and 2^l subsets in total. A set of size k has B_k unique partitions (see Definition A.2 in the appendix). Therefore the algorithm has to calculate the weight of at most $\sum_{k=1}^{l} \binom{l}{k} \cdot B_k$ subgraphs in each step. Since l is fixed for all graphs in G, and calculating the weight for each subgraph can be done in constant time (see Lemma 3.5). In total, this yields a total running time of $\mathcal{O}(|D| \cdot \sum_{k=1}^{l} \binom{l}{k} \cdot B_k)$. □

4.5 Solving the Rooted Problem

We want to finish this chapter by modifying the dynamic program to solve the rooted version of the problem (see Definition 3.3). We briefly present how this is done.

For paths and trees this can be achieved by changing the return statement of the dynamic program (Algorithm 4.2, line 8) to return the following instead:

$$\textbf{return } H^{\text{in}}_{v_n}. \tag{4.9}$$

Theorem 4.7. *Using return statement* (4.9) *in Algorithm 4.2 computes a solution for the* Rooted Weighted Subgraph Problem *and* Rooted Weighted Induced Subgraph Problem *on trees correctly in $\mathcal{O}(n)$ time.*

Proof. This stems from the fact, that we can choose any node in the tree as the root of the tree. Let T be any tree and $r \in V$ be a vertex. Choosing r as the root, we have $r \in H^{\text{in}}_{v_n}$ (since $v_n = r$) and due to Theorem 4.2 the modified dynamic program yields an optimal solution for wsp(G) containing r, that is an optimal solution for rwsp(G, r). □

Since every path is a tree, Algorithm 4.2 can be used to solve the Rooted Weighted Subgraph Problem and Rooted Weighted Induced Subgraph Problem on paths, too.

For series-parallel graphs, subgraphs containing r need to be favored over subgraphs not containing r. In each step of the algorithm the extension of $H^s_d, H^t_d, H^{\emptyset}_d, H^{s,t,C}_d$ and $H^{s,t,N}_d$ is changed as follows.

Let $\mathcal{H}$ denote the set of subgraphs which are eligible for extending one of the solutions H (according to (4.3) and (4.5)). If there exists a graph $\bar{H} \in \mathcal{H}$ with $r \in \bar{H}$ then

$$H = \operatorname{argmax}\left\{w(\bar{H})\colon \bar{H} \in \mathcal{H} \text{ with } r \in \bar{H}\right\}. \tag{4.10}$$

Otherwise we extend the solution as previously:

$$H = \operatorname{argmax}\left\{w(\bar{H})\colon \bar{H} \in \mathcal{H}\right\}. \tag{4.11}$$

Theorem 4.8. *Using the update rules* (4.10) *and* (4.11) *described above the procedure presented in Section 4.3 computes a solution for the* Rooted Weighted Subgraph Problem *and* Rooted Weighted Induced Subgraph Problem *on series-parallel graphs correctly in $\mathcal{O}(|D|) =$
$OO(m)$ time.*

Proof. Let G be any series-parallel graph and $r \in V$ be a vertex. Update rules (4.10) and (4.11) favor solutions which contain r. Using Theorem 4.3 the claim follows. □

Chapter 5

Densest k-Subgraph Problem

A similar problem to the Weighted Subgraph Problem is the Densest k-Subgraph (dks) maximization problem, of finding the densest k-vertex subgraph of a given graph. This chapter is based on [FS97]. The goal is to examine whether one can use the techniques for solving or approximating the Weighted Subgraph Problem and its variations.

Densest k-Subgraph Problem	
Given:	An undirected graph $G = (V, E, \gamma)$ and a parameter k.
Problem:	Find a subgraph G^* of G, induced on k vertices, such that G^* is of maximum density.

We denote the density of G^* among all the subgraphs by $d^*(G, k)$. It can be shown that the Densest k-Subgraph problem is $\mathcal{NP}$-complete.

In [FS97] Feige, Kortsarz, and Peleg developed two polynomial time approximation algorithms for Densest k-Subgraph. They showed the following two theorems:

Theorem 5.1. *There is a polynomial time algorithm A that approximates* Densest k-Subgraph *within a factor of $2\sqrt[3]{n}$. That is, for every graph G and every $1 \leq k \leq n, A(G,k) \geq \frac{d^*(G,k)}{2\sqrt[3]{n}}$.*

Theorem 5.2. *There is a polynomial time algorithm B that approximates* Densest k-Subgraph *within a factor of $n^{\frac{1}{3}-\varepsilon}$, for some $\varepsilon > 0$. That is, for every graph G and every $1 \leq k \leq n, B(G,k) \geq \frac{d^*(G,k)}{n^{\frac{1}{3}-\varepsilon}}$*

Unfortunately, simply computing the densest subgraph of a graph is not a good approximation for the Weighted k-Subgraph Problem (or the Weighted Induced k-Subgraph Problem) as the following example illustrates:

Example 5.3. Figure 5.1 shows that we cannot approximate max-wiksp by solving dks. The densest 4-subgraph H_{DKS} of G (see 5.1b) has density

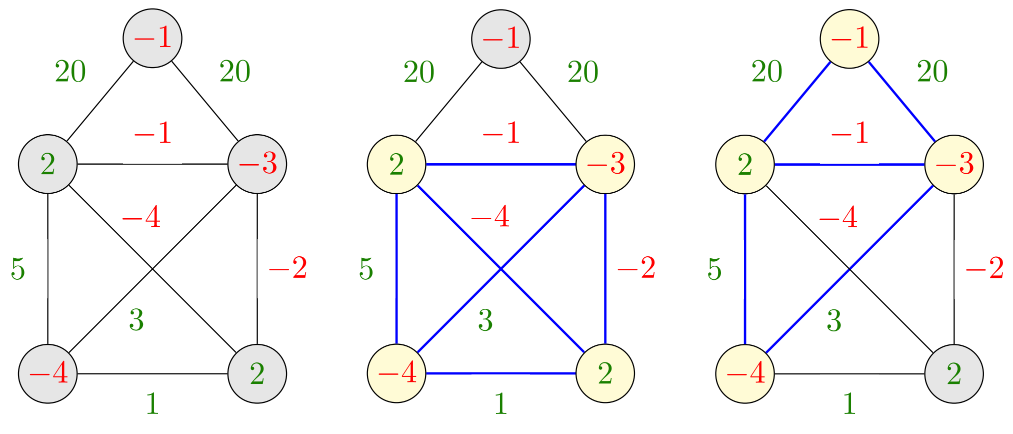
$$d_{H_{\text{DKS}}} = 2\frac{|E(H_{\text{DKS}})|}{|V(H_{\text{DKS}})|} = \frac{12}{4} = 3$$

and weight

$$w(H_{\text{DKS}}) = -1.$$

On the contrary the maximum weigthed 4-subgraph of G (see 5.1c) has density

$$d_{H_{\text{WKSP}}} = 2\frac{|E(H_{\text{WKSP}})|}{|V(H_{\text{WKSP}})|} = \frac{10}{4} < 3$$



(a) Input graph G. (b) Densest 4-subgraph of G. (c) A maximum weighted 4-subgraph of G.

Figure 5.1: Difference between densest 4-subgraph and maximum weighted 4-subgraph.

and weight

$$w(H_{\text{DKS}}) = 41 > -1.$$

It is easy to see that by increasing the weight on the edges not contained in H_{DKS} or reducing the weights of the nodes and egdes contained in H_{DKS} the gap becomes arbitrarily large.

There is a weighted version of the Densest k-Subgraph problem, where edges have nonnegative weights, and the goal is to find the k-vertex induced subgraph with the maximum total weight of edges.

Feige, Kortsarz and Peleg sketched in [FS97] how to reduce the weighted problem to the unweighted Densest k-Subgraph problem with a loss of at most $\mathcal{O}(\log n)$ in the approximation ratio.

Procedure 5.4 (Solving the weighted Densest k-Subgraph problem)**.**

1. Scale edge weights such that the maximum edge weight is n^2.
2. Round up each edge weight to the nearest (nonnegative) power of 2.
3. Solve $2 \log n$ DKS problems, one for each edge weight (with all other edges removed).
4. Select the best of the $\mathcal{O}(\log n)$ solutions.

Corollary 5.5. *There is a polynomial time algorithm C that approximates the weighted* Densest k-Subgraph *problem within a factor of $\alpha \cdot \log n \cdot n^{\frac{1}{3}-\varepsilon}$, for some $\varepsilon > 0$ and $\alpha \in \mathbb{N}$.*

Proof. This follows from Theorem 5.2 and Procedure 5.4. □

In general, we cannot use this procedure in the above from to approximate the Weighted Induced k-Subgraph Problem since we have to consider node weights and negative edge weights. For special cases where edge weights are non-negative and node weights are significantly smaller than the absolute values of the edge weights one could use procedure 5.4 to approximately solve the Weighted Induced k-Subgraph Problem on those instances.

In conclusion, using an approximation algorithm for the weighted Densest k-Subgraph problem to approximately solve the Weighted Induced k-Subgraph Problem is only feasible for a small subset of instances. For general instances this procedure is either not employable or would result in a very bad approximation ratio, especially if the absolute values of node weights are larger than the edge weights.

Chapter 6

Approximations and Heuristics

This chapter is dedicated to solving the Weighted Subgraph Problem approximately or heuristically. The goal of this chapter is to determine whether Weighted Subgraph Problem can be approximated and, if so, finding a procedure with provable ratio. Throughout this chapter several procedures are presented. In Chapter 8 the practical relevance of these procedures is determined.

In Section 6.1 we show that approximating the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem within a constant factor is $\mathcal{NP}$-complete. Subsequently, this hardness is shown to hold even for a budget-constrained version of the problem in Section 6.2. Hereafter, Section 6.3 presents a preprocessing scheme for reducing instances of the Weighted Subgraph Problem and Section 6.4 proposes a postprocessing procedure to improve sub-optimal or heuristic solutions. We complete this chapter by proposing two heuristic procedures in Sections 6.5 and 6.6.

6.1 Hardness of Approximation

Since the Weighted Subgraph Problem is $\mathcal{NP}$-complete the next natural step is to try to solve the problem approximately.

Unfortunately it turns out that the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem are $\mathcal{NP}$-complete to approximate by a constant factor. We start by proving the approximation hardness for the rooted version of the problem and will generalize the result for the unrooted problem.

Theorem 6.1. *Given an undirected graph with node weights $d\colon V \to \mathbb{R}$, edge weights $c\colon E \to \mathbb{R}$, and a root node $r \in V$, the* Rooted Weighted Subgraph Problem *and the* Rooted Weighted Induced Subgraph Problem *are $\mathcal{NP}$-complete to approximate within a constant factor. Moreover, this hardness result holds even under Assumption 3.6.*

For the proof of Theorem 6.1 we use the Satisfiability Problem. Before defining the problem we need some notations: A *literal* is either a variable, called positive literal, or the negation of a variable, called negative literal. A *clause* is a disjunction of literals (or a single literal). A formula is in *conjunctive normal form* if it is a conjunction of clauses (or a single clause). The Satisfiability Problem is the problem of deciding whether a given Boolean formula in conjunctive normal form has an assignment that makes the formula evaluate to true, that is the formula is satisfiable.

Satisfiability Problem (sat)	
Given:	A set of literals $(x_1, \dots, x_n)$ and clauses $(C_1, \dots, C_m)$ that comprise a formula f.
Question:	Is there an assignment to $x_1, \dots, x_n$ such that $f(x_1, \dots, x_n) = 1$?

Proof of Theorem 6.1. For this proof we consider the maximization version of the problem. Consequently, let $0 < \alpha < 1$ be an arbitrary constant. Due to Lemma 3.11 the result also holds for the minimization variant.

Let $(x_1, \dots, x_n, C_1, \dots, C_m)$ be an arbitrary instance of SAT. For every literal $x_i, 1 \leq i \leq n$ we add a new clause $C_{m+i} = (x_i \vee \bar{x}_i)$ to the formula. This does not change the satisfiability of the formula since these clauses are trivially satisfied. We obtain a modified instance $(x_1, \dots, x_n, C_1, \dots, C_{m'})$ with $m' = m + n$. We describe how to construct an equivalent graph. The nodes are arranged in four levels. Let $M > n+1$.

- The first level contains the root vertex r. It is given weight 0.
- On the second level there is a node r' with weight 0 that is connected to r via an edge with weight $-m'M + n + 1$.
- On the third level there are $2n$ vertices. One for each literal x_i and $\bar{x}_i$. All of them with weight -1 and connected to r' via an edge with weight 0.
- The fourth level contains m' vertices, each corresponding to a clause and with weight M. These clause-vertices are connected to the literal-vertices of level three that they contain.

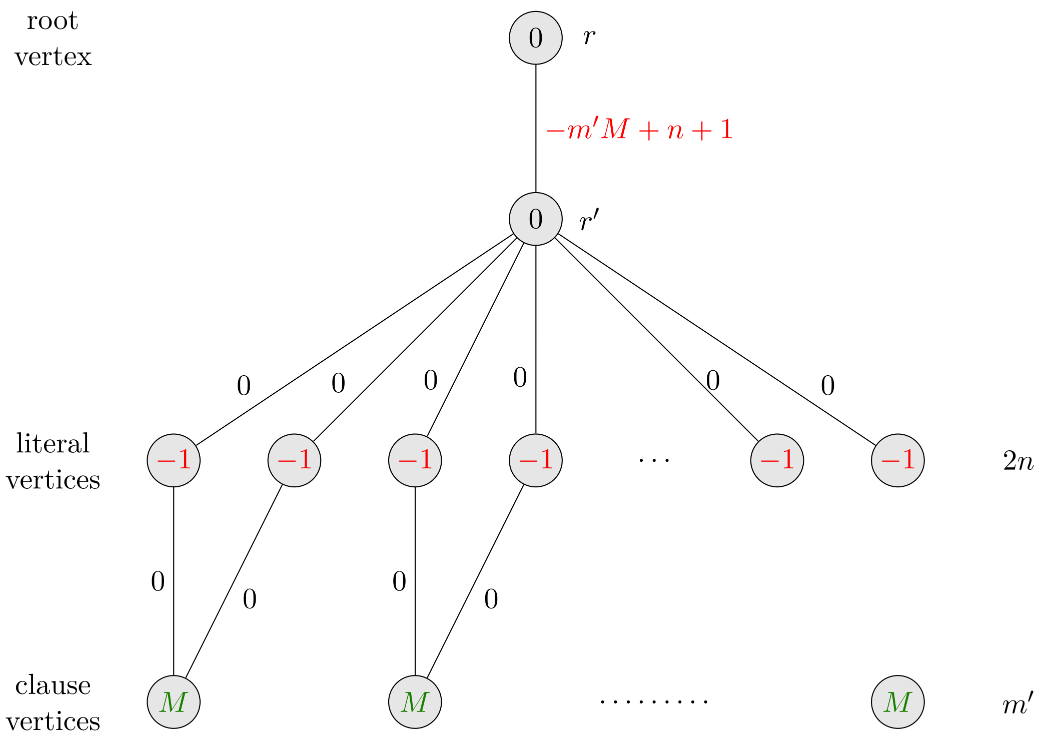


Figure 6.1: Construction of the graph G for reducing an instance of SAT to an instance of MAX-RWSP.

Figure 6.1 shows the construction of G. Since G is a tree, MAX-WSP and MAX-WISP are equivalent (see Lemma 3.12). The trivial solution just containing r has weight 0. To get a better value we need to include the edge (r, r') and all clause-vertices on level four. Since $M > n+1$ including only $m'-1$ clause vertices results in a weight at most $-m'M+n+1+(m'-1)M < -M+n+1 < 0$. In order to reach clause C_{m+i} one of the literal-vertices on level three corresponding to x_i and $\bar{x_i}$ must be included for each $1 \leq i \leq n$. Consequently, at least n literal-vertices are included. Also, to improve the value of the weight function at most n vertices on level three can be included obtaining a subgraph with weight $-m'M + n + 1 - n + m'M = 1$. If $n + 1$ vertices are included then the weight would be at most $-m'M + n + 1 - n + 1 + M'M = 0$ which is not an improvement. Thus, a solution with weight larger than 0 corresponds to a satisfying assignment and has weight 1.

Suppose there exists a polynomial time α-approximation algorithm APPROX for RWSP or RWISP. We now claim, that the formula is satisfiable if and only if APPROX returns a solution with weight larger than α.

If the formula is satisfiable then the optimal solution has weight 1 so APPROX would return a solution with weight at least $\alpha \cdot 1 > 0$. Conversely, if the formula is not satisfiable the optimal solution has weight 0. Then APPROX would return a solution with weight at most (exactly) 0.

In particular, any polynomial time α-approximation algorithm would be able to distinguish the two cases, and thus to solve SAT in polynomial time. Thus α-approximating RWSP and RWISP is $\mathcal{NP}$-complete.

To obtain a hardness result for the restricted case under Assumption 3.6, we modify the reduction as follows. The nodes are arranged in six levels.

- We add an level between r' and the literal-vertices containing a vertex r'' with weight $-m'M + n + 1$. An edge (r', r'') with weight 0 is added.
- The edges between all literal-vertices and r' are removed. Instead each literal vertex is connected to r'' via an zero-weighted edge.
- The weight of the clause-vertices is changed to 0.
- A level below the forth level is added containing yet another vertex for each clause. All of those is given weight 0 and they are connected to the corresponding clause-vertex on level four via an edge of weight M.

All of the edges in this reduction have non-negative weight and all vertices have non-positive weight. It is easy to see that the argumentation above works as it did in the original reduction. □

Continuing, we use the same proof idea and built upon the construction created to obtain a hardness result for approximating the WEIGHTED SUBGRAPH PROBLEM and the WEIGHTED INDUCED SUBGRAPH PROBLEM.

Theorem 6.2. *Given an undirected graph with node weights $d\colon V \to \mathbb{R}$ and edge weights $c\colon E \to \mathbb{R}$, the* WEIGHTED SUBGRAPH PROBLEM *and the* WEIGHTED INDUCED SUBGRAPH PROBLEM *are $\mathcal{NP}$-complete to approximate within a constant factor. Moreover, this hardness result holds even under Assumption 3.6.*

Proof. As in the previous proof, we consider the maximization version and let $0 < \alpha < 1$ be an arbitrary constant.

Let $f = (x_1, \dots, x_n, C_1, \dots, C_m)$ be an arbitrary instance of SAT. We expand the formula and construct a graph G' as in the proof of Theorem 6.1 (either for RWSP or RWISP). This construction is now extended to a graph G as follows. Let $K > \frac{1}{\alpha} \cdot Mm'$.

- Create K copies of G'.
- Connect the root nodes of all K copies with each other by an edge with weight 0, creating a 0-clique.

The graph G is shown in Figure 6.2. Note, that all vertices and edges added were given weight 0. Thus, this construction works for both the general as well as the restricted version of WSP or WISP.

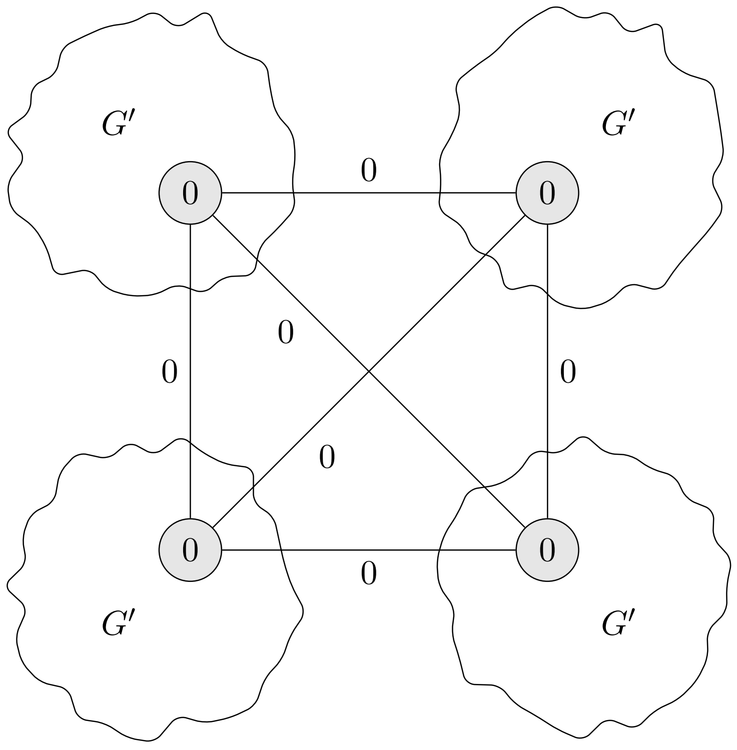


Figure 6.2: Construction of the graph G for reducing an instance of SAT to an instance of MAX-WSP.

From Theorem 6.1 we know that an optimal solution H' of G' which contains the root vertex r has weight 1 if and only if the formula is satisfiable and weight 0 else. If a subgraph of G' does not have to contain its root r, its weight is at most $m'M$ by collecting all weight at the bottom level of the graph. More precisely it is at most $m'M - n$ since it has to collect at least n literal-vertices on level 3 to connect all clause-vertices. Summarizing, if $H' = \text{MAX-WSP}(G')$ then:

$$\begin{aligned} w(H') &= 1, && \text{if } r \in H' \text{ and } f \text{ is satisfiable,} \\ w(H') &= 0, && \text{if } r \in H' \text{ and } f \text{ is not satisfiable,} \\ w(H') &\leq m'M - n \leq m'M, && \text{if } r \notin H'. \end{aligned} \tag{6.1}$$

Now consider the graph G. Suppose the formula f is not satisfiable. If the root r of any copy of G' is selected, this copy does not contribute any value to the weight of a subgraph of G (see 6.1). Therefore, the optimal subgraph is a subgraph of only one of the copies of G' with weight less than $m'M$. It cannot select vertices in more than one of the copies from G' since then it would have to connect those, forcing the roots of those copies to be in the solution. Conversely, suppose the formula f is satisfiable. If the subgraph of G does exclusively contain vertices from one of the copies of G' its weight is at most $m'M$. But, connecting the roots of all copies and collecting weight 1 from each copy in the process (see 6.1) yields a subgraph of G with weight $K > \frac{1}{\alpha} \cdot m'M > m'M$. Thus, a solution with weight larger than $m'M$ corresponds to a satisfying assignment and has weight K.

Suppose there exists an polynomial time α-approximation algorithm ALG for WSP or WISP. If the formula is satisfiable then the optimal solution has weight K so ALG would return a solution with weight at least $\alpha K > m'M$. Conversely, if the formula is not satisfiable the optimal solution has weight at most $m'M$. Thus ALG would return a solution with weight at most $m'M$, too.

In conclusion, any polynomial time α-approximation algorithm would be able to distinguish the two cases, and thus to solve SAT in polynomial time. Thus α-approximating WSP and WISP is $\mathcal{NP}$-complete. □

6.2 Budget-Constraint Problem

In principle, one could approximately solve MAX-WSP by formulating it as a Knapsack problem. For a bound B one seeks a subgraph H whose value $w(E) = \sum_{e \in E(H)} c(e)$ is maximal under the secondary condition that $w(V) = \sum_{v \in V(H)} d(v) \geq B$ holds. In this section, we want to investigate how the resulting problem with knapsack constraints differs from the original problem in terms of complexity and approximability.

The Knapsack problem is defined as follows:

Knapsack Problem	
Given:	Nonnegative integers $n, c_1, \dots, c_n, v_1, \dots, v_n$ and an integer $B \in \mathbb{Z}$.
Problem:	Find a subset $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} c_i \leq B$ and $\sum_{i \in S} v_i$ is maximal.

Knapsack is an $\mathcal{NP}$-complete problem which is well-studied in literature. We refer the reader to [AMO93].

A simple Knapsack formulation would not be interesting due to it not requiring the connectivity of the subgraph. Therefore, in order to use this approach for solving the WEIGHTED SUBGRAPH PROBLEM, the Knapsack formulation is altered and we obtain a budget-constraint version of the problem: Given an undirected graph $G = (V, E, \gamma)$ the BUDGET-CONSTRAINT WEIGHTED SUBGRAPH PROBLEM is defined as:

Definition 6.3 (Budget-Constraint Weighted Subgraph Problem)**.** Given an undirected graph $G = (V, E, \gamma)$ with node weights $d \colon V \to \mathbb{R}$ and edge weights $c \colon E \to \mathbb{R}$, a weight function w according to Definition 3.1, and an integer $B \in \mathbb{N}$. The BUDGET-CONSTRAINT WEIGHTED SUBGRAPH PROBLEM (BCWSP) asks for a connected subgraph H of G where $w(E)$ is maximized under the secondary condition that $w(V) \geq B$.

Let $G = (V, E, \gamma)$ be an undirected graph with node weights $c \colon V \to \mathbb{R}$ and edge weights $d \colon E \to \mathbb{R}$. Let $H = \text{MAX-WSP}(G)$ be an optimal solution with weight $w(H) = w* = w^*_E + w^*_V$, where $w^*_E = \sum_{e \in E(H)} c(e)$ and $w^*_V = \sum_{v \in V(H)} d(v)$. Furthermore, let $H(B) = \text{BCWSP}(G, B)$ be an optimal solution for the bound $B \in \mathbb{Z}$.

We denote the weight of $H(B)$ by $w^*(B) = w(H(B)) = w^*_E(B) + w^*_V(B)$. It holds that $w^*_V(B) \leq B$ and $w^*(B) \leq w^*$ for all bounds $B \in \mathbb{Z}$.

It follows that, for $B = w^*_V$, we have $w^*_E(B) \geq w^*_E$ since H is a feasible solution for BCWSP(G, B). Let

$$p := \begin{cases} \max_{\substack{v \in V \\ d(v)>0}} d(v), & \text{if } \exists v \in V \text{ with } d(v) > 0, \\ 0, & \text{otherwise} \end{cases}$$

be the largest and

$$q := \begin{cases} \min_{\substack{v \in V \\ d(v)<0}} d(v), & \text{if } \exists v \in V \text{ with } d(v) < 0, \\ 0, & \text{otherwise} \end{cases}$$

be the smallest node weight in G. Then $w(V) \geq nq$ and $w(V) \leq np$. Theoretically, one can solve MAX-WSP by solving BCWSP for all $B \in W := \{nq, nq + 1, \dots, -1, 0, 1, \dots, np - 1, np\}$ and taking the best solution.

Suppose there exists an polynomial α-approximation algorithm ALG(B) for the budget-constraint problem (6.3). Using the idea described above, we can use ALG(B) to obtain an approximation algorithm ALG for MAX-WSP. Since $w^*_V \in W$, ALG would return a solution H' with weight $w(H') \geq \alpha w^*_E + w^*_V \geq \alpha(w^*_E + w^*_V)$. In particular, approximation ration of ALG would be α.

But ALG is pseudo-polynomial since it is depended on the values of p and q. Instead of running ALG(B) for all $B \in W$ we run it for all $B \in W_{1+\varepsilon}$ where

$$W_{1+\varepsilon} := \Big\{ -(1+\varepsilon)^{\lceil \log_2 |q| \rceil}, \dots, -(1+\varepsilon)^2, -(1+\varepsilon), -1, 0, \\ 1, 1+\varepsilon, (1+\varepsilon)^2, \dots, (1+\varepsilon)^{\lceil \log_2 p \rceil} \Big\}$$

for some $\varepsilon > 0$. This will make it a polynomial algorithm. Using this grid, ALG$(\cdot)$ is executed for some $\bar{B} \in \left[\frac{w^*_V}{1+\varepsilon}, w^*_V\right]$. Since H is feasible for ALG$(\bar{B})$, this will yield a solution $w(H') \geq \alpha w^*_E + \frac{w^*_V}{1+\varepsilon} \geq \frac{\alpha}{1+\varepsilon}(w^*_E + w^*_V)$.

In summary, this describes an polynomial time $\frac{\alpha}{1+\varepsilon}$-approximation algorithm for the WEIGHTED SUBGRAPH PROBLEM. Thus, Theorem 6.2 yields the following corollary:

Corollary 6.4. *Given an undirected graph $G = (V, E, \gamma)$ with node weights $d\colon V \to \mathbb{R}$, edge weights $c\colon E \to \mathbb{R}$, and an integer B, the* BUDGET-CONSTRAINT WEIGHTED SUBGRAPH PROBLEM *is $\mathcal{NP}$-complete to approximate within a constant factor α for any* $0 < \alpha < 1$.

6.3 Preprocessing

We describe two preprocessing phases adapted from [EK14] that simplify an instance of the WEIGHTED SUBGRAPH PROBLEM without losing optimality. These reduction rules make a new graph with a smaller number of vertices and edges in such a way that the MAX-WSP solution for the original graph can be easily recovered from the MAX-WSP solution for the simplified graph. The rules can be slightly changed for reducing instances of MIN-WSP. This procedure can be executed before solving MAX-WSP either optimally or heuristically.

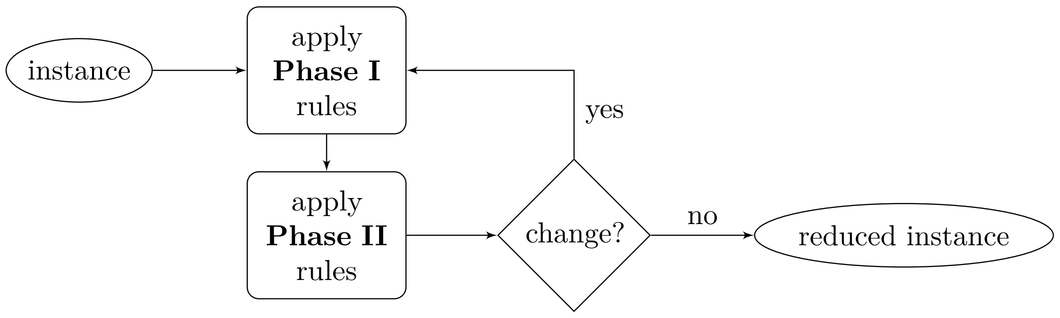


Figure 6.3: Preprocessing scheme.

A MAX-WSP instance passes through two phases of rules that are run exhaustively until no rules apply anymore. The result is a reduced instance.

Phase I

1. *Isolated node rule.* Let v be an isolated vertex. The node v will only be part of an optimal solution if it is the solution itself since it is not connected to the rest of the graph. Consequently, we can remove v from the graph and, in case the weight of v is positive, save that weight as an unlikely, but possible solution. Identifying all isolated nodes takes $\mathcal{O}(n)$ time.

2. *Adjacent nodes rule.* Let $e = (u, v)$ be an edge with $w(e) \geq 0$, $w(e) + w(u) \geq 0$, and $w(e) + w(v) \geq 0$. In this case, if one vertex is part of the solution the edge and the other vertex can also be included without decreasing the total weight. Thus, we can contract the edge e into a new vertex w with weight $w(w) = w(e) + w(u) + w(v)$. Finding all adjacent nodes takes $\mathcal{O}(m)$ time.

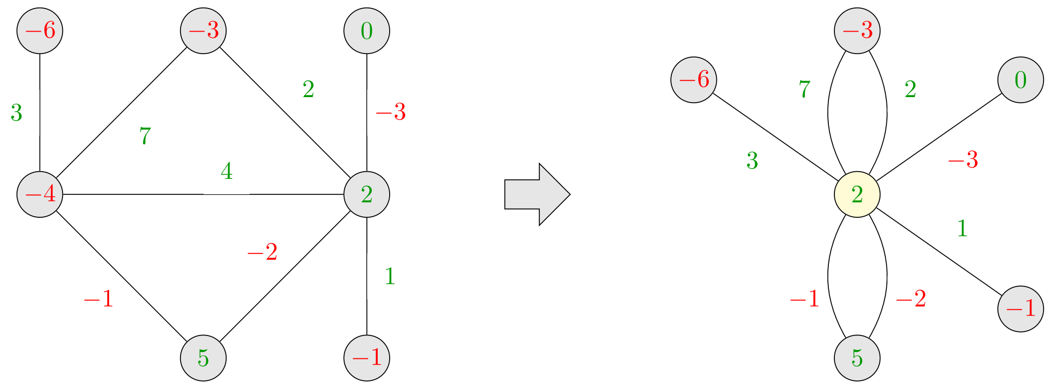


Figure 6.4: Applying the adjacent nodes rule.

3. *Parallel edges rule.* Let u and v be nodes with parallel edges between them. In that case we merge all non-negative ones into a single edge with weight of the sum of their weights. The reason being that if one of those edges will be chosen, it is only beneficial to choose the other non-negative edges, too. After that, we can remove all edges between u and v except the one with maximum weight. If there is a positive edge between u and v the last step removes all negative edges between them. Otherwise it will keep only one negative edge, the one with maximal weight, essentially making the graph simple. This step takes $\mathcal{O}(m)$ time.

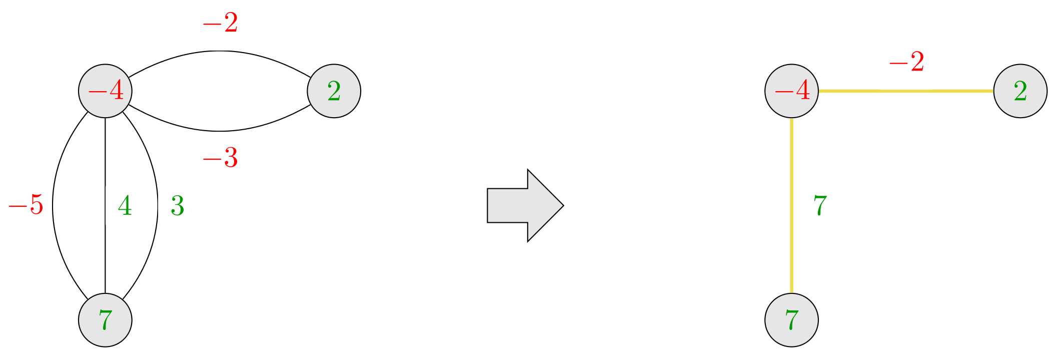


Figure 6.5: Applying the parallel edges rule.

4. *Chain rule.* Let v be a vertex with degree $g(v) = 2$ with corresponding incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$. If all three weights $w(v)$, $w(e_1)$ and $w(e_2)$ are non-positive, then v, e_1 and e_2 can be replaced by a single edge $e = (u, w)$ with a weight $w(e) = w(v) + w(e_1) + w(e_2)$. Either both e_1 and e_2 will be part of an optimal solution or none of them. In the latter case they are used as a connection between positive parts. Merging negative chains is implemented in a single pass by iteratively trying to apply the rule for all the nodes. Identifying all nodes which satisfy the condition takes $\mathcal{O}(n)$ time.

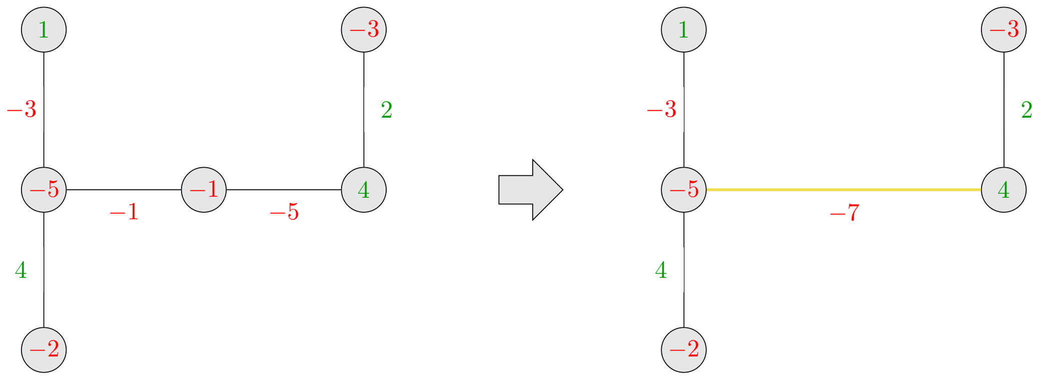


Figure 6.6: Applying the chain rule.

Phase II

1. *Mirrored hubs rule.* Let $u, v \in V$ be two distinct nodes with non-positve weights. Without loss of generality assume that $w(u) \leq w(v)$. If u and v are adjacent to the same nodes and all edge weights $w(e)$, e incident to u, are non-positive then we can remove u from the graph. The reason is that v will always be preferred over u in an optimal solution, because it is adjacent to exactly the same nodes as u and costs less. More generally, this reduction is also executable if the neighborhood of u is a subset of neighborhood of v. Finding all pairs takes $\mathcal{O}(\Delta \cdot n^2)$ time where Δ is the maximum degree of the graph.

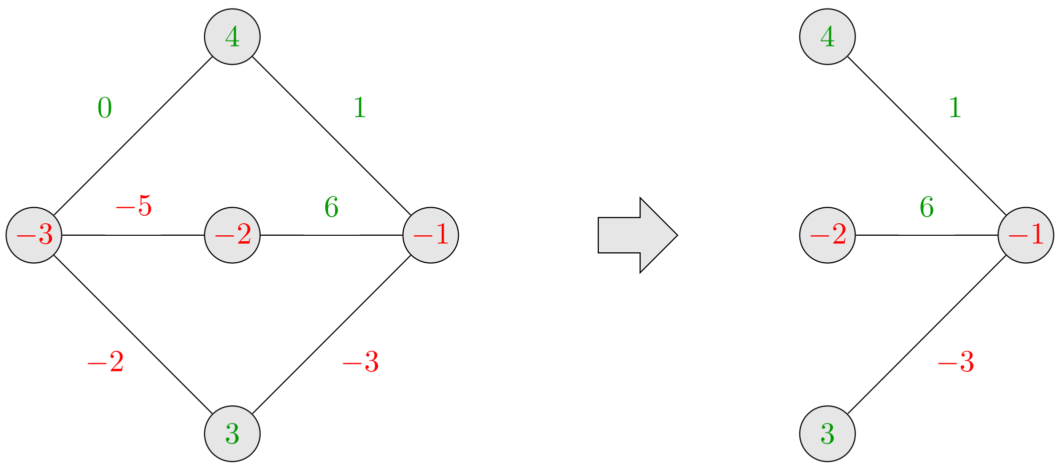


Figure 6.7: Applying the mirrored hubs rule.

During this process we keep a mapping from the merged nodes to sets of original nodes to map solutions of the reduced instance to solutions of the original instance.

6.4 Postprocessing

Similar to the preprocessing procedure presented in the last Section 6.3 one can execute a postprocessing procedure to improve a given heuristic solution for the Weighted Subgraph Problem. Given a sub-optimal or heuristic solution we describe operations that can be performed to improve the weight of the subgraph. Again, we give a procedure for maximizing the value of the weight function. The procedure can be changed slightly for the minimization case.

A sub-optimal subgraph H of G passes through four rules that are run exhaustively until no rules apply anymore. The result is a subgraph with larger weight.

1. *Positive edges rule.* Let $e \in G$ be an edge in G. If $e = (u, v)$ is not in H but $u, v \in H$ and $w(e) \geq 0$, adding e to H will not reduce the weight of the subgraph. Identifying all missing positive edges takes $\mathcal{O}(m)$ time.
2. *Negative node rule.* Let $v \in H$ be an vertex with $w(v) + \sum_{e \in \delta_H(v)} w(e) < 0$. If H stays connected when removing v, the weight of the subgraph can be improved by doing so. This step takes $\mathcal{O}(n + m)$ time.
3. *Negative edges rule.* Let $e \in H$ be an edge with weight $w(e) < 0$. Analogous to the last rule if H stays connected when removing e, it is safe to do so. Identifying all negative edges takes $\mathcal{O}(m)$ time.
4. *Neighboring nodes rule.* Let v be a neighbor of a node $u \in H$ with $w(v) + \sum_{e \in \delta_H(u)} w(e) \geq 0$. Then we can add u along with the edges in the sum without reducing the weight of the subgraph. Going through all neighbors and all incident edges can be done in time $\mathcal{O}(n + m)$

It follows from the description of the operations that the subgraph returned by the procedure has a larger weight then the input subgraph.

6.5 Node Set Heuristic

Due to the Weighted Subgraph Problem being $\mathcal{NP}$-complete to approximate within a constant factor we present a heuristic procedure for obtaining approximative solutions.

Given an undirected graph $G = (V, E, \gamma)$. Suppose we are given the set of nodes $V(H)$ of an optimal solution $H = \text{MAX-WSP}(G)$. How do we select the edges? What is the complexity of this problem? Can it be solved efficiently?

We describe a procedure that reduces a copy of the graph G to an optimal MAX-WSP-subgraph H' of G.

Procedure 6.5. Given an undirected graph $G = (V, E, \gamma)$ and a set of nodes $V(\bar{H})$ of an optimal solution $\bar{H}$ for MAX-WSP on G do the following steps to create a graph H'. Start with H' being a copy of G.

1. Remove all vertices $v \in V \setminus V(\bar{H})$ from H'.
2. Select all non-negative edges $P = \{e \in E | w(e) \geq 0\}$ in H'. If the resulting graph $(V(\bar{H}), P)$ is connected, STOP: Then $H \coloneqq (V(\bar{H}), P)$ is an optimal solution for MAX-WSP.
3. Contract each connected component C of $(V(\bar{H}), P)$ to a single node v_C in H' with weight $w(v_C) = w(C)$.
4. Compute a maximum edge-weighted spanning tree T in H'. Then $H \coloneqq (V(\bar{H}), P \cup E(T))$ is an optimal solution for MAX-WSP.

Theorem 6.6. *Given a set of nodes $V(\bar{H})$ of an optimal solution $\bar{H}$ for* MAX-WSP *on a graph $G = (V, E, \gamma)$ Procedure 6.5 computes an optimal solution for* MAX-WSP *in time $\mathcal{O}(m + n \log n)$.*

Proof. Assume the solution H returned by Procedure 6.5 is not optimal. Since both subgraphs have the same set of vertices $V(H) = V(\bar{H})$, which means that also the vertex weights $w(V(H)) = w(V(\bar{H}))$ are equal, the selected edges have to differ and $w(E(H)) < w(E(\bar{H}))$.

Suppose the Procedure 6.5 terminated in step 2. Removing all vertices $v \in V \setminus V(\bar{H})$ in step 1 (and hereby removing incident edges) from H' did not remove edges selected by the solution $\bar{H}$. Otherwise if $\bar{H}$ selected one of those edges it would have to select the corresponding incident vertex from $v \in V \setminus V(\bar{H})$ which is not possible. But then H contains all non-negative edges from H' and thus all nonegative edges from $\bar{H}$ as well implying that $w(E(H)) \geq w(E(\bar{H}))$ which is a contradiction.

Suppose the procedure terminated in step 4 and assume $H \coloneqq (V(\bar{H}), P \cup E(T))$ is not optimal, meaning $w(H) < w(\bar{H})$. But then the maximum edge-weighted spanning tree in H' has less weight than the edges connecting $(V(\bar{H}), P)$ in $\bar{H}$ which is contradiction. Thus H is an optimal solution for MAX-WSP.

For the complexity observe that step 1 needs $\mathcal{O}(n)$, step 2 $\mathcal{O}(n + m)$, and step 3 $\mathcal{O}(n + m)$ time. For step 4 we can use Kruskals Algorithm to compute in maximum weighted spanning tree in time $\mathcal{O}(m + n \log n)$ [KN12]. $\square$

Suppose we are given a random set of nodes S, can we still use Procedure 6.5 with that set as an input to compute a feasible solution for the Weighted Subgraph Problem? Unfortunately no, since we cannot execute step 1 in the same way. Step 1 used the fact that the nodes originated from an optimal solution. The problem now is that after removing all vertices $v \in V \setminus S$ it might not be possible to connect all nodes from S. For this reason we might need to add additional nodes to S to obtain a feasible solution.

To obtain a practical heuristic it might be essential to pick a "good" set of nodes for the procedure instead of a random one. Consider the following procedure. Note that if it is possible to execute Procedure 6.5 for a random set of nodes, we get a lower bound for the value of the optimal max-wsp-subgraph. Let N be a constant such that computing max-wsp exactly on a graph with at most N nodes can be done within a reasonable amount of time. In Chapter 8 of this thesis we investigate in the practical use of this heuristic and how to choose N.

Procedure 6.7 (Node Set Heuristic for max-wsp)**.** Given an undirected graph $G = (V, E, \gamma)$ do the following steps to create a subgraph H. Start with H being a copy of G.

1. Take all non-negative nodes of G, that is $S \coloneqq \{v \in V : w(v) \geq 0\}$ and add them to $V(H)$.
2. Select all non-negative edges $P = \{e \in E : w(e) \geq 0\}$ in G' and add them to $E(H)$. If the graph H is connected, STOP: Then H is an solution for max-wsp.
3. Otherwise, contract each connected component C of H in G to a single node v_C with weight $w(v_C) = w(S) + w(P)$.
4. Pick $|V(G)| - N$ vertices in G and solve a modified version of wsp where all picked vertices are forced to be in the solution using an exact algorithm.
5. Apply the postprocessing scheme from Section 6.4 to the output of the modified wsp version to obtain H.
6. Repeat steps 4 and 5 multiple times and take the best solution returned.

6.6 Spanning Tree Heuristic

Similarly to last section, we describe a heuristic procedure for the Weighted Subgraph Problem. This time, the heuristic uses the dynamic programming algorithm from Chapter 4. Before applying the following procedure we recommend using the preprocessing scheme from Section 6.3.

Given an undirected graph $G = (V, E, \gamma)$ with node weights $c : V \to \mathbb{R}$ and edge weights $d : E \to \mathbb{R}$. Assume we want to maximize the value of the weight function. The heuristic works as follows:

1. Compute a maximum edge-weighted spanning tree T on the graph G.
2. Solve the Weighted Subgraph Problem on T using the dynamic program 4.2 described in Section 4.2 and obtain a subgraph $H_T = \text{max-wsp}(T)$.
3. Apply the postprocessing procedure to H_T to obtain heuristic solution H.

We compare the practical use in terms of running time and solution quality of this heuristic to an IP solver in Chapter 8.

Chapter 7

Integer Programs

This chapter describes integer programs representing the problems discussed in previous chapters. In the next section we have a look at the implementation and practical results of some of the algorithms described throughout this thesis.

Let $G = (V, E, \gamma)$ an undirected graph with nonnegative node weights $c\colon V \to \mathbb{R}_{\geq 0}$ and nonnegative edge weights $d\colon E \to \mathbb{R}_{\geq 0}$. In all following IP/LP approaches, there are binary variables y_v indicating whether a vertex v is selected for the subgraph and binary variables z_e indicating selected edges. By $y(S)$ we denote the sum of all variable values y_v for $v \in S$: $y(S) := \sum_{v \in S} y_v$. We formulate our IP/LP approaches as maximization problems, but they can be easily modified for solving minimization problems.

7.1 Rooted Weighted Subgraph Problem

Before we formulate an optimization problem for the Weighted Subgraph Problem or the Weighted Induced Subgraph Problem, we start by looking at the rooted variants, namely the Rooted Weighted Subgraph Problem and Rooted Weighted Induced Subgraph Problem. As it turns out those programs are slightly easier to set up since formulating connectivity constraints in the rooted case is more intuitive.

Formulating the objective function is straightforward

$$\sum_{e \in E} z_e \cdot c(e) + \sum_{v \in V} y_v \cdot d(v). \tag{7.1}$$

For the induced variant, RWISP, the requirement of the subgraph being induced can be modeled by the constraints

$$\begin{aligned} z_e &\geq y_u + y_v - 1 && \text{for all } e \in E \text{ with } \delta(e) = (u, v), \\ z_e &\leq y_v && \text{for all } e \in E \text{ and } v \in \delta(e). \end{aligned} \tag{7.2}$$

Given a root vertex $r \in V$, setting $y_r := 1$ forces the r to be in the solution. Thus, we are able ensure connectivity of the subgraph by the constraints

$$|V| \cdot z(\delta(S)) \geq y(\bar{S}) \quad \text{for all } S \subseteq V \text{ with } r \in S. \tag{7.3}$$

Basically, one inequality is added for every cut $(S, \bar{S})$ of V with $r \in S$. If there is a node $v \in \bar{S}$ with $y_v = 1$, this ensures that at least one of the edges in the cut has to be selected, too. Thus, the IP for MAX-RWSP can be formulated as follows:

$$\begin{aligned}
\max \quad & \sum_{e\in E} z_e \cdot c(e) + \sum_{v\in V} y_v \cdot d(v) && \\
\text{s.t.} \quad & z_e \geq y_u + y_v - 1 && \text{for all } e \in E \text{ with } \delta(e) = (u,v), \\
& z_e \leq y_v && \text{for all } e \in E \text{ and } v \in \delta(e), \\
& y_r = 1, && \\
& |V| \cdot z(\delta(S)) \geq y(\bar{S}) && \text{for all } S \subseteq V \text{ with } r \in S, \\
& z_e \in \mathbb{B} && \text{for all } e \in E, \\
& y_v \in \mathbb{B} && \text{for all } v \in V.
\end{aligned} \tag{CUT$_r$}$$

The IP (CUT$_r$) can be easily modified for solving the minimization problems MIN-RWISP or the MIN-RWSP in the most direct way by minimizing the objective function (7.1).

Theorem 7.1. *Given a root vertex $r \in V$, IP* (CUT$_r$) *returns an optimal solution for the* Rooted Weighted Induced Subgraph Problem. *If the constraints* (7.2) *are dropped an optimal solution for* Rooted Weighted Subgraph Problem *is returned.*

Proof. Let $G = (V, E, \gamma)$ be a graph and suppose we are given a feasible solution $(\bar{y}, \bar{z})$ for the IP (CUT$_r$) representing a graph H. Then $\bar{y}_r = 1$ so the solution contains the root node $r \in H$. Further, suppose we have $\bar{z}_e = 0$ but $\bar{y}_u = \bar{y}_v = 1$ for two incident nodes $u, v \in \delta(e)$. But this is not possible due to the constraints (7.2) forcing $\bar{z}_e$ to be one. Conversely, the constraints (7.2) ensure that, if $\bar{z}_e = 1$, then $\bar{y}_u = 1$ and $\bar{y}_v = 1$ for $u, v \in \delta(e)$. So H represents an induced subgraph of G. Now assume H is not connected. Let C be the connected component of H containing r. Since H is not connected $\bar{z}(\delta(S)) = 0$ and there are nodes $v \in \bar{C}$ with $\bar{y}_v = 1$, so $\bar{y}(\bar{C}) > 0$. This contradicts constraint (7.3), and thus H can only have one connected component.

For the converse, assume we are given a connected induced subgraph of H containing r. To obtain a feasible solution of we set $\bar{y}_v =: 1$ for all $v \in H$ and $\bar{y}_v =: 0$ for all $v \in G \setminus H$. Analogously, $\bar{z}_e =: 1$ for all $e \in E(H)$ and $\bar{z}_e =: 0$ for all $e \in E(G) \setminus E(H)$. Since H is an induced subgraph, the constraints (7.2) are satisfied. Let S be an arbitrary subset of V with $r \in S$. If $y(\bar{S}) > 0$ at least one node of H is in $\bar{S}$. But since H is connected $\bar{z}(\delta(S)) > 0$ and therefore the constraint (7.3) is satisfied.

If the constraints (7.2) are dropped, the subgraph is (not necessarily) induced, but all other arguments continue to hold. □

Corollary 7.2. *In order to obtain a solution for the* Weighted Subgraph Problem *and the* Weighted Induced Subgraph Problem *one can solve the IP* (CUT$_r$) *obtaining solutions for the* Rooted Weighted Subgraph Problem *and the* Rooted Weighted Induced Subgraph Problem *for all $r \in V$ and take the maximum or minimum of all those solutions.*

In the above form it is not clear, that the relaxation of IP (CUT$_r$) can be solved in polynomial time since it has an exponential number of constraints. The problem hereby are the connectivity constraints (7.3), because they imply exponentially many inequalities. Our goal in the following is to separate the connectivity constraints and solve the remaining LP relaxation. We know that solving the separation problem in polynomial time is equivalent to solving the LP relaxation of the IP in polynomial time [Wol98].

Suppose we are given a $(\bar{y}, \bar{z})$ satisfying all constraints except the connectivity constraints (7.3). We show that we can check in polynomial time whether $(\bar{y}, \bar{z})$ satisfy the connectivity constraints. To achieve this we introduce a flow to our graph. For this we need to modify our graph to be directed. The auxiliary graph $\overrightarrow{G} = (V, R)$ is defined as follows: Each edge $e = (u, v) \in E$ corresponds to two arcs (u, v) and (v, u) with lower capacity $\ell(uv) = \ell(vu) = 0$ and upper capacity $u(uv) = u(vu) = |V| \cdot \bar{z}(e)$. For all nodes $v \in V$ we add an arc (v, r) incident to the root r. The lower capacity for those is $\ell(vr) = \bar{y}_v$ and the upper capacity is $u(vr) = \infty$ (see Figure 7.1).

$$\begin{aligned}
&\overrightarrow{G} = (V, R) \text{ with} \\
&R = \{(u, v) | (u, v) \in E\} \cup \{(v, u) | (u, v) \in E\} \cup \{(v, r) | v \in V\}, \\
\ell(e) = &\begin{cases} 0, \text{ if } e = (u, v) \in E \text{ or } e = (u, v) \in E, \\ \bar{y}_v, \text{ if } e = (v, r) \notin E \end{cases}, \\
u(e) = &\begin{cases} |V| \cdot \bar{z}(e), \text{ if } e = (u, v) \in E \text{ or } e = (u, v) \in E, \\ \infty, \text{ if } e = (v, r) \notin E \end{cases}.
\end{aligned}$$

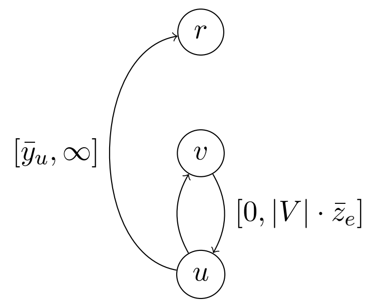


Figure 7.1: Construction of the auxiliary graph G'.

In order to prove the next lemma we need *Hoffman's circulation theorem.*

Theorem 7.3 (Hoffman's circulation theorem)**.** *Let $G = (V, R, \alpha, \omega)$ be a directed graph and let $\ell, u \colon R \to \mathbb{R}^+$ satisfy $\ell(r) < u(r)$ for every $r \in R$. Then either there exists a circulation $f \colon R \to \mathbb{R}$ with $\ell(r) \leq f(r) \leq u(r)$ for every $r \in R$ or there exists $S \subseteq V$ such that*

$$u(S) := \sum_{a \in \delta^+(S)} u(a) < \sum_{a \in \delta^-(S)} \ell(a) =: \ell(\bar{S})$$

For the proof of Theorem 7.3 we refer to [AMO93].

Lemma 7.4. *Given $(\bar{y}, \bar{z})$ satisfying all constraints except the connectivity constraints* (7.3) *of the IP* (CUT$_r$)*. There exists a circulation with respect to ℓ and u in the auxiliary graph $\overrightarrow{G}$ if and only if $(\bar{y}, \bar{z})$ satisfies the connectivity constraint* (7.3)*.*

Proof. Assume $(\bar{y}, \bar{z})$ satisfies all constraints except the connectivity constraint of the IP (CUT$_r$) and there exists a circulation with respect to ℓ and u in $\overrightarrow{G}$. Using Hoffman's circulation theorem 7.3 it follows that $u(S) \geq \ell(\bar{S})$ for all $S \subseteq V$. Let

$S \subseteq V$ be an arbitrary subset of V. If $r \in S$ we get that:

$$\bar{y}(\bar{S}) = \sum_{v \in \bar{S}} \bar{y}_v = \ell(\bar{S}) \leq u(S) = \sum_{a \in \delta^+(S)} u(a) = |V| \cdot \bar{z}(\delta(S)).$$

If $r \notin S$ there is nothing to show. Thus $(\bar{y}, \bar{z})$ is a feasible solution for the IP (CUT$_r$).

Conversely, suppose that the connectivity constraint (7.3) holds for $(\bar{y}, \bar{z})$. Again, let $S \subseteq V$ be an arbitrary subset and assume $r \in S$. We have:

$$\begin{aligned} \ell(\bar{S}) &= \bar{y}(\bar{S}) \\ &\leq |V| \cdot \bar{z}(\delta(S)) \\ &= u(S) \end{aligned}$$

If $r \notin S$, then $\ell(\bar{S}) = 0$ and $u(S) = |V| \cdot \bar{z}(\delta(S)) + \infty$, so $\ell(\bar{S}) \leq u(S)$ for all $S \subseteq V$. We obtain a feasible circulation with respect to ℓ and u by applying Theorem 7.3. □

In order to show that the LP relaxation of IP (CUT$_r$) is solvable in polynomial time, once again, we need to modify our graph $\overrightarrow{G}$ by adding a supersource s and a supersink t. We obtain a new graph $G' = (V', R')$ where

$$\begin{aligned} V' &:= V(\overrightarrow{G}) \cup \{s, t\} \\ R' &:= R(\overrightarrow{G}) \cup \{(s, v), (v, t) \text{ for all } v \in V\} \end{aligned}$$

without lower capacities (i.e. $\ell'(r) = 0$ for all $r \in R'$) and

$$\begin{aligned} u'(r) &= u(r) - \ell(r) && \text{for all } r \in R(\overrightarrow{G}), \\ u'(s, v) &= \ell(\delta^+(v)) = \sum_{r \in \delta^+(v)} \ell(r) && \text{for all } v \in V(\overrightarrow{G}), \\ u'(v, t) &= \ell(\delta^-(v)) && \text{for all } v \in V(\overrightarrow{G}). \end{aligned}$$

Lemma 7.5. *A circulation with respect to ℓ and u in $\overrightarrow{G}$ exists if and only if the maximum (s, t)-flow in G' (with respect to u') has value $F := \sum_{r \in R} \ell(r)$.*

Proof. Observe that $F = \sum_{r \in R} \ell(r) = \sum_{v \in V} \sum_{r \in \delta^+(v)} \ell(r) = \sum_{v \in V} \ell(\delta^+(v))$ and also $F = \sum_{v \in V} \ell(\delta^-(v))$, which means the arcs from s and to t are necessarily saturated in a flow of value F. Given a feasible circulation β in $\overrightarrow{G}$, consider flow f' in G' with

$$\begin{aligned} f'(s, v) &= \ell(\delta^+(v)) && \text{for all } v \in V, \\ f'(v, t) &= \ell(\delta^-(v)) && \text{for all } v \in V, \\ f'(r) &= \beta(r) - \ell(r) && \text{for all } r \in R. \end{aligned}$$

It follows that:

$$\begin{aligned} 0 = f'(s, v) &= \ell(\delta^+(v)) = u'(s, v), \\ 0 = f'(v, t) &= \ell(\delta^+(v)) = u'(v, t), \\ 0 = f'(r) &= \beta(r) - \ell(r) \leq u(r) - \ell(r) = u'(r), \end{aligned}$$

and

$$\begin{aligned}
\sum_{r\in\delta^+(v)} f'(r) &= f'(v,t) + \sum_{\substack{r\in\delta^+(v)\\ r\neq(v,t)}} \beta(r) - \ell(r)\\
&= \ell(\delta^-(v)) - \ell(\delta^+(v)) + \sum_{\substack{r\in\delta^+(v)\\ r\neq(v,t)}} \beta(r)\\
&= -\ell(\delta^+(v)) + \ell(\delta^-(v)) - \sum_{\substack{r\in\delta^-(v)\\ r\neq(v,t)}} \beta(r)\\
&= -f'(s,v) - \sum_{\substack{r\in\delta^-(v)\\ r\neq(v,t)}} \beta(r) - \ell(r)\\
&= -\sum_{r\in\delta^-(v)} f'(r)
\end{aligned}$$

for all $v \in V' \setminus \{s,t\}$. Therefore, the flow f' is a feasible (s,t)-flow of value F in G'.

Conversely, given a feasible (s,t)-flow of value F in G', consider β in $\overrightarrow{G}$ with

$$\beta(r) = f'(r) + \ell(r).$$

This satisfies the capacity constraints

$$\ell(r) \leq f'(r) + \ell(r) = \beta(r) \leq u'(r) + \ell(r) = u(r)$$

for all $r \in R$ and the flow conservation constraints

$$\begin{aligned}
\sum_{r\in\delta^+(v)} \beta(r) &= \sum_{r\in\delta^+(v)} f'(r) + \ell(r)\\
&= \ell(\delta^+(v)) + \sum_{r\in\delta^+(v)} f'(r)\\
&= -\ell(\delta^-(v)) - \sum_{r\in\delta^-(v)} f'(r)\\
&= -\sum_{r\in\delta^-(v)} f'(r) + \ell(r)\\
&= -\sum_{r\in\delta^-(v)} \beta(r)
\end{aligned}$$

for all $v \in V$. Thus β is a feasible circulation in $\overrightarrow{G}$. □

Corollary 7.6. *Given $(\bar{y}, \bar{z})$ satisfying all constraints except the connectivity constraint* (7.3) *the separation problem can be solved in polynomial time.*

Proof. The auxiliary graph $\overrightarrow{G}$ and the graph G' can both be constructed in linear time. Since the maximum flow problem is solvable in polynomial time, this follows from Lemma 7.4 and Lemma 7.5. □

7.2 Encoding the Presence of a Root

We now turn to the optimization problems for the Weighted Subgraph Problem or the Weighted Induced Subgraph Problem. In the previous approach we took advantage of the root vertex being forced to be in the solution to set up the IP (CUT$_r$). For the unrooted variant, none of the vertices are forced to be in the solution, so we need to come up with another way of ensuring graph connectivity.

Instead of relying on a root node being present, we use binary auxiliary variables x_v to encode the presence of a root node. The following constraints are added:

$$\begin{aligned} \sum_{v \in V} x_v &= 1, \\ x_v &\le y_v \quad \text{for all } v \in V. \end{aligned} \tag{7.4}$$

Those state that there is exactly one root node and that a node can only be the root node if it is present in the solution. Therefore we are able to reformulate the connectivity constraint from IP (CUT$_r$) in a similar way.

$$y_v \le z(\delta(S)) + x(S) \quad \text{for all } v \in V, \{v\} \subseteq S \subseteq V. \tag{7.5}$$

The connectivity constraints have been modified and state that a node v can only be present in the solution if for all sets $S \subseteq V$ containing v, either the selected root node is in S, or an edge in the set $\delta(S)$ is in the solution.

The requirement of the subgraph being induced for WISP remains unchanged and is modeled by the constraints (7.2).

Hence, we obtain a formulation for the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem:

$$\begin{aligned}
\max \quad & \sum_{e \in E} z_e \cdot c(e) + \sum_{v \in V} y_v \cdot d(v) \\
\text{s.t.} \quad & z_e \ge y_u + y_v - 1 && \text{for all } e \in E \text{ with } \delta(e) = (u, v), \\
& z_e \le y_v && \text{for all } e \in E \text{ and } v \in \delta(e), \\
& \sum_{v \in V} x_v = 1, \\
& x_v \le y_v && \text{for all } v \in V, \\
& y_v \le z(\delta(S)) + x(S) && \text{for all } v \in V, \{v\} \subseteq S \subseteq V, \\
& z_e \in \mathbb{B} && \text{for all } e \in E, \\
& y_v \in \mathbb{B} && \text{for all } v \in V, \\
& x_v \in \mathbb{B} && \text{for all } v \in V.
\end{aligned} \tag{CUT}$$

We formulated IP (CUT) for MAX-WSP and MAX-WISP. It is obvious that by minimizing the objective function (7.1) the program solves MIN-WISP. For solving MIN-WSP, the constraints (7.2) need to be dropped.

Lemma 7.7. *IP* (CUT$_r$) *returns an optimal solution for the* Weighted Induced Subgraph Problem. *If the constraints* (7.2) *are dropped an optimal solution for* Weighted Subgraph Problem *is returned.*

Proof. Due to Theorem 7.1 it only remains to show that constraints (7.4) and (7.5) ensure graph connectivity.

Let $G = (V, E, \gamma)$ be a graph and suppose we are given a feasible solution $(\bar{x}, \bar{y}, \bar{z})$ for the IP (CUT$_r$) representing a graph H. Assume H is not connected. Because of the root constraints (7.4) there is exactly one node $r \in V(G)$ with $\bar{x}_r = 1$ and

$r \in H$. Since H is not connected, H has at least two connected components. Let C be a connected component of H which does not contain r. Then $\bar{x}(C) = 0$, $\bar{z}(\delta(C)) = 0$ and there are nodes $v \in C$ with $\bar{y}_v = 1$. But then $\bar{y}_v > \bar{z}(\delta(C)) + \bar{x}(C)$ which contradicts the connectivity constraints (7.5), and thus H can only have one connected component.

For the converse, assume we are given a connected subgraph of H. We choose any node $r \in V(H)$ and set $\bar{x}_r = 1$. For all other nodes $v \in V(H) \setminus r$ we set $\bar{x}_v = 0$. This implies that the root constraints (7.4) are satisfied. Let S be an arbitrary subset of V with $r \in S$. Then $\bar{x}(S) = 1$ and for all $v \in S$ it holds that $\bar{y}_v \leq \bar{z}(\delta(S)) + \bar{x}(S)$. If $r \notin S$ then $\bar{x}(S) = 0$. In that case assume $\bar{y}_v = 1$ for some $v \in S$. Since $r \notin S$ and H is connected, this implies $\bar{z}(\delta(S)) > 0$. If $\bar{y}_v = 0$, there is nothing to show. Consequently, the connectivity constraints (7.5) is satisfied. □

7.3 Flow Formulation

Another way to ensure graph connectivity is by a flow formulation. It is based on the idea that an induced subgraph contains a path from the root node r to every other node in $V \setminus r$. The formulation thus attempts to pick edges such that one unit of flow can be sent from the node r to every other node in $V \setminus r$. Given the undirected graph $G = (V, E, \gamma)$ let $\overrightarrow{G} = (V, A)$ be the corresponding bi-directed graph, where each edge $e = (u, v)$ is by replaced by two directed edges $\overrightarrow{e} = (u, v)$ and $\overleftarrow{e} = (v, u)$. For each arc $a \in A$, the variable f_a represents the flow from node u to node v. The other variables are defined as before.

In the first step, similar to the cut based formulation, we use a root vertex $r \in V$ to formulate a flow based integer program for solving the Rooted Weighted Subgraph Problem or the Rooted Weighted Induced Subgraph Problem. Setting $y_r := 1$ forces the root node to be in the solution. Using the variables $f_a, a \in A$ the following constraints are added:

$$\begin{aligned} y_v &\leq \sum_{a \in \delta^-(v)} f_a - \sum_{a \in \delta^+(v)} f_a \quad && \text{for all } v \in V \setminus r, \\ |V| \cdot z_e &\geq f_{\overleftarrow{e}} + f_{\overrightarrow{e}} && \text{for all } e \in E. \end{aligned} \tag{7.6}$$

The first inequality states that if node $v \neq r$ is chosen, the flow going into v minus the flow leaving v is one. If v is not chosen, that difference is 0. The flow is only allowed to go through arcs whose corresponding edge has been chosen which is captured in the second inequality. Both constraints ensure one unit flows from the node r to every other node in V if it is chosen for the subgraph, and there exists a path from the node r to every other chosen node resulting in a connected subgraph of G.

This gives the next IP formulation:

$$
\begin{array}{lrll}
\max & \displaystyle\sum_{e\in E} z_e \cdot c(e) + \sum_{v\in V} y_v \cdot d(v) & & \\
\text{s.t.} & z_e \geq y_u + y_v - 1 & \text{for all } e \in E \text{ with } \delta(e) = (u,v), & \\
& z_e \leq y_v & \text{for all } e \in E \text{ and } v \in \delta(e), & \\
& y_r = 1, & & \\
& y_v \leq \displaystyle\sum_{a\in\delta^-(v)} f_a - \sum_{a\in\delta^+(v)} f_a & \text{for all } v \in V \setminus r, & (\text{FLOW}_r) \\
& |V| \cdot z_e \geq f_{\overleftarrow{e}} + f_{\overrightarrow{e}} & \text{for all } e \in E, & \\
& f_a \in \mathbb{N} & \text{for all } a \in A, & \\
& z_e \in \mathbb{B} & \text{for all } e \in E, & \\
& y_v \in \mathbb{B} & \text{for all } v \in V. &
\end{array}
$$

Again, the IP (FLOW$_r$) can be easily modified for solving the minimization problems MIN-RWISP or the MIN-RWSP by minimizing the objective function (7.1).

Theorem 7.8. *Given a root vertex $r \in V$, IP* (FLOW$_r$) *returns an optimal solution for the* Rooted Weighted Induced Subgraph Problem. *If the constraints* (7.2) *are dropped an optimal solution for the* Rooted Weighted Subgraph Problem *is returned.*

Proof. Let $G = (V, E, \gamma)$ be a graph and suppose we are given a feasible solution $(\bar{y}, \bar{z})$ for the IP (FLOW$_r$) representing a graph $H \leq G$. By Theorem 7.1 it only remains to show that the flow constraints (7.6) imply the connectivity of H. So suppose H is not connected. Then there is a node $v \in V(H)$ with $\bar{y}_v = 1$ but there is no path from r to v in H. Due to constraint (7.6) there has to be one unit of flow going to v and that flow can only use edges in H. Since there is no path from r to v, it is not possible to send flow to v.

Conversely, assume we are given a connected induced subgraph of H containing r. Similarly to the proof of Theorem 7.1 we set $\bar{y}_v := 1$ for all $v \in H$ (0 otherwise), $\bar{z}_e := 1$ for all $e \in E(H)$ (0 otherwise). Again, it suffices to show that we can choose the flow variables f_a in such a way that the flow constraints (7.6) are satisfied. For all $v \in H$ we choose a (shortest) path P from v to r in $\overrightarrow{H}$ and increase the value of f_a by one unit for all $a \in P$. Therefore the first inequality in (7.6) is satisfied and $0 \leq f_a \leq |V| \cdot z_e$ for all $a \in A$. It remains to show that the second inequality in (7.6) holds. For each node v of H there exist at most $\sum_{v\in V} \bar{y}_v = |V(H)| \leq |V|$ paths going through it. Assume there is flow going through a node u in both directions. Then there has to be a neighbor u of v with flow going in both directions too and there have to be paths from r to w_1 and w_2 passing through both u and v. By redirecting both paths as in Figure 7.2 we ensure that the flow through both u and v is going in one direction and the new paths are shorter. Therefore the flow in each node is going in exactly one direction and (7.6) holds. □

As in the last section, we can use binary auxiliary variables x_v to encode the presence of a root node. Hence, the root constraints (7.4) are added. The flow constraints need to be slightly adjusted:

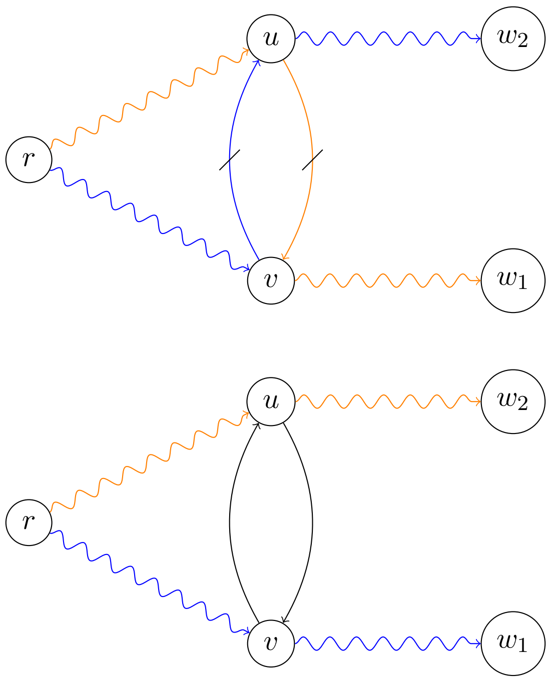


Figure 7.2: Redirecting the flow through u and v.

$$\begin{aligned} y_v - x_v \cdot |V| &\leq \sum_{a \in \delta^-(v)} f_a - \sum_{a \in \delta^+(v)} f_a && \text{for all } v \in V, \\ |V| \cdot z_e &\geq f_{\overleftarrow{e}} + f_{\overrightarrow{e}} && \text{for all } e \in E. \end{aligned} \tag{7.7}$$

The difference being that chosen root node has to send flow of value at most $|V| - 1$. For any other node v it states the same as the flow constraints (7.6) of the rooted variant.

Hence, we can formulate an IP for the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem using the flow formulation to ensure connectivity.

$$
\begin{aligned}
\max \quad & \sum_{e\in E} z_e \cdot c(e) + \sum_{v\in V} y_v \cdot d(v) && \\
\text{s.t.} \quad & z_e \geq y_u + y_v - 1 && \text{for all } e \in E, \\
& && \text{with } \delta(e) = (u,v), \\
& z_e \leq y_v && \text{for all } e \in E \text{ and } v \in \delta(e), \\
& \sum_{v\in V} x_v = 1, && \\
& x_v \leq y_v && \text{for all } v \in V, \\
& y_v - x_v \cdot |V| \leq \sum_{a\in\delta^-(v)} f_a - \sum_{a\in\delta^+(v)} f_a && \text{for all } v \in V, \\
& |V| \cdot z_e \geq f_{\overleftarrow{e}} + f_{\overrightarrow{e}} && \text{for all } e \in E, \\
& f_a \in \mathbb{N} && \text{for all } a \in A, \\
& z_e \in \mathbb{B} && \text{for all } e \in E, \\
& y_v \in \mathbb{B} && \text{for all } v \in V, \\
& x_v \in \mathbb{B} && \text{for all } v \in V.
\end{aligned}
\tag{FLOW}
$$

Lemma 7.9. *IP* (FLOW) *returns an optimal solution for the* Weighted Induced Subgraph Problem. *If the constraints* (7.2) *are dropped an optimal solution for* Weighted Subgraph Problem *is returned.*

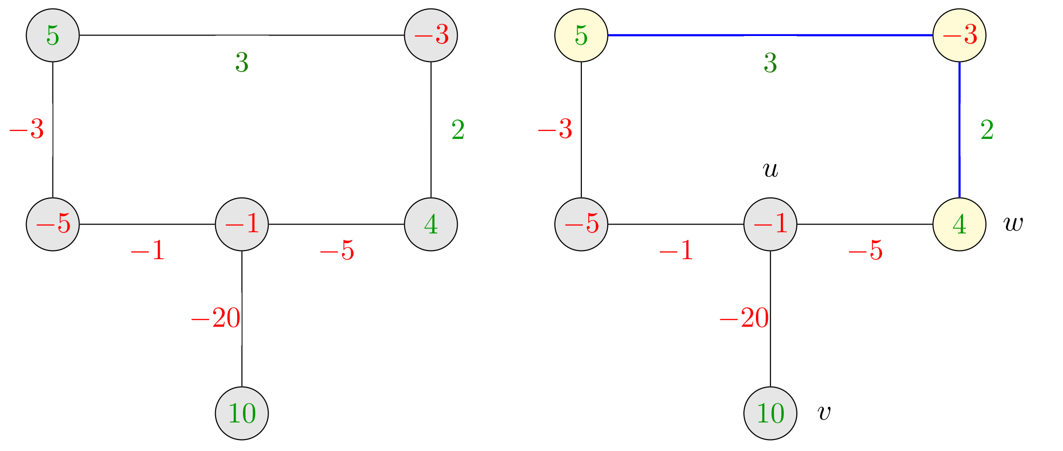
Proof. By Lemma 7.7 it remains to show that constraints (7.7) ensure graph connectivity. Due to the first inequality in (7.7) the chosen root node has to send flow of value at most $|V| - 1$ and all other chosen nodes have to receive at least one unit of flow. The argumentation works as in Theorem 7.8. □

Due to IP (FLOW$_r$) and IP (FLOW) having only polynomially many constraints we get the following:

Theorem 7.10. *The relaxations of IP* (FLOW$_r$) *and IP* (FLOW) *are solvable in polynomial time.*

Proof. The IP (FLOW$_r$) has $|V| + 3 \cdot |E|$ variables and $|V| + 6 \cdot |E|$ constraints which are both polynomial in the size of the input. As a result, it can be solved in polynomial time. The IP (FLOW) has $2 \cdot |V| + 3 \cdot |E|$ variables and $2 \cdot |V| + 6 \cdot |E| + 1$ constraints, so it computes an optimal solution in polynomial time. □

When relaxing IP (FLOW$_r$) and IP (FLOW) we need to be careful. Suppose $G = (V, E, \gamma)$ is an undirected graph and G contains a node v with a very large positive weight M. This node is only connected to the rest of G via a path (or a single edge) P with a lot of negative weight $-N$ with $|N| >> |M|$. So in the integral case it would not be beneficial to add v and therefore add P to the subgraph H. But in the LP relaxation one can assign z_e a small value in such a way that $z_e \cdot |V| > 1$ for all $e \in P$. This allows for one unit of flow to reach v and therefore collecting weight M. In the process only a small portion of the negative weight $-N$ contributes to the value of the weight function, thus improving the objective value.



(a) Input graph G.

(b) Maximum weighted subgraph H of G of weight $w(H) = 11$.

Figure 7.3: Example for a graph with a large gap between the optimal integral solution and the optimal solution of the LP relaxation.

Example 7.11. Consider the graph G from Figure 7.3. Solving IP (FLOW) yields a maximum weighted $H = \text{MAX-WSP}(G)$ subgraph of G with is $w(H) = (5 - 3 + 4) + (3 + 2) = 11$. In contrast, the LP relaxation of IP (FLOW) is able to obtain a better solution by setting $y_u = 1, y_v = 1, z_{uw} = 0.1$ and $z_{uv} = 0.1$. The result is a subgraph H' of weight

$$w(H') = 11 - 0.1 \cdot 5 - 1 - 0.1 \cdot 20 + 10 = 17.5.$$

Thus we should put a high branch priority on the edge variables in order to reduce such a behavior.

7.4 Separation IP

To solve the IP (CUT$_r$) (or IP (CUT)) more efficiently we remove the connectivity constraint (7.5)

$$y_v \leq z(\delta(S)) + x(S) \qquad \text{for all } v \in V, \{v\} \subseteq S \subseteq V$$

and solve the remaining IP. We identify violated constraints by checking if the returned subgraph is connected. If it is, we are finished since we found an optimal solution. If not, then for at least one of the connected components $C_1, \ldots, C_k$ of H we have $y_v > z(\delta(S)) + x(S)$ for all vertices v in that component. This means the connectivity constraint does not hold for that connected component. In other words, there exists a subset $S \subseteq H, S = C_i$ for any $i \in \{1, \ldots, k\}$ such that $y_v > z(\delta(S)) + x(S)$. As a result, we add such violated constraints

$$y_v \leq z(\delta(S)) + x(S) \qquad \text{for all } v \in S.$$

to the formulation and resolve again. This is repeated until a connected subgraph is returned. We denote this IP by (SEP).

7.5 Restricting to k-Subgraphs

In the last few sections we have formulated integer programs for the Weighted Subgraph Problem, the Weighted Induced Subgraph Problem, and their rooted variants. In this section we briefly describe how to formulate programs for the WEIGHTED K-SUBGRAPH PROBLEM and the WEIGHTED INDUCED K-SUBGRAPH PROBLEM.

Let $1 \leq k \leq |V|$ be any number. The IP formulations above can be modified to solve WKSP(G, k) or WIKSP(G, k) by adding the following constraint:

$$\sum_{v \in V} y_v = k. \tag{7.8}$$

This constraint ensures that exactly k vertices are chosen to be the subgraph. The following result is a obvious consequence:

Corollary 7.12. *When adding constraint* (7.8) *to IP* (CUT) *or IP* (FLOW)*, an optimal solution for the* WEIGHTED INDUCED K-SUBGRAPH PROBLEM *or the* WEIGHTED K-SUBGRAPH PROBLEM *will be returned.*

Chapter 8

Experimental Study

The empirical part of this thesis systematically investigates our implementations of the algorithms for the WEIGHTED SUBGRAPH PROBLEM. In order to get an idea of the practical use of the presented algorithms, we compare them to the IP solver Gurobi[1]. Gurobi is a commercial solver but has free restricted licenses for academics.

8.1 Setup

Our experiments are conducted on random graphs of different classes. The vertices and edges are randomly weighted.

To measure the performance of each algorithm, we use Pythons $timer()$ and compare average running times. Our code was executed on a 4-core Intel machine with 16 GB of main memory. Unless otherwise noted, each series consists of 100 queries.

All our algorithms were implemented in Python and we used the packages Networkx[2] and gurobipy. Networkx contains useful classes and functions for working with graphs and gurobipy is a python wrapper to Gurobi. Our source code can be found on the CD handed in together with this thesis. All code was designed to run on one processor core only.

For some of our algorithms, data structures for rooted trees and binary trees were implemented and used. Furthermore, procedures that create random graphs of the desired class were implemented.

Networkx provides a large variety of built-in graph generators. For constructing random paths, trees and, (not necessarily connected) graphs we used some of these generators.

We describe how we constructed random series-parallel graphs with m edges. We construct a decomposition tree and the corresponding series-parallel graph in the following way. We begin with m leaves representing m edges. Those are also the current roots of the forest. In each step, we randomly choose 2 root vertices and add a father v to them. The father is a new root, while its children are no longer roots. Also, v is randomly assigned a binary operation (series composition or parallel composition). We stop when there is only one root in the forest, thus making it a tree. In each step, the number of roots in the forest is reduced by one. After $m-1$ steps we obtain a tree with m leaves corresponding to a decomposition tree for a series-parallel graph with m edges.

A series-parallel graph generated in this way has between 2 (if all compositions are parallel compositions) and $m+1$ vertices (if all compositions are series compositions). On average, a series-parallel graph with m edges constructed as described above has $n = 2m - \frac{3}{2}(m-1)$ vertices. Consequently, we generate series-parallel graphs with n

[1]https://www.gurobi.com/
[2]https://networkx.github.io/

nodes by generating series-parallel graphs with a random number of edges $m > n - 1$ until the number of desired nodes is achieved. The probability distribution over m is chosen in such a way that the expected number of edges is $2n - 3$.

The construction of random connected graphs with n nodes and m edges, where $n-1 \leq m \leq n \cdot (n-1)$, is done as follows: The general idea is to generate a (uniformly chosen) random spanning tree with n nodes and $n - 1$ edges. Until the requested number of edges has been reached, we add an edge between any two random nodes to obtain a connected graph. Constructing a uniform spanning tree is done by generating a random walk. We create two partitions, S and T. Initially we store all nodes in S. We then pick a random node from S, remove it and mark it as visited by putting it in T. In each step, we randomly pick the next node from the neighbors of the current node. If the new node is not in T, meaning it has not been visited, we add an edge from the current to the new node. We set the new node as the current node and repeat. We stop when S is empty. In each step of the loop, one edge is added to an unvisited node and thus not creating a cycle, while the number of elements in S is reduced by one. Since we chose and removed a random node from S before the loop, after $n - 1$ steps we obtain a spanning tree with n nodes and $n - 1$ edges. Finally, we add random edges until the number of desired edges is reached.

We did not experience any performance differences between minimizing and maximizing the objective value. Therefore we will focus on maximizing the objective value for the remainder of this chapter. For the sake of consistency, all node and edge weights are integers between -10 and 10 chosen uniformly at random. The unit of the running times we measured is seconds.

8.2 Improvement of the IP Formulation

As we described in Chapter 7 several IP formulations for the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem were implemented. For solving IPs, the IP solver Gurobi was used. This section compares the performance and computation times of these implementations.

The graph instances used are randomly generated connected graphs as described above. To that end, the number of edges m was chosen between $n - 1$ and $n \cdot (n - 1)$ uniformly at random.

n	IP CUT$_r$	IP CUT	IP FLOW$_r$	IP FLOW	IP SEP
5	0.015507	0.005125	0.013143	0.003008	0.002361
10	0.640264	0.265688	0.048111	0.004004	0.002554
11	1.796731	0.862912	0.059632	0.007191	0.003175
12	4.055318	1.776218	0.059833	0.008838	0.004095
13	10.146821	4.451217	0.075073	0.013947	0.004207
14	24.090539	9.247563	0.078361	0.014200	0.005257
15	60.995379	22.514286	0.104218	0.014576	0.008065
20	-	-	0.172314	0.025817	0.041567
30	-	-	0.449542	0.057043	0.050377
40	-	-	0.746555	0.102363	2.675402
50	-	-	1.031724	0.221115	4.249537
60	-	-	1.499296	0.231455	10.940920

Table 8.1: Computational performance on random instances of the IP formulations.

As we can see in table 8.1, both non-optimized IPs, namely IP CUT$_r$ and IP CUT, which have exponential many constraints cannot compete with the other IP formulations. Already for graphs with 10 vertices, they are significantly slower than IP FLOW$_r$ and IP FLOW. For graphs with more than 15 vertices, it was not uncommon to experience "out of memory" errors. Hence, we were not able to measure the performance of IP CUT$_r$ and IP CUT for graphs with more than 15 vertices. Both flow formulations, the rooted variant IP FLOW$_r$ and the unrooted variant IP FLOW were able to beat all other IP formulations, especially on larger graphs. The rooted version is slower by a factor of more than 10 which is probably due to the setup time because it has to construct and solve n IPs (one for each node in the graph). IP SEP started out being faster than both flow formulations, probably because it only needed a few iterations and thus the setup time was very low. But as the graphs get larger, a lot more iterations are needed and the computation time increased drastically. For graphs with 60 vertices, it was slower than both flow formulations by a factor of more than 6.

Figure 8.1 illustrates the running times of all algorithms using a logarithmic scale for the time needed.

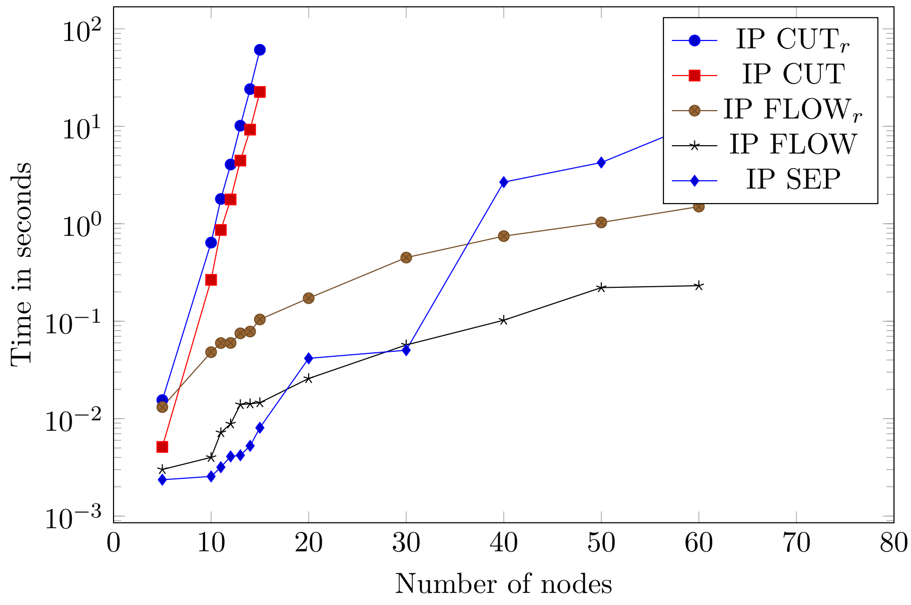


Figure 8.1: Illustration of the computation times of the IP formulations.

For the remainder of this chapter, we mainly use IP FLOW for measuring the performance of solving the WEIGHTED SUBGRAPH PROBLEM using Gurobi. Also, we noticed that graph density seems to have a big impact on the running time of all IP formulations. Since that impact was percentage-wise the same for all IP formulations, we did not consider it for this segment.

8.3 Performance of the Dynamic Program

In the last section, we have seen that the flow formulation (see IP FLOW) was the most efficient way of solving the WEIGHTED SUBGRAPH PROBLEM using Gurobi. In this section, the performance of the dynamic program from section 3 is compared to the performance of Gurobi.

We implemented the dynamic program from Chapter 4 for paths, trees, and series-parallel graphs. To compare the performance of the implementations we randomly generated graphs of those classes with different sizes. The size of a graph is given by the number of nodes in the graph.

In Table 8.2 average running times of the algorithms on paths, trees, and series-parallel graphs are shown.

	paths		trees		series-parallel graphs	
n	dynamic prog	IP FLOW	dynamic prog	IP FLOW	dynamic prog	IP FLOW
10	0.000106	0.013639	0.000194	0.005969	0.000656	0.008037
20	0.000160	0.034162	0.000386	0.024114	0.001478	0.056467
30	0.000220	0.120945	0.000662	0.086717	0.002345	0.070305
40	0.000270	0.340605	0.000886	0.138077	0.003098	0.152056
50	0.000322	0.863015	0.001063	0.316319	0.003952	0.230241
60	0.000376	2.099835	0.001363	0.990999	0.004976	1.349487
75	0.000603	11.706951	0.001930	1.576299	0.006210	4.346551
100	0.000627	28.071109	0.002127	4.881766	0.008490	10.715087

Table 8.2: Computational performance of the dynamic program on random paths, trees and series-parallel graphs.

As we can see in Table 8.2 our procedure is significantly faster than Gurobi on all instances. By increasing the number of vertices in the graph, the dynamic program really shows its power. Already for graphs with 30 or fewer vertices, significant differences in computation times can be seen. For a graph with 100 vertices, the computation time is faster by a factor of more than 2000. For paths and series-parallel graphs this factor is even larger.

In the next series, we wanted to examine the influence of graph density on the performance of the dynamic program. For this, we generated random series-parallel graphs with m edges until the desired number of vertices n was achieved.

n	m	dynamic prog	IP FLOW
50	50	0.002479	0.629711
50	75	0.003967	0.102970
50	100	0.005365	0.061309
50	250	0.014865	0.134188
50	500	0.028244	1.047685
50	1000	0.063950	13.621358

n	m	dynamic prog	IP FLOW
100	100	0.005209	5.852813
100	250	0.014053	0.250138
100	500	0.027513	0.616554
100	1000	0.067332	26.753307

Table 8.3: Impact of graph density on the computational performance.

Turns out, that density has a linear impact on the performance of the dynamic program which was to be expected since its theoretical running time is $\mathcal{O}(m)$. In Table 8.3 we can see that the running time seems to double every time the number of edges is doubled. In all cases, the dynamic program is faster than Gurobi. For rather sparse and dense graphs this gap becomes larger. For a graph with 50 vertices and 1000 edges the dynamic program is faster than Guribi by a factor of more than 200.

8.4 Polyhedral Results

In this section, we compare the proposed IP formulations practically with respect to their quality of LP bounds. In the following, let $\mathcal{P}_{LP}(\cdot)$ denote the polytopes of the

LP-relaxations of the IP models presented in section 7 and $\text{OPT}_{LP}(\cdot)$ their optimal LP-values. Moreover, let OPT_{IP} denote the objective value of the IP.

We generated random connected graphs with n nodes. Hereby, the number of edges m was chosen between $n-1$ and $n \cdot (n-1)$ uniformly at random. In Table 8.4 the objective value and the gap (as percentages) of each instance is shown:

	LP CUT		LP CUT$_r$		LP FLOW	
OPT_{IP}	OPT_{LP}(CUT)	gap [%]	OPT_{LP}(CUT$_r$)	gap [%]	OPT_{LP}(FLOW)	gap [%]
24	24.50	2.04	27.59	13.00	29.00	17.24
29	29.00	0.00	29.00	0.00	29.00	0.00
14	17.50	20.00	22.11	36.68	29.00	51.72
29	29.50	1.69	30.86	6.01	31.50	7.94
33	33.50	1.49	33.90	2.65	34.00	2.94
31	31.50	1.59	31.89	2.79	32.00	3.12
12	12.00	0.00	12.50	4.00	13.00	7.69
23	23.00	0.00	23.00	0.00	23.00	0.00
22	30.00	26.67	36.11	39.08	41.00	46.34
54	55.50	2.70	55.50	2.70	55.50	2.70
16	16.00	0.00	16.77	4.59	18.00	11.11
47	47.00	0.00	47.00	0.00	47.00	0.00
40	40.00	0.00	40.00	0.00	40.00	0.00
56	57.50	2.61	65.57	14.59	68.00	17.65
47	47.00	0.00	47.00	0.00	47.00	0.00

Table 8.4: Gaps [%] reached by the LP relaxations.

In Table 8.4 we report the results obtained for the LP relaxations. The gap between the optimal objective value and the LP bound varies greatly between 0 and more than 50 percent. Both heuristics were able to obtain the optimal objective value for some instances. In general, OPT_{LP}(CUT) reaches the smallest gaps with an average of 3.91 percent, the maximal gap being 26.67 percent. It is followed by $\text{OPT}_{LP}(\text{CUT}_r)$ with an average of 8.40 percent, and a maximum of 39.08 percent. Finally OPT_{LP}(FLOW) reaches an average gap of 11.23 percent and a maximal gap of 51.72 percent.

For all instances, we can see that the objective value of the flow formulation is larger than the value of the rooted cut formulation. Furthermore, the objective value of the rooted cut formulation is larger than the value of the unrooted cut formulation. This lets us suspect that

$$\text{OPT}_{LP}(\text{CUT}) \leq \text{OPT}_{LP}(\text{CUT}_r) \leq \text{OPT}_{LP}(\text{FLOW})$$

which also gives the impression that

$$\mathcal{P}_{LP}(\text{CUT}) \subseteq \mathcal{P}_{LP}(\text{CUT}_r) \subseteq \mathcal{P}_{LP}(\text{FLOW}).$$

8.5 Preprocessing

We continue by investigating whether the preprocessing scheme from section 6.3 yields a noticeable benefit. How big is the reduced instance compared to the original instance? How large is the performance boost when using the preprocessing scheme, if any?

For this segment, we generated random connected graphs (with parallel edges) as described in Section 8.1. The following table 8.5 enlists the sizes of the original instances compared to the average sizes of the reduced instances.

Original instance		Reduced instance	
n	m	n	m
10	20	2.28	1.28
10	30	4.74	4.43
10	40	2.93	1.54
50	100	15.35	22.80
50	150	7.90	6.13
50	200	2.21	1.61
75	150	20.00	36.34
75	225	7.12	6.43
75	300	4.09	3.65

Original instance		Reduced instance	
n	m	n	m
100	200	29.22	42.17
100	300	12.92	15.76
100	400	9.64	8.11
250	500	70.05	103.38
250	750	37.89	46.32
250	1000	14.90	16.75
500	1000	150.91	249.41
500	1500	57.14	75.07
500	2000	25.18	32.69

Table 8.5: Average graph size of reduced instance obtained by applying the preprocessing scheme from section 6.3 compared to the original graph size.

Of course, these results depend on the weights in the graph. Table 8.5 shows that there is a correlation between the size of the reduced instance and the density of the original instance. The denser the input graph is, the smaller the reduced graph will be. This is probably caused by the fact, that weights on vertices and edges use the same weight distribution. Also, the number of parallel edges in the graph is higher which is utilized by the preprocessing procedure. In general, graphs are reduced by at least 50 percent for all instances that we tested.

For the performance measurement, we solved an instance as follows. First, we solved the instance to optimality and measured the time needed. Then we applied the preprocessing scheme to the same graph and solved the reduced instance. The time measured includes the preprocessing scheme as well as the time needed to solve the reduced instance. For computing a solution IP (FLOW) was used. Table 8.6 shows the computations times for both cases.

n	m	with preprocessing	without preprocessing
10	20	0.004879	0.005269
10	30	0.006946	0.007743
10	40	0.007620	0.010667
50	100	0.051437	0.060431
50	150	0.045415	0.047085
50	200	0.057990	0.066576
100	200	0.185324	0.202422
100	300	0.102997	0.115853
100	400	0.150227	0.186277
250	500	1.330858	1.456818
250	750	0.495078	0.508311
250	1000	0.942581	1.025909
500	1000	4.531664	5.518316
500	1500	2.875006	2.007461
500	2000	3.938611	4.455854

Table 8.6: Computational performance when using the preprocessing scheme from section 6.3.

In Table 8.6 we observe that on average, using the preprocessing scheme yields a noticeable performance boost. As the instances get larger, the decrease in computation time is more substantial. For graphs with 100 vertices or less the difference is not very large. But for graphs with 500, vertices the speedup in computation time is almost 1 second. In general, using the preprocessing procedure seems to decrease the running time by almost 10 percent.

8.6 Heuristics

Continuing, we would like to compare the proposed heuristic algorithms and the quality of their solutions. In the following, let HEU$_{NS}$ denote Procedure 6.7 and HEU$_{ST}$ the procedure from section 6.6.

First, let us start with the computational performance of the heuristic solutions. For this test randomly generated connected graphs with n nodes were used. As previously, the number of edges m was chosen between $n-1$ and $n \cdot (n-1)$ uniformly at random. For HEU$_{NS}$ we used IP FLOW to solve the modified version of WSP (see step 4 in Procedure 6.7). From the results above we conduct that IP FLOW can be efficiently solved for graphs with 100 vertices. Thus, we set $N := 100$ as an upper bound of the number of nodes for the modified instance.

n	IP FLOW	HEU$_{NS}$	HEU$_{ST}$
10	0.008435	0.003538	0.000870
25	0.024684	0.012868	0.002372
50	0.047114	0.075030	0.004126
100	0.204004	0.393630	0.010188
250	0.964690	4.934293	0.035741
500	6.725403	36.911216	0.107158
1000	81.908922	297.658889	0.427159

Table 8.7: Computational performance of the heuristic procedures compared to IP (FLOW).

Table 8.7 presents the running times of Gurobi using the flow formulation (FLOW) and of both heuristic methods HEU$_{NS}$ and HEU$_{ST}$. The node set heuristic presented in section 6.5 cannot compete with Gurobi. Already for graphs with 250 vertices, significant differences in computation times can be seen. This is caused by a large number of inequalities in the modified instance of the problem. We fixed a set of the nodes to be in the solution. Thus, we reduced the number of nodes Gurobi has to select, but not the number of edges. This behavior gets worse if we increase the number of nodes. For a graph with 1000 vertices, the method is worse than Gurobi by a factor of 300. Hence, we conclude that Procedure 6.7 is not of practical relevance. In contrast, HEU$_{ST}$ can be computed very efficiently even for large instances. For only 50 vertices the difference is the procedure from section 6.6 is faster than Gurobi by a factor of 10. For 500 vertices the factor has already increased to 60 and for 1000 vertices the procedure is faster by a factor of 200.

Now we want to regard the solution quality of HEU$_{NS}$ and HEU$_{ST}$. For each heuristic Table 8.8 shows the minimal, maximal, and average gap reached within 100 iterations.

	HEU$_{NS}$		
n	max. gap [%]	min. gap [%]	avg. gap [%]
10	24.44	0.00	2.69
25	16.54	0.00	2.73
50	9.51	0.00	2.68
100	9.31	0.69	3.80
250	10.10	2.36	4.99
500	7.79	4.33	6.70
1000	7.56	4.68	6.05

	HEU$_{ST}$		
n	max. gap [%]	min. gap [%]	avg. gap [%]
10	33.33	0.00	9.31
25	32.71	1.68	12.45
50	23.08	5.56	13.05
100	24.40	6.49	13.66
250	17.50	8.90	12.84
500	15.30	10.79	12.68
1000	15.23	11.46	13.31

Table 8.8: Minimal, maximal, and average gaps reached by the heuristic procedures on graphs of different size.

We have seen above that the computation time of HEU$_{NS}$ cannot compete with Gurobi. Nevertheless, Table 8.8 shows that the solution quality is impressive. Its average gap to OPT$_{IP}$ is approximately 12.5 percent. The maximal gap decreases from 33.33 percent on graphs with 10 nodes to 15.23 percent on graphs with 1000 nodes while the minimal gap increases from 0 to 11.46 percent for the instances we tested. For almost all instances HEU$_{NS}$ obtains a better heuristic solution than HEU$_{ST}$. Its average gap to OPT$_{IP}$ is approximately 3 percent. For small instances of up to 50 vertices, the procedure from section 6.6 was able to obtain the optimal objective value. For bigger instances, the maximal gap is at most 10.10 percent on the instances we tested, while its minimal gap increased to almost 5 percent.

Therefore, we conclude that HEU$_{ST}$ can be used to obtain decent heuristic solutions and lower bounds for the optimal objective value. Although HEU$_{NS}$ reaches tighter bounds in terms of the objective value, we do not recommend using it due to its bad computation time.

Chapter 9

Conclusion

Throughout the chapters of this thesis, we have seen that the Weighted Subgraph Problem can be studied in many different variations. We showed that all of those variations are $\mathcal{NP}$-complete. Nevertheless, we managed to solve the problem efficiently on certain classes of graphs, including paths, trees, series-parallel, and decomposable graphs. This was achieved by constructing a polynomial time dynamic programming algorithm. Also, several integer programming formulations for the problem and its variations were given.

Furthermore, we showed that the Weighted Subgraph Problem and its variations are $\mathcal{NP}$-complete to approximate within a constant factor. Unfortunately, we were not able to develop an approximation algorithm, but we proposed two heuristic procedures. Moreover, a preprocessing scheme for reducing instances and a postprocessing procedure for improving heuristic solutions were presented.

In the empirical part of this thesis, the dynamic programming algorithm and the spanning tree heuristic turned out to be a lot faster than Gurobi. Besides the practical use of the procedures, it is a nice theoretical result that in some cases we can solve the Weighted Subgraph Problem and the Weighted Induced Subgraph Problem in polynomial or even linear time.

It remains open to give an approximation algorithm for the Weighted Subgraph Problem with a provable ratio. In the time given to write this thesis, we did not manage to achieve this.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0-13-617549-X.

[BLW87] M.W Bern, E.L Lawler, and A.L Wong. “Linear-time computation of optimal subgraphs of decomposable graphs”. In: *Journal of Algorithms* 8.2 (1987), pp. 216–235. ISSN: 0196-6774.

[Bod96] Hans Bodlaender. “A Partial K-Arboretum of Graphs With Bounded Treewidth”. In: *Theoretical Computer Science* 209 (Feb. 1996), pp. 1–45.

[EK14] Mohammed El-Kebir and Gunnar W. Klau. “Solving the Maximum-Weight Connected Subgraph Problem to Optimality”. In: *CoRR* abs/1409.5308 (2014).

[FS97] Uriel Feige and Michael Seltser. “On the Densest K-Subgraph Problem”. In: *Algorithmica* 29 (1997), p. 2001.

[Klo94] Ton Kloks. *Treewidth, Computations and Approximations*. Vol. 842. Lecture Notes in Computer Science. Springer, 1994. ISBN: 3-540-58356-4.

[KN12] S.O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. XLeitfäden der Informatik. Vieweg+Teubner Verlag, 2012. ISBN: 9783322921123.

[KP93] G. Kortsarz and D. Peleg. “On choosing a dense subgraph”. In: (Nov. 1993), pp. 692–701.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Reading, MA: Addison-Wesley, 1994.

[RRT94] S Ravi, Daniel Rosenkrantz, and Giri Tayi. “Heuristic and Special Case Algorithms for Dispersion Problems”. In: *Operations Research* 42 (Apr. 1994), pp. 299–310.

[Tam91] A. Tamir. “Obnoxious Facility Location on Graphs”. In: *SIAM Journal on Discrete Mathematics* 4.4 (1991), pp. 550–567.

[VTL79] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. “The Recognition of Series Parallel Digraphs”. In: STOC ’79 (1979), pp. 1–12.

[Wim87] Thomas Victor Wimer. “Linear Algorithms on K-terminal Graphs”. AAI8803914. PhD thesis. Clemson, SC, USA, 1987.

[Wol98] L.A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998. ISBN: 9780471283669.

Chapter A

Definitions

Definition A.1 (Partition)**.** A *partition* of a set X is a set of nonempty subsets of X such that every element $x \in X$ is in exactly one of these subsets, i.e. the set X is a disjoint union of the subsets. Equivalently, a family of sets P is a partition of X if and only if all of the following conditions hold:

(i) $\emptyset \notin P$,

(ii) $\bigcup_{A \in P} A = X$,

(iii) $A \cap B = \emptyset$ for all $A, B \in P$ with $A \neq B$.

The sets in P are said to cover X. The *rank* of P is $|X| - |P|$, if X is finite. If P_1 and P_2 are partitions of X and every set of P_2 is a subset of some set of P_1, we call P_2 a *refinement* of P_1. We denote the number of partitions of n by p_n.

Definition A.2 (Bell Number)**.** The *Bell number* B_n is the number of partitions of a set with n elements. The Bell numbers satisfy a recurrence relation involving binomial coefficients:

$$\begin{aligned} B_0 &= B_1 = 1, \\ B_n &= \sum_{k=0}^{n-1} \binom{n-1}{k} B_k. \end{aligned}$$

Formulas for the n-th Bell number are:

$$\begin{aligned} B_n &= \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}, \\ B_n &= \sum_{k=1}^{n} \frac{k^n}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!}. \end{aligned}$$

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. This applies also to all graphics, drawings, maps and images included in the thesis. This thesis was not previously presented to another examination board and has not been published.

Kaiserslautern, the 29th of March, 2019

(Felix Hoffmann)