Node–Disjoint Length–Restricted Paths

Diplomarbeit

von

Andreas Bley

Januar 1997

Technische Universität Berlin
Fachbereich Mathematik
Studiengang Wirtschaftsmathematik

Erstgutachter: Prof. Dr. M. Grötschel
Zweitgutachter: Prof. Dr. G. M. Ziegler

Zusammenfassung

In dieser Diplomarbeit befassen wir uns mit verschiedenen Zusammenhangsproblemen in Graphen. In diesen speziell für den Entwurf von Telekommunikationsnetzwerken interessanten Problemen werden Fragestellungen des Knotenzusammenhangs mit Pfadlängenbeschränkung in einem Graphen betrachtet. Diese Verknüpfung ist besonders dann interessant, wenn Telekommunikationsnetzwerke für ressourcenintensive Nachrichtendienste, wie zum Beispiel Videokonferenzen, kostengünstig zu entwerfen sind. In solchen Fällen ist neben einer entsprechenden Ausfallsicherheit, die i.d.R. durch mehrere parallele Verbindungen zwischen den betroffenen Zielorten garantiert wird, auch noch eine Beschränkung für die Anzahl der Vermittlungsknoten auf diesen Verbindungen zu beachten, da sonst keine ausreichende Vermittlungsqualität garantiert werden kann.

Nach einer Erläuterung der praktischen Hintergründe und der Anwendungsmöglichkeiten unserer Ergebnisse sowie einer kurzen Einführung in die für uns relevanten Gebiete der Graphentheorie werden im ersten Kapitel die uns in dieser Arbeit interessierenden Probleme definiert. Für jedes dieser Probleme wird eine Modellierung als ganzzahliges linerares Programm angegeben.

Im ersten Teil des zweiten Kapitels wird eine Einführung in den für uns relevanten Teil der Komplexitätstheorie gegeben. Diese Einführung beinhaltet sowohl die Definitionen der hier interessierenden Komplexitätsklassen $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{PO}$, $\mathcal{NPO}$, $\mathcal{APX}$ sowie einiger weiterer Approximationsklassen, als auch eine Darstellung der für uns wesentlichen Resultate. Im Anschluß an diese Darstellung werden zahlreiche Komplexitätsaussagen für unsere Probleme getroffen und bewiesen. In diesem Kapitel, welches der erste Hauptteil dieser Diplomarbeit ist, zeigen wir zum Beispiel, daß das Problem, zwischen zwei Knoten in einem Graphen die maximale Anzahl von knotendisjunkten längenbeschränkten Wegen zu bestimmen, für allgemeine Längenbeschränkungen $\mathcal{NPO}$-vollständig ist, ein bis dahin nicht bekanntes Resultat. Zwei weitere bisher unbekannte Resultate, nämlich daß das Problem, in einem Graphen mit nichtnegativen Kantengewichten eine kostenminimale Menge von k knotendisjunkten längenbeschränkten Wegen zwischen zwei Knoten zu finden, bereits ab einer Längenbeschränkung von fünf Kanten pro Weg für allgemeine k $\mathcal{NPO}$-vollständig und nicht mit polynomial kodierbarer Gütegarantie approximierbar ist, falls $\mathcal{P} \neq \mathcal{NP}$, werden ebenfalls bewiesen. Weiterhin werden in diesem Teil der Arbeit noch Beweise für die polynomiale Lösbarkeit einiger Spezialfälle der in Kapitel 1 definierten

Probleme gegeben und einige andere bereits bekannte Komplexitätsaussagen für diese Probleme bewiesen.

Im dritten Kapitel werden einige Resultate aus der Literatur vorgestellt, die in direktem Zusammenhang zu dieser Arbeit stehen.

Im zweiten Hauptteil dieser Diplomarbeit, den Kapiteln 4 und 5, wird ein kompletter algorithmischer Apparat zur Behandlung der im ersten Kapitel eingeführten Probleme entwickelt. Dabei befassen wir uns im vierten Kapitel mit den Problemen, in denen "nur" ein Demand zu berücksichtigen ist. Ein Demand ist ein abstrakter Kommunikationsbedarf, der durch seine beiden Zielknoten, eine Mindestanzahl von knotendisjunkten Wegen zwischen diesen Knoten und eine Längenbeschränkung für diese Wege definiert ist. Wir stellen zahlreiche algorithmische Ansätze für diese sogenannten single-demand Probleme vor. Das Spektrum reicht dabei von Preprocessing Methoden über Netzwerkfluß Techniken und speziellen Algorithmen für längenbeschränkte Augmentierung gegebener nichtoptimaler Lösungen bis hin zu vollständigen Enumerationsverfahren. Sofern möglich, werden Gütegarantien für die Algorithmen angegeben.

Im fünften und letzten Kapitel werden Algorithmen für die multi-demand Probleme, in denen mehrere Demands zu berücksichtigen sind, vorgestellt. Diese Algorithmen nutzen die im vierten Kapitel entwickelten Methoden. Wir stellen zwei Klassen von Heuristiken und ein exaktes Verfahren zur Lösung des gewichteten multi-demand Problems und einen Algorithmus zur Lösung des multi-demand Zulässigkeitsproblems vor.

Im Anhang sind einige praktische Ergebnisse unserer Algorithmen dargestellt. Diese Ergebnisse zeigen die Effizienz der von uns entwickelten algorithmischen Vorgehensweisen an praktischen Daten.

Acknowledgments

This thesis would not have been possible without the help of some people, whom I would like to acknowledge here.

I am very grateful to my supervisor Martin Grötschel and to my advisor Roland Wessäly for their accessability and support. It was a pleasure to work with them.

Many people made my years at the TU–Berlin pleasant and interesting. Thanks to the people of the Discrete Mathematics Group at TU–Berlin for many interesting lectures and discussions. Also thanks for some nice basketball games at the end of every week. The people at ZIB deserve credit as well for supporting my research. Especially, I would like to thank Roland Wessäly and Andreas Eisenblätter, who read through various drafts and provided many useful comments.

Thanks to all my friends who kept things lively, especially to Raimo and to Margherita for the nights in Deponie No. 3, discussions about music, software, and movies, cycle trips around the town, and so on. Special thanks go to Alexander Below for reading various drafts and asking the right questions.

Finally, I am very grateful to my family. They have always been there for me.

Contents

Chapter 0

Introduction

0.1 Motivation

The various disjoint path problems are among the main topics in network design. They have a wide range of practical applications, ranging from telecommunications to transport problems.

In the last decade the state-of-the-art in communication technologies has rapidly changed. Many of the global service providers introduced fiber optic technology. This technology, due to its enormous transmission capacity and speed, offered the prospect of ushering in many new information network services, for example video-on-demand and video teleconferencing. The immense requirements of capacity and speed made these services practically impossible in the pre-fiber-optics times.

On the other hand, the installation of fiber optic technologies bears business risk for the companies. Thus, due to the power and of course due to the big costs of the new technology, the companies tend to design very sparse long-range fiber optic networks connecting only the most important locations. In these networks there are only few fiber optic paths between two such main locations. Usually the locals are linked to the nearest main location in a star-like manner by classical copper lines. This is quite different from the mesh-like topology of classical copper-based networks.

Although these sparse fiber optic networks are sufficient as long as everything works fine, they may collapse in the case of a failure. Since global connection is guaranteed by only a few fiber optic lines, a failure of those lines or the switches can result in traffic loss or could even disconnect some parts of the network. Hence, it is necessary to take into account such failure situations when designing such a network. Several "survivability" concepts are considered in the literature in order to avoid such catastrophic failures. Mostly, certain edge- or node-connectivities are required in the network (see [GMS95]).

For some services there is another important constraint, that must be considered when designing a communication network. Especially for services as video teleconferencing or digital speech transmission (and sometimes even e-mail), the transmission time is one of the most important keys for customer satisfaction. The transmission speed in the fiber lines is given by the technology. Therefore, the number of switches used for transmitting the data is the critical factor. This leads to the concept of short path connectivity, where the number of lines (or the number of switches) for each transmission path is bounded.

In our work we try to join the two concepts. Originally, the work was motivated by a project with **e-plus** Mobilfunk GmbH, one of the mobile phone providers in Germany. In this project a network dimensioning tool called DISCNET—DImensioning of Survivable Cellular phone NETworks—was developed for the use at **e-plus** (see [AGW96], [AW96b], [AW96a]). The problems considered in our thesis correspond to the survivability concepts used in the project with **e-plus**. We are given a graph, that is an abstract telecommunication network, and a set of demands. Each demand between two vertices of the graph has its individual diversification parameter requiring a specified number of vertex-disjoint paths between the two vertices, and its individual path length restriction limiting the length of these paths. The results of our work will be used to solve the survivability problem in DISCNET.

In Chapter 1 we formally define the problems discussed in this thesis. In Chapter 2 the complexity of these problems is analyzed. We will show our main theoretical results in this chapter. Namely, we show various hardness results for the general problems and prove the easiness of some special cases. Some theoretical results from the literature dealing with the duality of connecting path sets and separating vertex sets are presented in Chapter 3. In Chapter 4 and in Chapter 5 we present our algorithmic approaches to the single-demand and multi-demand problems, respectively. This includes the high-level description of the heuristics and exact algorithms and frameworks describing the way these algorithms work together. Some computational results are displayed in the appendix.

0.2 Definitions

Formally, a **graph** is a triple $G = (V, E, \Psi)$ consisting of a nonempty set V, called the **vertices** (or nodes), a set E, called the **edges** (or lines), and a relation of **incidence** $\Psi : E \to V^2$ that associates with each edge two vertices, called its **ends**. Usually, we just write $G = (V, E)$ and assume that the incidence relation is given implicitly in E.

Two vertices that are the ends of an edge are **adjacent** to one another (**neighbors**). A vertex that has no neighbors is **isolated**. The **degree** $|\delta(v)|$ of a vertex v is the number of edges incident to v. An edge with identical ends is called a **loop**. If two edges join the same pair of ends, they are called **parallel**. A graph is **simple** if it has neither loops nor parallel edges.

Since some of our algorithms work on digraphs only, we will give the analogous definitions for the directed case.

A **digraph** (directed graph) is a triple $D = (V, A, \Psi)$ consisting of a nonempty set V, called the **vertices** (or nodes), a set A, called the **arcs**, and a relation of **incidence** $\Psi : A \to V^2$ that associates with each arc an ordered pair of vertices called its **ends**. Usually, we just write $D = (V, A)$ and assume that the incidence relation is given implicitly in A. For each arc $a = (v, w)$ we call v the **tail** and w the **head** of a. Parallel arcs and loops are defined as for graphs. Two arcs $a = (v, w)$ and $b = (w, v)$ are called **antiparallel**. We call the graph $G = (V, E)$ the **underlying graph** of the digraph $D = (V, A)$ if there is a bijection between the arcs of D and the edges of G, such that for each arc $a = (v, w) \in A$ there is an edge $e = (v, w) \in E$ and for each edge $e = (v, w) \in E$ the arc $a = (v, w)$ or the arc $a' = (w, v)$ is in A. The **associated digraph** $D(G)$ of a graph G is the digraph obtained from G by replacing each edge by two antiparallel arcs with the same ends.

In the following, let $G = (V, E)$ denote a graph and $D = (V, A)$ a digraph.

The number of vertices $|V|$ of G (or D) is called the **order** of G (or D). If both V and E (or V and A) are finite sets, then $G = (V, E)$ (or $D = (V, A)$) is called **finite**.

We will consider only simple finite graphs and digraphs in this thesis, because this is sufficient for our purposes.

A **walk** in G (or a directed walk in D) is a sequence $W = v_0 e_1 v_1 \dots e_l v_l$ of vertices $v_0, \dots, v_l$ and edges (arcs) $e_1, \dots, e_l$ of G (of D), such that the vertices v_{i-1} and v_i are the ends of edge e_i (are head and tail of e_i) for each $1 \leq i \leq l$. Vertex v_0 is called the **tail** of W, vertex v_l is called the **head** of W, and all other vertices are called **internal vertices** of W. We say that W is a (v_0, v_l)**-(di)walk** or that W connects v_0 and v_l. Given a weight function $c : E \to \mathbb{N}$ on the edges (or $c : A \to \mathbb{N}$ on the arcs), the **weight** $c(W)$ of a walk W is the sum of the weights of the edges (or arcs) in W. The number of edges (or arcs) in W is its **length**. We call a walk W l-bounded if its length does not exceed l. A (di)walk W is called a (directed) **path** if all its vertices $v_0, \ldots, v_l$ are distinct. A (di)walk W with $v_0 = v_l$ is **closed**. A closed (di)walk, whose edges (or arcs) $e_1, \ldots, e_l$ are distinct, is a **circuit** (or directed circuit).

The members of a collection of (directed) (s,t)-paths are **vertex-disjoint** or **internally disjoint** if no two have an internal vertex in common. We say a collection of (directed) (s,t)-paths is **edge-disjoint** (or **arc-disjoint**) if no two paths in the collection have an edge (an arc) in common.

A graph $\hat{G} = (\hat{V}, \hat{E})$ is a **subgraph** of $G = (V, E)$ if $\hat{V} \subseteq V$ and $\hat{E} \subseteq E$. For a graph $G = (V, E)$ and $X \subseteq V$ we denote by $G - X$ the graph constructed by removing all vertices $v \in X$ and all edges incident to at least one of these vertices from G. By $G[W] := G - (V \setminus W)$ we denote the subgraph **induced** by W. Analogously, for $F \subseteq E$ we denote by $G - F$ the graph constructed by removing all edges $e \in F$ from G. The definitions for the directed case are analogous.

A graph $G = (V, E)$ is said to be **connected** if there is a path between any two vertices. A graph G is **k-connected** if $|V| > k$ and for any $X \subseteq V$ with $|X| \leq k - 1$ the graph $G - X$ is connected. This is equivalent to saying that there are at least k vertex-disjoint paths between any pair of vertices of G. Analogously, G is called **k-edge-connected** if the deletion of any set of at most $k - 1$ edges leaves a connected graph. A digraph D is said to be **strongly connected** if there is a directed path from any vertex to any other vertex; D is **strongly k-connected** if $|V| > k$ and $G - X$ is strongly connected for any $X \subseteq V$ with $|X| \leq k - 1$.

The **components** of a graph $G = (V, E)$ are the (inclusion) maximal connected subgraphs of G. The **strong components** of a digraph $D = (V, A)$ are the maximal strongly connected subgraphs of D.

If $G = (V, E)$ is a graph and $X \subseteq V$, then the set of edges $\delta(X) := \{(v, w) \in E \,|\, v \in X, w \notin X\}$ is a **cut**. If $s \in X$ and $t \notin X$ we call $\delta(X)$ an **(s,t)-cut**. For a digraph $D = (V, A)$ and a subset $X \subseteq V$ of the vertices let $\delta^+(X) := \{(v, w) \in A \,|\, v \in X, w \notin X\}$, $\delta^-(X) := \delta^+(V \setminus X)$, and $\delta(X) :=$

$\delta^+(X) \cup \delta^-(X)$. The arcset $\delta^+(X)$ is called a **directed cut**. For $X, Y \subseteq V$ we define $[X, Y] := \{(x, y) \in A \,|\, x \in X, y \in Y\}$. A vertex set $S \subseteq V$ is a **stable set** if $[S, S] = \emptyset$.

Analogously, if $G = (V, E)$ is a graph and X, S, Y is a partition of V with $X, Y \neq \emptyset$ and there is no edge between X and Y, we call S a **vertex cut**. This is equivalent to saying that S is a vertex cut iff $G - S$ is disconnected. If $s \in X$ and $t \in Y$, then we call S an **(s,t)-vertex cut** or say that S **separates** s and t. Vertex cuts are also called **articulation sets**.

Chapter 1

The Demand Problems

In this chapter we give an overview of the problems discussed in this thesis. We will formally define the problems and give an integer (linear) programming model for each problem.

We consider only vertex-disjoint path problems, but similar results can be shown for edge-disjoint path problems.

For all problems the input consists of a simple undirected graph $G = (V, E)$ (the supply graph representing the network) and a set D of one or more demands. A **demand** is a connectivity requirement between two vertices (nodes) of the graph. It is characterized by its individual **length restriction** and its **diversification parameter**. Let $n = |V|$ and $m = |E|$. Formally, a demand $d \in D$ consists of four parameters:

- s^d, $t^d \in V$: the two nodes of the demand (source and target),
- k^d, $1 \leq k^d \leq n-1$: the diversification parameter, specifying the minimum number of paths to be used to satisfy the demand, and
- l^d, $1 \leq l^d \leq n-1$: the length bound (number of edges) for these paths.

An (s^d, t^d)-path is called **valid** with respect to a demand d, if the path is an l^d-bounded (s^d, t^d)-path, i.e., a simple (s^d, t^d)-path of length at most l^d. For every demand $d \in D$ we denote the set of valid paths by $\mathcal{Q}^d$. Let I^d be an index set of $\mathcal{Q}^d$. A subset $Q \subseteq \mathcal{Q}^d$ of valid paths for d is called valid or vertex-disjoint, if all paths in Q are pairwise vertex-disjoint except in s^d and t^d (internally-disjoint). We say that a demand d is satisfied by a subgraph

S of G, if there are at least k^d valid paths (vertex-disjoint and l^d-bounded) from s^d to t^d in S.

The problems discussed in the following chapters are grouped into **single-demand problems** and **multi-demand problems**. In the single-demand problems there is exactly one demand to satisfy in the graph, whereas in the multi-demand problems we have several demands, that have to be satisfied simultaneously.

For each of these two groups we study the following two problems:

1. The **feasibility problem**, asking whether it is possible to satisfy the given demands in the graph.
2. The **weighted (or design) problems**, where the aim is to choose a cost minimal subgraph that satisfies the given demands with respect to some given edge costs.

Furthermore, in the single-demand case we will study the problem of finding the maximum number of vertex-disjoint l-bounded paths between two vertices. We call this problem the **optimization problem**.

1.1 The Single-Demand Problems

In this section we present the problems which are related to a single demand. For each single-demand problem the input consists of a graph $G = (V, E)$ and a single demand d. (D contains only the demand d). For notational convenience we omit the demand index d.

The basic problem is the single-demand feasibility problem, where the aim is to decide, whether the given demand d can be satisfied by the graph G. We denote this problem by SFP.

The other two problems presented in this section are optimization problems related to SFP. The first one, the weighted single-demand problem WSP, is the problem of satisfying a given demand d with minimum total cost, with respect to some given edge costs c_e. The second optimization problem is to maximize the number of vertex-disjoint length-restricted paths for the given demand. We denote this problem by SOP.

For each single-demand problem a **solution** is a valid set of paths $Q \subseteq \mathcal{Q}$, no matter whether $|Q| \geq k$ or not. A solution with $|Q| \geq k$ is called a **feasible solution**.

1.1.1 The Single-Demand Feasibility Problem

In the single-demand feasibility problem SFP we have to find a feasible solution, i.e., a valid set of paths $Q \subseteq \mathcal{Q}$ with $|Q| \geq k$ (at least k vertex-disjoint l-bounded paths from s to t), or to proof that no such set exists.

This decision problem can easily be modelled as an integer program. With every solution $Q \subseteq \mathcal{Q}$ we associate its characteristic vector $\Upsilon_Q \in \{0,1\}^I \subseteq \mathbb{R}^I$, that is the 0/1-vector whose entries y_i indicate which paths $p_i \in \mathcal{Q}$ are in Q:

$$y_i = \begin{cases} 1 & \text{if } p_i \in Q, \\ 0 & \text{otherwise.} \end{cases}$$

Using this indicator variables, SFP can be formulated as the problem of finding a solution of the following integer linear system:

$$\begin{aligned} \sum_{i \in I : v \in p_i} y_i &\leq 1 && \forall v \in V \setminus \{s,t\} \\ \sum_{i \in I} y_i &\geq k && \\ y_i &\in \{0,1\} && \forall i \in I \end{aligned} \tag{SFP-IP}$$

This system has two types of inequalities. We call the first class of inequalities,

$$\sum_{i \in I : v \in p_i} y_i \leq 1 \qquad \forall v \in V \setminus \{s,t\}, \tag{1}$$

the vertex disjointness inequalities, or shorter **disjointness inequalities**. These inequalities ensure that every vertex, except s and t, lies on at most one path.

$$\sum_{i \in I} y_i \geq k \tag{2}$$

The **satisfaction inequality** (2) ensures that the demand will be satisfied by the selected paths.

In Section 2.2 we will show that the decision problem with length bounds $l \leq 4$ is solvable in polynomial time, but that it is $\mathcal{NP}$–complete for length bounds $l \geq 5$.

We will present some heuristics and an exact method for SFP in Chapter 4.

1.1.2 The Single-Demand Optimization Problem

As mentioned above, there are two optimization problems directly related to SFP. One of them is the problem of finding the maximum number of vertex-disjoint l-bounded (s,t)-paths. We call this the single-demand optimization problem SOP.

In this problem we are given a graph G and a demand d. Certainly, we can omit the diversification parameter k^d of the demand—we are maximizing its value.

Using the variables and disjointness inequalities (1) introduced for SFP, we can easily formulate the optimization problem as the following integer program:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in I} y_i & & \\ \text{s.t.} \quad & \sum_{i \in I : v \in p_i} y_i \leq 1 & \forall v \in V \setminus \{s,t\} \\ & y_i \in \{0,1\} & \forall i \in I \end{aligned} \qquad (\mathsf{SOP}\text{-IP})$$

In Section 2.4 we will show that this problem is $\mathcal{MAX}$-$\mathcal{SNP}$–hard for general length bounds and that it is polynomially solvable for length bounds $l \leq 4$.

In Chapter 4 we present some heuristics and an exact algorithm for SOP.

1.1.3 The Weighted Single-Demand Problem

Another optimization problem directly related to SFP is the following:

Given a graph G, a demand d, and nonnegative edge costs c_e associated with each edge $e \in E$, our objective is to select a subset of the edges such that the demand can be satisfied and this set of edges has minimal cost among all those subsets. In other words, we have to find a cost-minimal feasible solution. We will refer to this problem as the weighted single-demand problem WSP.

Again, we need a 0/1-variable y_i for each path $p_i \in \mathcal{Q}$. Additionally, we introduce 0/1-variables x_e for each edge $e \in E$. These variables form the characteristic vector χ_F of the set F of edges used by the selected paths to satisfy the demand:

$$x_e = \begin{cases} 1 & \text{if } y_i = 1 \text{ for a } p_i \in Q \text{ with } e \in p_i, \\ 0 & \text{otherwise.} \end{cases}$$

We call the inequalities

$$\sum_{i \in I: e \in p_i} y_i \leq x_e \qquad \forall e \in E \tag{3}$$

imposed by these conditions the **edge inequalities**. Using the inequalities (1),(2) derived from SFP and these new inequalities (3), we obtain the following integer linear program:

$$\begin{array}{lll} \text{minimize} & c^T x & \\ \text{s.t.} & \displaystyle\sum_{i \in I: v \in p_i} y_i \leq 1 & \forall v \in V \setminus \{s, t\} \\ & \displaystyle\sum_{i \in I: e \in p_i} y_i \leq x_e & \forall e \in E \\ & \displaystyle\sum_{i \in I} y_i \geq k & \\ & y_i \in \{0, 1\} & \forall i \in I \\ & x_e \in \{0, 1\} & \forall e \in E \end{array} \tag{WSP-IP}$$

Although this optimization problem does not look very hard, we will show some nice (in a slightly destructive sense) complexity results in Section 2.3. If $l \geq 5$ there is no polynomial time algorithm that approximates WSP within a factor of $\alpha(n)$ of the optimum, where $\alpha(n)$ is any function with $\alpha \in O(2^{p(n)})$ and $p(n)$ is any polynomial in n, unless $\mathcal{P} = \mathcal{NP}$. We can even show that WSP is $\mathcal{NPO}$–complete in general. In this section we also prove that a number of special cases of WSP is solvable in polynomial time.

Some heuristics and an exact algorithm for WSP are presented in Chapter 4.

1.2 The Multi-Demand Problems

In the multi-demand problems the input consists of a graph G and a set D of demands with $|D| \geq 1$. Recall that, by definition, each demand can have its own individual length bound and diversification parameter.

We want to satisfy the demands $d \in D$ independently with valid paths in $\mathcal{P}^d$. This means that for each demand $d \in D$ the paths p_i^d chosen to satisfy d have to be pairwise vertex-disjoint, but paths chosen for different demands may have common edges or nodes. Obviously, this concept is a generalization of the single demand case.

1.2.1 The Multi-Demand Feasibility Problem

In the multi-demand feasibility problem MFP we are given a graph G and a demand set D. The problem is to decide, whether all the demands can be satisfied by the given graph. Clearly, this problem is decomposable. We can solve this problem by independently solving SFP for each demand.

Hence it is not surprising that, analogously to Section 1.1.1, we can give an integer programming formulation for MFP. As we did for the model of SFP, we introduce the path indicator variables y_i^d for each demand $d \in D$ and each (s^d, t^d)-path p_i^d. Combining the disjointness inequalities (1) and the satisfaction inequalities (2) for all demands, we obtain the following system. This system has an integer solution if and only if there is a solution for MFP:

$$\begin{aligned} \sum_{i \in I^d : v \in p_i^d} y_i^d &\leq 1 && \forall d \in D, \quad \forall v \in V \setminus \{s^d, t^d\} \\ \sum_{i \in I^d} y_i^d &\geq k^d && \forall d \in D \\ y_i^d &\in \{0, 1\} && \forall d \in D, \quad \forall i \in I^d \end{aligned} \tag{MFP-IP}$$

As mentioned above, MFP is a generalization of SFP and therefore it is $\mathcal{NP}$–complete, too.

In Chapter 5 we will present some algorithmic approaches to MFP which are based on the various heuristics for the single-demand problems.

1.2.2 The Weighted Multi-Demand Problem

The weighted multi-demand problem WMP is the problem that motivated the work on the problems discussed here. It is a formalization of a special network design problem, where individual survivability conditions are imposed for several communication demands.

The input consists of a graph G, a demand set D, and nonnegative edge costs c_e for each edge $e \in E$. The objective is to select a subset of the edges at minimum cost, such that each demand $d \in D$ is satisfied by the subgraph induced by these edges.

Unlike MFP, this problem cannot be decomposed into the single demand problems.

Again, to formulate this problem as an integer linear program, we use 0/1-variables y_i^d for all demands $d \in D$ and for all paths $p_i^d \in \mathcal{P}^d$. Analogously to the model of WSP, we need 0/1-variables x_e for each edge $e \in E$, indicating whether edge e is chosen to satisfy a demand. Combining inequalities (1),(2) and (3) for all demands, we get the complete model:

$$
\begin{array}{lrll}
\text{minimize} & c^T x & & \\
\text{st.} & \displaystyle\sum_{i \in I^d : v \in p_i^d} y_i^d \leq 1 & \forall d \in D, & \forall v \in V \setminus \{s^d, t^d\} \\
& \displaystyle\sum_{i \in I^d : e \in p_i^d} y_i^d \leq x_e & \forall d \in D, & \forall e \in E \\
& \displaystyle\sum_{i \in I^d} y_i^d \geq k^d & \forall d \in D & \\
& y_i^d \in \{0,1\} & \forall d \in D, & \forall i \in I^d \\
& x_e \in \{0,1\} & \forall e \in E &
\end{array}
\tag{WMP-IP}
$$

Because WMP is a generalization of WSP our "non-approximation" results in Section 2.3 also hold for WMP and, of course, WMP is $\mathcal{NPO}$–complete, too.

Chapter 2

The Complexity of the Problems

Now that we have presented the problems, we want to analyse their complexity. It is important to know how hard a problem or a certain special case is, if one wants to design and implement algorithms to handle this problem.

In the first section we will give an overview of the complexity classes we are dealing with. This includes definitions, basic facts, and some relations between the complexity classes.

Thereafter we will present some of our main results. For all problems presented in the previous chapter we will show some completeness results.

2.1 Computational Complexity

It is not in the scope of this thesis to give a complete introduction to the theory of computational complexity. We only present basic definitions and facts as far as they are needed. No proofs are given. For a detailed introduction to complexity theory see [PY88],[Pap94], [BC94], or [ST95].

We start with a very informal introduction to the main complexity classes and their relations. Thereafter, we will shortly present the basic tools related to the approximation of problems.

2.1.1 Exact Computation

A **problem** (in the sense of computation) is specified as a relation between input and output. An algorithm that takes an input of a certain format, called an **instance**, solves the problem by computing an output, called **solution**, that satisfies the input–output relation.

Formally, a computational problem Π can be defined by a triple (I, X, Γ), where I is the set of all instances of Π, X is the set of all possible solutions for Π, and Γ is a predicate deciding whether a possible solution is feasible for the given instance. For $\phi \in I$ we denote by $X(\phi)$ the set of all feasible solutions for ϕ.

We call a problem a **decision problem** if the only two solutions are "yes" and "no".

An **optimization problem** is a problem, where we have an additional measure for the "quality" of the solution and a goal (maximization or minimization). Formally, it is a 5-tuple $(I, X, \Gamma, c, goal)$, where (I, X, Γ) is a problem, c is an additional function that associates with any instance $\phi \in I$ and with any feasible solution $x \in X(\phi)$ a value $c(\phi, x) \in \mathbb{N}$, and the *goal* is to maximize or to minimize c for the given instance.

It is easy to see that an optimization problem having only two possible objective values is a decision problem. Hence, to every optimization problem Π we can assign a decision problem Π' by asking "Is there a solution of value $\geq k$", if the goal was to maximize c, or analogous "Is there a solution of value $\leq k$", if the goal was to minimize c. We will call such an assigned decision problem Π' an underlying decision problem of Π.

We define the **complexity** of a problem as the order of computational resources that are needed to solve a problem. We distinguish between time- and space-complexity of a problem. Note that the complexity of a problem does *not* depend on the particular algorithm used to solve the problem.

We introduce **complexity classes** as sets of problems, whose complexity (of a particular computational resource) is limited by a certain bound, e.g., problems that have a time bound that is polynomially bounded in the input size of the problem.

Usually it is necessary to introduce a mathematical model of computation and data encoding when dealing with computational complexity. We will introduce this model rather briefly and informally. We will restrict our attention to the RAM model (random access memory machine). It is know, that this model has exactly the same computational power like the (older) Turing machine model.

Given an instance $\phi \in \Pi$ we denote the **coding length** of ϕ by $\langle\phi\rangle$. In standard binary representation this is the number of bits needed to encode ϕ. This coding length is a reasonable measure for the size of an instance of a problem.

Let Π be a computational problem and A be an algorithm which solves Π. We say that A runs in polynomial time or that A is a **polynomial time algorithm** if it can be implemented on a RAM machine such that the number of bit operations performed to solve an instance ϕ of Π is bounded by a polynomial in $\langle\phi\rangle$. We say that A runs in **log-space** if the number of bits additionally needed for the computation is bounded logarithmically in $\langle\phi\rangle$. A problem Π is said to be solvable in polynomial time (or polynomially) or in log-space if there is an Algorithm A that solves Π in polynomial time or in log-space, respectively.

One of the central complexity measures of a problem is the time-complexity. In the following we will define some time-complexity classes used in the theory of computational complexity. For decision problems we have the following two complexity classes.

Definition 2.1 ($\mathcal{NP}$) *$\mathcal{NP}$ is the class of all decision problems, for which every positive solution ("yes") can be verified in polynomial time.*

Definition 2.2 ($\mathcal{P}$) *$\mathcal{P}$ is the class of all decision problems solvable in polynomial time.*

For optimization problems we define two complexity classes that are based on the complexity of the underlying decision problems.

Definition 2.3 ($\mathcal{NPO}$) *An optimization problem $(I, X, \Gamma, c, goal)$ is in $\mathcal{NPO}$, if*

(i) *the set of instances I is recognizable in polynomial time,*

(ii) *for any instance $\phi \in I$ the length of all feasible solutions $s \in X(\phi)$ is bounded by a polynomial p in $\langle\phi\rangle$ and for any $\phi \in I$ and any $s \in X$ with $\langle s\rangle \leq p(\langle\phi\rangle)$ it is decidable[1] in polynomial time whether $s \in X(\phi)$ (Γ is decidable in polynomial time), and*

[1] Decidable in polynomial time means here that the decision problem, whether x and ϕ satisfy the input output relation Γ, can be solved in polynomial time.

(iii) *for any $\phi \in I$ and any $x \in X(\phi)$ the measure $c(\phi, x)$ is computable in polynomial time.*

Definition 2.4 ($\mathcal{PO}$) *An optimization problem in $\mathcal{NPO}$ belongs to the class $\mathcal{PO}$, if it is solvable in polynomial time.*

It is clear by the definitions, that $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{PO} \subseteq \mathcal{NPO}$.

Next we need the concept of **reducibility**. We say problem Π can be reduced to problem Π', if there are two transformations T and S such that T maps every instance ϕ of Π to an instance ψ of Π' and S maps every feasible solution x' of ψ to a feasible solution x of ϕ. To be reasonable, the complexity of such a reduction has to be somehow bounded. Usually, one restricts to polynomial-time or log-space reductions, that is both transformations T and S are polynomially-time bounded or log-space bounded, respectively. In the following we use the term reduction for polynomial-time reductions, if not stated otherwise.

One may think of a reduction from problem Π to problem Π' as an algorithmic framework for Π. First, we "run" transformation T on the given instance of Π, then we run an algorithm for Π' on the transformed instance, and finally we reconstruct a solution for the original instance by applying transformation S on the solution of the transformed instance.

It is clear that the composition of two reductions is a reduction itself. Furthermore, if both reductions are polynomial-time or log-space bounded, then so is the composition.

Using the concept of reducibility, we can study the relative "hardness" of a problem. It is easy to see, that the complexity classes defined above are closed under polynomial-time reductions and under log-space reductions. Furthermore, we are able to characterize hardest problems in a complexity class.

Definition 2.5 (Hardness and completeness) *let $\mathcal{C}$ be a complexity class. Problem Π is called $\mathcal{C}$–hard, if all problems in $\mathcal{C}$ reduce to it (by an appropriate[2] reduction). If Π itself is in $\mathcal{C}$, then Π is called $\mathcal{C}$–complete.*

Note that a problem which is known to be $\mathcal{C}$–hard may have a much higher complexity than any problem in $\mathcal{C}$.

[2] In this thesis we only use polynomially-time bounded reductions. Using other reductions, one can define various complexity classes as classes of problems that are closed under these reductions. See L–reductions in Section 2.1.2 for example.

Although it is not clear by the definition, it is known, that there indeed exist $\mathcal{P}$–complete and $\mathcal{NP}$–complete problems as well as $\mathcal{PO}$– and $\mathcal{NPO}$–complete problems. For example, 3-Sat is known to be $\mathcal{NP}$–complete. One of the central problems in the theory of computational complexity is $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, which is still unknown.

2.1.2 Approximation

There are a lot of optimization problems we cannot hope to solve to optimality in practice. In general, there are two ways to handle optimization problems not in $\mathcal{PO}$. The first one is to solve these problems exactly in non-polynomial time. This approach is not practicable for large scale instances due to the immense need of time. Often it is more useful to compute approximate solutions of these problems in polynomial time. Unfortunately, it is possible that even such an approximation (within reasonable bounds) is not possible in polynomial time. In this section we will present basic tools to analyse the approximation complexity of problems.

Definition 2.6 (Approximation) *Let Π be an optimization problem and A be an algorithm that returns a feasible solution of value $A(\phi)$ for any instance ϕ of Π. We denote the optimal solution value of ϕ with $Opt(\phi)$. Given a constant $\epsilon \geq 0$, we call A an **ϵ-approximation algorithm** for P, if*

$$\frac{|A(\phi) - Opt(\phi)|}{Opt(\phi)} \leq \epsilon \quad \text{for all } \phi \in \Pi.$$

We call ϵ the approximation guarantee of A.

Now we can refine the complexity classes we have defined for optimization problems in the previous section.

Definition 2.7 ($\mathcal{APX}$) *An $\mathcal{NPO}$ problem Π belongs to the class $\mathcal{APX}$, if there is a (polynomial-time) ϵ-approximation algorithm for Π for some $\epsilon > 0$.*

Often one needs to approximate an optimization problem Π to a given bound $\epsilon \geq 0$.

Definition 2.8 (Approximation scheme) *Let Π be an optimization problem and A be an algorithm accepting an instance $\phi \in \Pi$ and a value $\epsilon > 0$*

as input and computing a solution x of ϕ with value $A_\epsilon(\phi)$. We call A an ***approximation scheme*** *for Π if*

$$\frac{|A_\epsilon(\phi) - Opt(\phi)|}{Opt(\phi)} \leq \epsilon \quad \textit{for all } \phi \in \Pi, \epsilon > 0.$$

Definition 2.9 ($\mathcal{PTAS}$) *An $\mathcal{NPO}$–problem Π is in the class $\mathcal{PTAS}$ if it admits a polynomial-time approximation scheme. A polynomial-time approximation scheme is an approximation scheme, whose time-complexity is polynomially bounded in $\langle\phi\rangle$ (for all $\epsilon > 0$), where ϕ is the given instance of Π.*

Definition 2.10 ($\mathcal{FPTAS}$) *An $\mathcal{NPO}$–problem Π is in the class $\mathcal{FPTAS}$, if it admits a fully polynomial-time approximation scheme, that is an approximation scheme, whose time-complexity is polynomially bounded in $\langle\phi\rangle$ and $\langle\epsilon\rangle$.*

It is easy to see that $\mathcal{FPTAS} \subseteq \mathcal{PTAS} \subseteq \mathcal{APX} \subseteq \mathcal{NPO}$.

The concept of problem reduction presented in the last section is useless for the study of approximation complexity. We get no information about the quality of a solution after applying a reduction from the transformations. We have to restrict our attention to a subclass of reductions to get reasonable results. The following definition of an L–reduction is due to Papadimitriou and Yannakakis [PY88].

Definition 2.11 (L–reduction) *Let Π and Π' be two optimization problems. For solutions x and x' of Π and Π', respectively, let $c(x)$ and $c(x')$ denote the costs of the solutions. An L–reduction from Π to Π' is a pair of log-space transformations T (for the instances) and S (for the solutions) with the following properties:*

(i) *There is a constant $\alpha > 0$, such that for all instances ϕ of Π $T(\phi)$ is an instance of Π' with*

$$Opt(T(\phi)) \leq \alpha \cdot Opt(\phi).$$

(ii) *There is a constant $\beta > 0$, such that for every feasible solution x' of $T(\phi)$ $S(x')$ is a feasible solution of ϕ with*

$$|Opt(\phi) - c(S(x'))| \leq \beta \cdot |Opt(T(\phi)) - c(x')|.$$

There are other approximation preserving reductions, like $\mathcal{APX}$–reductions or $\mathcal{PTAS}$–reductions (see [BC94]). The difference is that these reductions are polynomial-time bounded instead of log-space bounded and that they transform the gap to the optimal solution value nonlinearly. We focus on L–reductions, because of the linear transformation of the optimal solution values and the gap to the optimal solution value.

Of course, the composition of two L–reductions is an L–reduction.

The following theorem, which is a direct consequence of the definition, shows how the approximation guarantee behaves under these L–reductions.

Theorem 2.12 *Let* Π *and* Π' *be two optimization problems and let* (T, S) *be an L–reduction from* Π *to* Π' *with constants* α *and* β*, as defined above. If there is a polynomial-time* ϵ*-approximation algorithm* A' *for* Π'*, then there is a polynomial-time* $\left(\frac{\alpha\beta\epsilon}{1-\epsilon}\right)$*-approximation algorithm* A *for* Π*.*

There is another complexity class related to approximation—the $\mathcal{MAX}$-$\mathcal{SNP}$ class introduced by Papadimitriou and Yannakakis [PY88]. This class is defined syntactically. We do not want to present the correct definition here, because of the large notational overhead we would need for the definition. Papadimitriou and Yannakakis showed that this class is closed under L–reductions (actually, it is defined to be the closure of another class under L–reductions). They showed that there is a polynomial time ϵ-approximation algorithm for every problem in $\mathcal{MAX}$-$\mathcal{SNP}$ and that no $\mathcal{MAX}$-$\mathcal{SNP}$–complete problem can have a polynomial time approximation scheme, unless $\mathcal{P} = \mathcal{NP}$. Furthermore, Papadimitriou and Yannakakis have shown that MAX-3-SAT is $\mathcal{MAX}$-$\mathcal{SNP}$–complete. Since we will need this result in Section 2.4, we will state it as a theorem here. The problem MAX-3-SAT is defined as follows: Given a set U of (boolean) variables and a collection C of disjunctive clauses over U of at most 3 literals (a literal is a variable or a negated variable), we want to find a truth assignment for the variables in U that maximizes the number of clauses satisfied by this assignment.

Theorem 2.13 (MAX-3-SAT)

(i) MAX-3-SAT *is* $\mathcal{MAX}$-$\mathcal{SNP}$*– and* $\mathcal{APX}$*–complete.*

(ii) *There is a* $\rho > 0$ *such that there is no polynomial-time* ϵ*-approximation algorithm for* MAX-3-SAT *with* $\epsilon < \rho$*, unless* $\mathcal{P} = \mathcal{NP}$*. Currently, the best know value for* ρ *is* $1/37$*.*

After this short introduction to the theory of computational complexity we will present a number of complexity results for the demand problems introduced in Chapter 1.

2.2 The Complexity of SFP

The following result is due to Itai, Perl, and Shiloach, who presented various complexity results for classes of edge and vertex-disjoint path problems in [IPS82]. We modified their proof of Theorem 2.14 to make it easier to understand.

Theorem 2.14 SFP *is $\mathcal{NP}$–complete for length bounds $l \geq 5$.*

Proof: It is easy to see that SFP is in $\mathcal{NP}$. To show that SFP is $\mathcal{NP}$–hard, we will show that 3-SAT is polynomially reducible to SFP. Since it is known that 3-SAT is $\mathcal{NP}$-hard, the $\mathcal{NP}$-hardness of SFP follows.

The decision problem 3-SAT is defined as follows: Given a set U of (boolean) variables and a collection C of disjunctive clauses over U of at most 3 literals, the aim is to find a truth assignment for the variables in U, such that each clause is satisfied by this assignment. This problem is $\mathcal{NP}$–complete [Pap94].

Given an instance ϕ of 3-SAT we construct an instance ψ of SFP by clipping together (sub-)graphs that correspond to the variables or to the clauses of ϕ.

Let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_r$ be an instance of 3-SAT with variables $x_1, \ldots, x_q$. Let m_i denote the number of occurrences of variable x_i (negated or unnegated). The graph G_ϕ we construct will contain $r + \sum_{i=1}^{q} m_i$ vertex-disjoint (s,t)-paths if and only if there is a truth assignment for the variables in U that satisfies all clauses in ϕ. All these paths will have length $l = 5$.

The graph $G = G_\phi$ is composed of $q + r$ subgraphs, r clause subgraphs H_k $k = 1, \ldots, r$ and q variable subgraphs G_i $i = 1, \ldots, q$.

The variable subgraph G_i, defined for each variable x_i, ensures that all occurrences of variable x_i in ϕ are set uniformly. The subgraph G_i is defined as shown in Figure 2.1. For each occurrence j of x_i in ϕ there is a separate path $p_i^j = (t, z_i^j, y_i^j, x_i^j, w_i^j, s)$. These paths have only s and t in common. Furthermore, there are cross-links between the single paths. For all pairs of paths p_i^j and p_i^{j+1} with $1 \leq j < m_i$ we add an additional vertex $\bar{x}_i^j$ and two additional edges $(\bar{x}_i^j, w_i^{j+1})$ and $(y_i^j, \bar{x}_i^j)$. Analogously, there is a cross-link

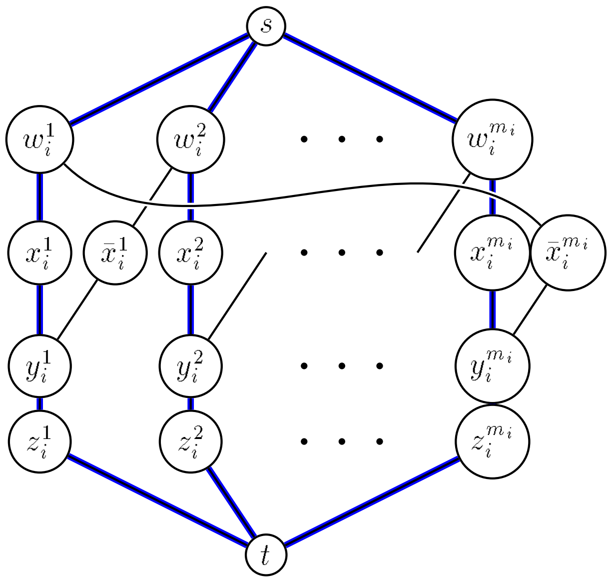


Figure 2.1: The variable subgraph G_i

between the last path $p_i^{m_i}$ and the first path p_i^1 with vertex $\bar{x}_i^{m_i}$. This forms m_i new (s,t)-paths $\bar{p}_i^j = (t, z_i^j, y_i^j, \bar{x}_i^j, w_i^{j+1}, s)$.

Observation 2.15

(i) *The paths p_i^j and $\bar{p}_i^j$ are the only (s,t)-paths in G_i with length $l \leq 5$.*

(ii) *Any collection of m_i vertex-disjoint (s,t)-paths in G_i contains either all p_i^j paths or all $\bar{p}_i^j$ paths.*

We will interpret the selection of path p_j^i as the "*false*"–setting of the j-th occurrence of variable x_i and the selection of path $\bar{p}_i^j$ as "true"–setting (exactly the other way around the reader expected). Thus, we can ensure the uniformity of all occurrences of the variable x_i by requiring exactly m_i vertex-disjoint paths of length $l \leq 5$ in G_i. All these variable subgraphs G_i are clipped together in s and t, the two common vertices of all G_i subgraphs.

Now let us look at the clause subgraphs in G_ϕ. For each clause C_k we have a clause subgraph H_k as defined in Figure 2.2. H_k contains the vertex c_k for the clause and vertices a_i^j, b_i^j, if variable x_i occurs in clause C_k for the j-th time in ϕ. Furthermore, the subgraph H_k contains either vertex x_i^j or vertex $\bar{x}_i^j$ (that are already defined in the variable subgraph G_i). H_k contains vertex

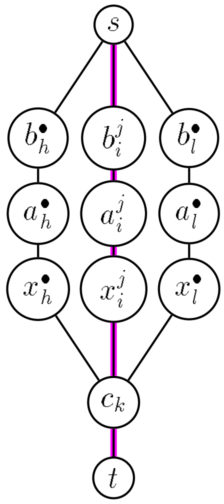


Figure 2.2: The clause subgraph H_k

x_i^j if the variable x_i occurs in clause C_k unnegated and it contains vertex $\bar{x}_i^j$ if variable x_i occurs negated. Last but not least, H_k contains the vertices s and t, where everything is clipped. The vertices in H_k are linked by edges as shown in the Figure 2.2. H_k contains the edge (t, c_k) and for every variable x_i it contains either the edges (c_k, x_i^j), (x_i^j, a_i^j), (a_i^j, b_i^j), and (b_i^j, s) or the edges $(c_k, \bar{x}_i^j)$, $(\bar{x}_i^j, a_i^j)$,(a_i^j, b_i^j), and (b_i^j, s) if the variable x_i occurs in ϕ for the j-th time unnegated or negated, respectively. This forms the (s,t)-path h_i^k or $\bar{h}_i^k$, respectively.

Observation 2.16 *There is only one "vertex-disjoint" (s,t)-path of length 5 in H_k. There is an (s,t)-path containing vertex x_i^j (or $\bar{x}_i^j$) in H_k, if and only if variable x_i occurs unnegated (or negated, respectively) in clause C_k the j-th time in ϕ.*

We can interpret a path h_i^k as satisfying clause C_k by setting the variable x_i to true and a path $\bar{h}_i^k$ as satisfying clause C_k by setting the (negated) variable x_i to false. (This time it should conform with the readers expectation!) Note that this interpretation agrees with the interpretation of the paths p_i^j and $\bar{p}_i^j$ in G_i.

We define the demand of ψ as $d := (s, t, l = 5, k = \sum_{i=1}^{q} m_i + r)$.

Obviously this reduction is polynomially-time (and log-space) bounded in $\langle \phi \rangle$.

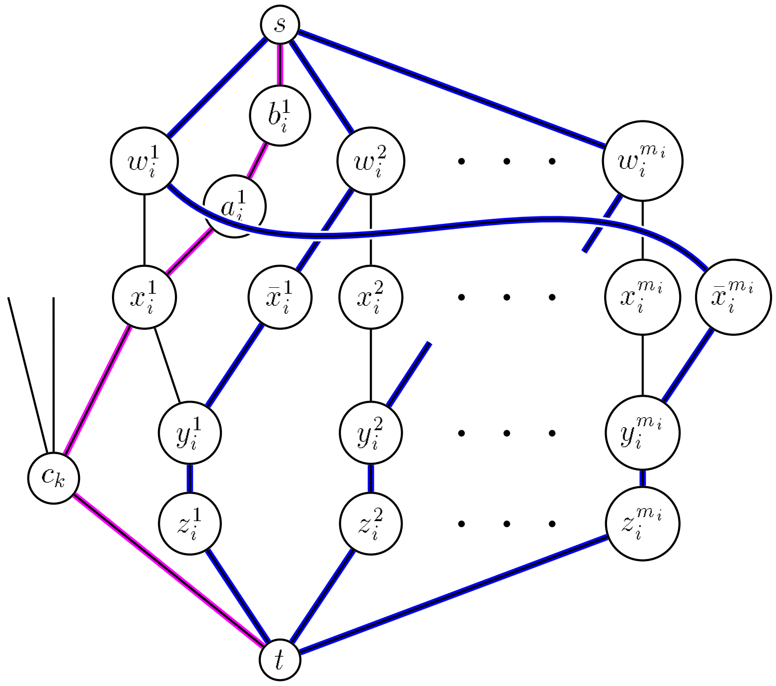


Figure 2.3: The clipping of the x_i and C_k subgraphs

The rest of the proof follows directly from the following lemma.

Lemma 2.17 *There is a set of* $\sum_{i=1}^{q} m_i + r$ *vertex-disjoint* (s,t)*-paths in* $G = G_\phi$ *with length* $l \leq 5$*, if and only if there is a truth assignment for the variables in* ϕ *such that all clauses are satisfied.*

Proof: "$\Leftarrow$": Suppose that ϕ is satisfiable and let x be a feasible truth assignment. We construct a set Q of $\sum_{i=1}^{q} m_i + r$ vertex-disjoint (s,t)-paths of length 5 in G. In every variable subgraph G_i we choose all p_i-paths to be in Q if the variable x_i is false, and all $\bar{p}_i$-paths if x_i is true.

Since x is a satisfying truth assignment for ϕ, we know that for each clause C_k there is at least one literal x_i (or $\bar{x}_i$, if negated) which evaluates to true. Thus, we can add the path h_i^k (or $\bar{h}_i^k$, respectively) to Q.

Obviously, h_i^k is vertex-disjoint to all paths already in Q, because $h_i^k \in Q$ if and only if $\bar{p}_i \in Q$, and vice versa. Consequently, we can add one additional path to Q for each clause such that the paths in Q remain vertex-disjoint. Altogether, we end up with $\sum_{i=1}^{q} m_i + r$ paths in Q, all of them of length 5.

"$\Rightarrow$": Suppose we have a set Q of $\sum_{i=1}^{q} m_i + r$ vertex-disjoint (s,t)-paths of length 5 or less. By the construction of G, we know that $|\delta(t)| = \sum_{i=1}^{q} m_i + r$. Since all edges in $\delta(t)$ belong to either a variable subgraph G_i or a clause subgraph H_k, there must be exactly m_i paths in every G_i. (They cannot "leave" G_i within their length bound of 5.) By Observation 2.15, it follows that for each variable subgraph G_i either all p_i-paths or all $\bar{p}_i$-paths are in Q. Since all these paths use either the vertices x_i^j $j = 1, \ldots, m_i$ or the vertices $\bar{x}_i^j$ $j = 1, \ldots, m_i$ uniformly, we can set the variable x_i according to the vertices used by these paths—true if we have $\bar{p}_i$-paths, false otherwise. It is easy to see that the paths through the vertices c_r correspond to literals that are true, because the h_k-paths (resp. $\bar{h}_k$) are vertex-disjoint to the $\bar{p}_i$-paths (resp. p_i). Thus, each clause in ϕ is true with this setting of the x_i variables. □

Remark 2.18 SFP *is polynomially solvable for length bounds* $l \leq 4$.

Remark 2.18 is a simple corollary of Theorem 2.35 which will be shown in Section 2.4.

2.3 The Complexity of WSP

The first theorem stated in this section is more a corollary to Theorem 2.14 than a theorem itself.

Theorem 2.19 WSP *is in* $\mathcal{NPO}$.

Proof: By Theorem 2.14, SFP is in $\mathcal{NP}$. Thus, all instances are recognizable in polynomial time, all solutions are "short" (their length is bounded by a polynomial in the input size) and we can decide in polynomial time whether such short "verifiers" are indeed feasible solutions. Obviously, the total edge costs can be calculated in polynomial time. Hence, WSP belongs to $\mathcal{NPO}$. □

Now, that we have seen that WSP is in $\mathcal{NPO}$, we will show two of our main results—WSP is hard. First we will show that no reasonable approximation for WSP is possible, unless $\mathcal{P} = \mathcal{NP}$, and then we will show that WSP is among the hardest optimization problems in $\mathcal{NPO}$—it is $\mathcal{NPO}$-complete.

Theorem 2.20 *Let n denote the number of vertices in the graph of the given instance of* WSP. *Then there is no polynomial time algorithm that solves* WSP

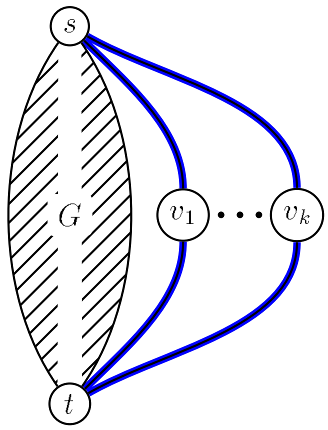


Figure 2.4: The G' graph

within a factor of $O(2^{p(n)})$ *of the optimum value for some polynomial* p*, unless* $\mathcal{P} = \mathcal{NP}$*, This also holds if the length bound* l *is fixed to 5.*

Proof: We will show that we can use any algorithm that approximates WSP within a "computable" factor to solve SFP. The idea is that we always guarantee the existence of paths as required by the demand by adding extra paths with high costs.

Let $\phi = (G, d)$ be an instance of SFP. Suppose we have a polynomial time algorithm A that always approximates WSP within a factor of $C \cdot 2^{p(n)}$, where n is the number of vertices in G, $C > 0$ is a constant, and p is a polynomial in n.

We construct an instance $\psi = (G', d, c)$ of WSP as follows. Let $G' = (V', E')$ be the graph as defined in Figure 2.4. If k is the diversification parameter of d, we just add k new vertices $v_1, \dots, v_k$ and link them with new edges to s and t.

The edge costs are defined as follows. For each edge (s, v_i) and (v_i, t) we set the cost $c_{(s,v_i)} = c_{(v_i,t)} = C \cdot 2^{p(n)}m$, where m is the number of edges in G. For all other edges $e \in E'$ set $c_e = 1$.

Note, that this transformation is polynomial. The number of vertices of G' is $n + k \leq 2n$ and the number of edges of G' is $m + 2k \leq m + 2n$, and thus $\langle G' \rangle \in O(n+m)$. Although the values of the edge costs are not polynomially bounded in the encoding length of ϕ, their encoding length is. For the largest edge cost $c_{max} = C \cdot 2^{p(n)}m$ we have $\langle c_{max} \rangle \leq \langle C \rangle + \langle 2^{p(n)} \rangle + \langle m \rangle$. Since C

is a constant and $m \in O(n^2)$, we have $\langle c_{max} \rangle \in O(p(n))$ for a polynomial p. Consequently, $\langle \psi \rangle$ is polynomially bounded in $\langle \phi \rangle$.

Note, the subgraph in G' induced by the "cheap" edges of cost 1 is exactly G. It is clear, that there is a positive solution for ϕ iff there is a solution for ψ with overall edge cost of at most m, because in this case none of the auxiliary edges is used.

On the other hand, if there is no positive solution for ϕ, then all solutions for ψ will need at least one of the "expensive" edges (s, v_i), and hence the overall edge costs of any solution to ψ will be at least $C \cdot 2^{p(n)} m$.

Now it is easy to see that algorithm A, which is supposed to give a solution of WSP within a factor of $C \cdot 2^{p(n)}$, could be used to solve SFP. If we run A on ψ and the total costs returned by A are less or equal to m, then we know that the solution returned by A consists only of edges in the subgraph G. Hence, we have a solution for ϕ. If the costs returned by A are bigger than m, there has to be at least one expensive edge in the solution created. Ergo, the costs of the solution returned by A have to be strictly larger than $C \cdot 2^{p(n)} m$. Then, by the approximation guarantee of A, there is no solution for ψ of cost less or equal than m and, hence, there is no positive solution for ϕ. $\square$

Corollary 2.21 WSP *is neither in* $\mathcal{APX}$ *nor in* $\mathcal{MAX}$-$\mathcal{SNP}$.

This corollary follows trivially from Theorem 2.20.

It is not only that WSP is hard to approximate, we can even show that it is among the hardest problems in $\mathcal{NPO}$.

Theorem 2.22 WSP *is* $\mathcal{NPO}$*-complete.*

Proof: To prove this result, we will modify the transformation given in Theorem 2.14 to prove the $\mathcal{NP}$-completeness of SFP. We give a reduction from the Minimum Weighted Satisfiability problem to WSP.

The Minimum Weighted Satisfiability problem is defined as follows: Given a set U of (boolean) variables, a collection C of disjunctive clauses and a nonnegative integer weight w_x for each variable $x \in U$, we are asked to find a truth assignment for U that satisfies each clause in C and minimizes the sum of the weights of the true variables. This problem is $\mathcal{NPO}$-complete [OM87].

Let ϕ be an instance of MINIMUM WEIGHTED SATISFIABILITY with variables $x_1, \ldots, x_q$, clauses $C_1, \ldots, C_r$, and nonnegative weights w_i, $1 \leq i \leq q$, associated with the variables.

We construct an instance $\psi = (G, d, c)$ of WSP nearly the same way we did in the proof of Theorem 2.14. When constructing the graph G, the only difference is that the clause subgraphs H_k, $1 \leq k \leq r$, may have more than 3 "wings", since the clauses in ϕ may contain more than 3 literals. The rest of the construction and the setting of the demand d does not change.

The costs c are defined as follows. For each variable subgraph G_i, $1 \leq i \leq q$, and for each edge $e = (y_i^1, \bar{x}_i^1)$ we set the edge costs $c_e := w_i$, that is the weight of the corresponding boolean variable in ϕ. For all other edges e we set the costs to $c_e := 0$.

The rest follows directly from the proof of Theorem 2.14. The feasible solutions of ϕ correspond to the feasible solutions of ψ. Furthermore, since a feasible solution Q of ψ contains the edge $(y_i^1, \bar{x}_i^1)$ in an (s,t)-path if and only if the corresponding boolean variable x_i in ϕ is true, the total costs of the corresponding solutions are the same. Hence, by solving ψ we can solve ϕ. Obviously, the reduction given above is polynomial. (In fact, it is an L–reduction with $\alpha = 1$ and $\beta = 1$.) Consequently, WSP is $\mathcal{NPO}$-complete, too. $\square$

Although the general weighted single-demand problem is $\mathcal{NPO}$–complete, there are some special cases that can be solved reasonably.

The first polynomially solvable special case, where the diversification parameter is fixed to $k = 1$, is a special case of the RESTRICTED SHORTEST PATH problem, where all edge lengths are equal. This special case can be solved in polynomial time by a modification of Dijkstra's shortest path algorithm. We will present this modified algorithm in Section 4.2.

Proposition 2.23 WSP *is polynomially solvable if the diversification is fixed to* $k = 1$.

The second polynomially solvable special case we want to present is the **"complete metric"** WSP . An instance of WSP is called complete metric if the given graph G is complete ($G = K_n$) and the edge costs satisfy the triangle inequality. In contrast to other graph problems, for example the travelling salesman problem, WSP becomes trivial when made complete metric. All (s,t)-paths in an optimal solution will have at most 2 edges, otherwise they can be shortened. Thus the problem reduces to the min-cost

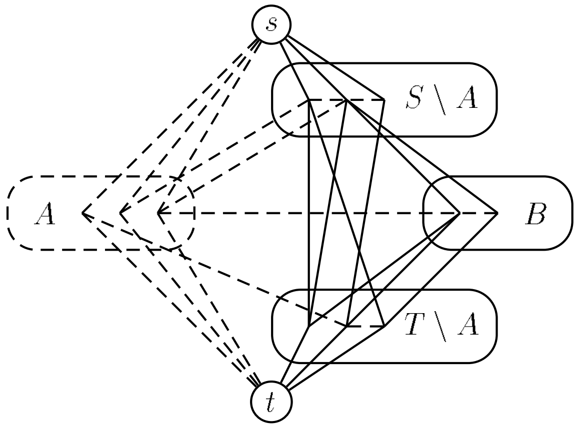


Figure 2.5: The graph G' for $l = 4$

flow problem where each path has length at most 2, which is solvable by a greedy algorithm.

Proposition 2.24 *The complete metric* WSP *is polynomially solvable.*

Finally, let us present another polynomially solvable case. Similar to the complete metric WSP we again have a cost function that satisfies the triangle inequality, but this time the graph does not need to be complete. We will call this case the **metric** WSP ..

Theorem 2.25 *The metric* WSP *is polynomially solvable for path length restrictions $l \leq 4$.*

Proof: Let $\phi = (G, d, c)$ be a metric instance of SOP with $l \leq 4$. We show that finding a feasible solution Q of vertex-disjoint l-bounded (s,t)-paths which minimizes the overall costs is solvable in polynomial time.

Let S be the set of all neighbors of s, T be the set of all neighbors of t, and $A = S \cap T$.

For $l = 1$ and $l = 2$ it is trivial that WSP is solvable in polynomial time. We just have to select the direct edge, if it exists, and if $l = 2$ we have to select all the paths that contain s,t and exactly one of the nodes in A, too. If we order these paths by increasing costs and choose the first k of them we obtain an optimal solution.

For $l = 3$ and $l = 4$ we can reduce the problem to a min cost flow problem. We decompose the problem such that we solve the problem first for the paths of length of at most 2 and then for the longer paths. Suppose we have an optimal solution Q. Then there is an optimal solution Q' such that every vertex $v \in A$ that appears in a path in Q is contained in a path p_v of length 2 in Q' or it is not contained in Q' at all. We obtain Q' from Q by shortening the paths that contain a vertex $v \in A$ using the triangle inequality. Clearly, (s, v, t) is the shortest path with respect to c that contains v. With the same argument it follows that S and T can be assumed to be stable sets—if some path in an optimal solution contains an edge $[S, S]$ or an edge in $[T, T]$ it can be shortened.

Now we can compute all vertex-disjoint paths of length $l = 1$ and $l = 2$ in G in increasing length order with respect to c. Let G' be the graph obtained from G by removing the vertices in A and the edges in $[S, S]$, in $[T, T]$, and the direct edge (s, t) (if it exists). Let B be the set of all vertices in $G' - (S \cup T)$ that have a neighbor in S and a neighbor in T (see Figure 2.5). Since all paths must be length bounded by at most 4, we can restrict to the case where B is a stable set—no path containing an edge in $[B, B]$ can be shorter than 5. It is easy to see that for $l = 4$ no vertex in $G' - (S \cup T \cup B)$ is contained in a solution, so we can restrict to graphs where $G' - (S \cup T \cup B)$ is empty. For $l = 3$ we even can restrict to graphs where $G' - (S \cup T)$ is empty. Now we can apply a min cost flow algorithm to compute for $i = 1, \ldots, k$ a min cost set Q_i of i vertex-disjoint (s, t)-paths in G'. Note, that the paths in Q_i satisfy the length restriction, otherwise they could be shortened. For each such $i = 1, \ldots, k$ we obtain a solution by choosing the paths in Q_i and the shortest $k - i$ paths computed for lengths 1 and 2, if they exist. Obviously, these paths are vertex-disjoint and the solutions created this way contain an optimal solution, if there is any feasible solution. It is easy to see, that this method is polynomial. $\square$

It is easy to see, that the instance $\psi = (G', d, c)$ of WSP we constructed in the proof of Theorem 2.20 is metric. Consequently, for length bounds $l \geq 5$ even the metric WSP is as hard as the nonmetric WSP, i.e., it cannot be approximated within any reasonable bound.

Corollary 2.26 *Let n denote the number of vertices in the graph of the given instance of the metric* WSP*. Then there is no polynomial time algorithm that solves the metric* WSP *within a factor of $O(2^{p(n)})$ of the optimum value for any polynomial $p(n)$ in n, unless $\mathcal{P} = \mathcal{NP}$. This also holds, if the length bound l is fixed to 5.*

2.4 The Complexity of SOP

After we have seen that SFP and WSP are hard, we might well expect that SOP is not much easier. In this section we will prove that this expectation is right. Although we cannot show results as strong as for WSP, we will show that SOP is "hard enough"—SOP is $\mathcal{MAX}$-$\mathcal{SNP}$– and $\mathcal{APX}$–hard and there is no polynomial time ϵ-approximation algorithm for SOP with $\epsilon < \frac{7}{37}$, unless $\mathcal{P} = \mathcal{NP}$.

Before proving this fact, first let us show that:

Theorem 2.27 SOP *is in* $\mathcal{NPO}$.

Proof: Again, this follows trivially from Theorem 2.14 which states that the underlying decision problem SFP is in $\mathcal{NP}$. Clearly the objective function value can be computed in polynomial time. □

To prove that SOP is $\mathcal{MAX}$-$\mathcal{SNP}$-hard, we give an L–reduction (which is as well an $\mathcal{APX}$– and $\mathcal{PTAS}$–reduction) from Max-3-Sat to SOP. For the proof of this result we need the following lemma.

Lemma 2.28 *Let ϕ be an instance of* Max-3-Sat *with r clauses and let $Opt(\phi)$ denote the value of an optimal solution. Then $Opt(\phi) \geq \frac{1}{2}r$.*

Proof: Let x be an arbitrary truth assignment for the variables in ϕ. If x satisfies less than $r/2$ clauses, then the negated assignment $-x$ of x satisfies more than $r/2$ clauses. Since no assignment can satisfy more than r clauses, the proof is complete. □

Theorem 2.29 SOP *is* $\mathcal{MAX}$-$\mathcal{SNP}$*–hard.*

Proof: The L–reduction used in this proof is somewhat similar to the reduction used in the proof of Theorem 2.14. Given an instance ϕ of Max-3-Sat we create an instance $\psi = (G, d)$ of SOP. The graph $G = G_\phi$ is constructed by clipping two classes of subgraphs such that, when setting an appropriate demand d, the paths in the solution for ψ will correspond to the variables and clauses in ϕ. But the way these subgraphs work together is a little different this time.

Again, we start with the description of the reduction. Let ϕ be an instance of Max-3-Sat with clauses $C_1, \ldots, C_r$ and (boolean) variables $x_1, \ldots, x_q$. For each $i = 1 \ldots q$ let m_i denote the maximum number of occurrences of x_i and $\bar{x}_i$ in ϕ. Let $m = \max\{m_i\}$.

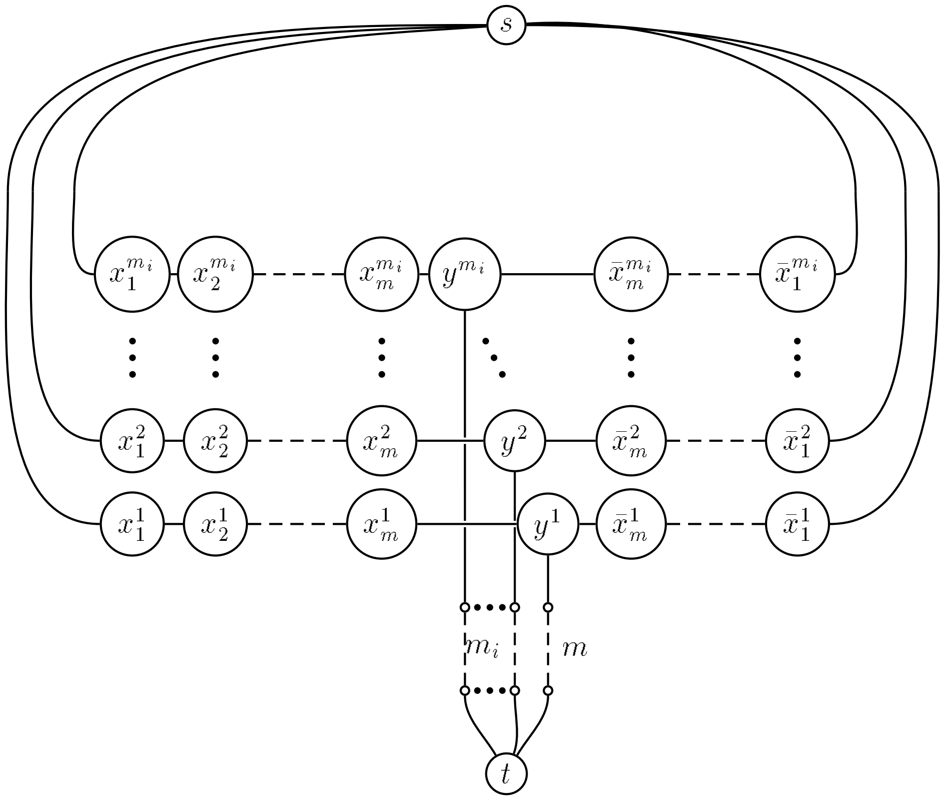


Figure 2.6: The variable subgraphs G_i

Our first class of subgraphs is the class of variable subgraphs G_i. They are defined for each variable x_i as shown in Figure 2.6. We omitted the third parameter i (the variable index) for the vertices to keep the notation simple. These subgraphs consist of m_i independent "mushroom–structures". In each such mushroom there are exactly m vertices between t and y^j, $1 \leq j \leq m_i$ (in the stem). They are needed to ensure that the resulting paths have a length as needed for the proof. Above y^j (in the cap) the path is split, and there are exactly m vertices denoted by x_k^j and $\bar{x}_k^j$, $1 \leq k \leq m$, in the same order for every mushroom–structure. Finally, the ends of those paths are linked to the vertex s.

Furthermore, we have the class of clause subgraphs H_k (see Figure 2.7). There is one such subgraph for each clause C_k in ϕ. A clause subgraph H_k always contains a special vertex c_k that is linked to t by an edge. Furthermore, it contains three or less (depending on how many literals are in this clause)

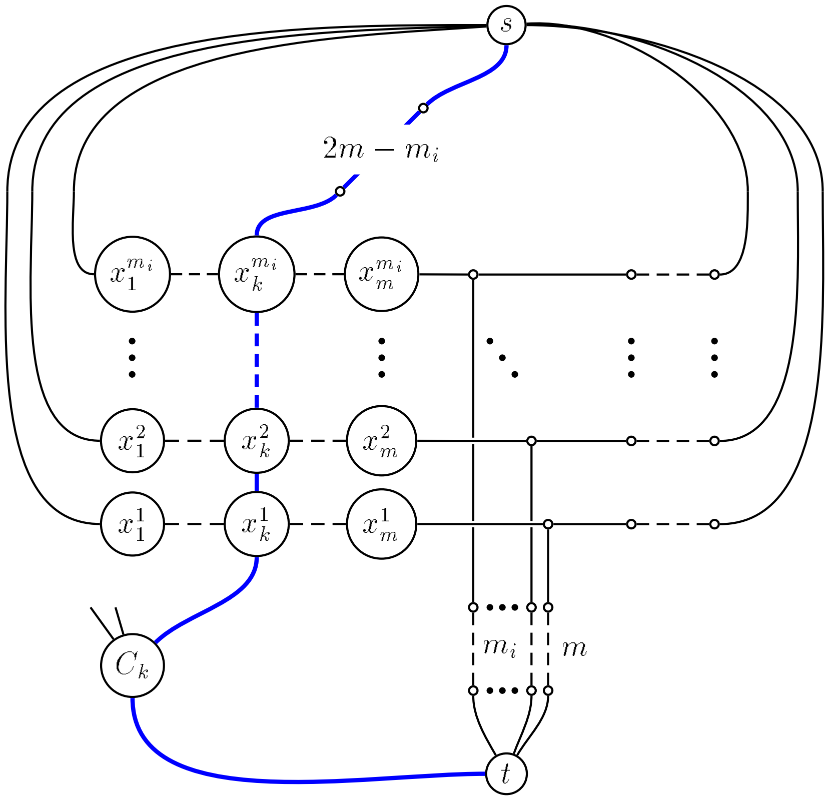


Figure 2.7: The clipping of G_i with H_k

paths from this vertex c_k to s, one for each literal in C_k. All these paths contain exactly $2m$ vertices. The first m_i vertices (in order from c_k to s) in such a path associated to variable x_i are the vertices x_k^1 to $x_k^{m_i}$ in the variable subgraph G_i. The rest of the vertices on the path is used to lengthen the path. We can assume without loss of generality that a variable x_i never occurs two times unnegated or two times negated in a single clause C_k. Since the x-vertices are also indexed by the clause index, this clipping will never cause two or more clause–paths to contain the same x-vertex.

We set the demand d to (s, t) and the length bound l to $2m + 2$.

We denote the paths $(s, x_1^*, \ldots, x_m^*, y^*, \ldots, t)$ by x-paths, the paths $(s, \bar{x}_1^*, \ldots, \bar{x}_m^*, y^*, \ldots, t)$ by $\bar{x}$-paths, and the paths $(s, \ldots, x_k^{m_i}, \ldots, x_k^1, c_k, t)$ or $(s, \ldots, \bar{x}_k^{m_i}, \ldots, \bar{x}_k^1, c_k, t)$ by c_k^i- or $\bar{c}_k^i$-paths, respectively.

Observation 2.30

(i) *The x-paths, the $\bar{x}$-paths, the c_k^i-paths and the $\bar{c}_k^i$-paths satisfy the length restriction of $l \leq 2m+2$.*

(ii) *All other paths that satisfy this length restriction must contain a vertex c_k.*

(iii) *All paths within the length bound of $2m+2$ that contain any vertex y^j have to be of the types x-path or $\bar{x}$-path.*

To complete the proof of Theorem 2.29 we need the following two lemmata.

Lemma 2.31 *Let Q be a solution of $\psi = (G,d)$. Then there is a solution $\tilde{Q}$ of ψ such that $\tilde{Q}$ contains at least as many paths as Q and $\tilde{Q}$ contains either* ***all*** *x- or* ***all*** *$\bar{x}$-paths in each G_i subgraph.*

Lemma 2.32

(a) *Let Q be a solution of $\psi = (G,d)$ that contains in each subgraph G_i either all x- or all $\bar{x}$-paths. Let p be the number of paths in Q. Then there is a solution x of ϕ that satisfies at least $c = p - \sum_i m_i$ many clauses.*

(b) *Let x be a solution of ϕ that satisfies c clauses. Then there is a solution Q of $\psi = (G,d)$ that contains exactly $p = c + \sum_i m_i$ many paths. In particular, the solution Q contains in each subgraph G_i either all x- or $\bar{x}$-paths.*

The first lemma describes the structure of the solutions that we use for the transformation. This structure is very similar to the structure of the solutions we used in the transformation to proof Theorem 2.14—there are exactly m_i vertex-disjoint length-restricted paths in each G_i whose special line-up guarantees the existence of a uniform variable setting.

The second lemma guarantees the existence of the two constants α and β in the L–reduction given above (see Definition 2.11).

Proof [Lemma 2.31]: Suppose there is a subgraph G_i such that Q neither contains all x-paths nor all $\bar{x}$-paths in G_i. We will show that we can reroute the paths that contain some vertex in G_i (potentially adding some more paths), such that either all x- or all $\bar{x}$-paths in G_i are used.

Let h be the minimum index in $\{1, \dots, m_i\}$ such that there is an (s,t)-path p containing vertex y^h. If such a path exists in Q, this path is either an x-path or an $\bar{x}$-path (see Observation 2.30). Without loss of generality, we can assume that path p is the x^h-path. Since p contains all vertices $x_1^h, \ldots, x_m^h$, there is no c_k^i-path in Q, otherwise this path would cross path p. Consequently, we can reroute each $\bar{x}^j$-path to the x^j-path for $j \geq h$. These vertices cannot belong to any other path satisfying the length restriction. We denote the set of paths created by this rerouting by Q_1. If $h = 1$ the proof is complete, because that there is no path containing a vertex y^j with $j < h$ in Q_1.

Now suppose that $h > 1$, and that there are paths in Q_1 containing some of the vertices x_k^j for $1 \leq j < h, 1 \leq k \leq m_i$. Since all of these paths have to contain one of the edges (s, x_1^j) with $1 \leq j < l$, there are at most $l - 1$ such paths in Q_1. Obviously, we can simply substitute these paths by the paths x^1 to x^{l-1}. This might only increase the number of paths. $\square$

Proof [Lemma 2.32]:

(a): Since Q contains either all x- or all $\bar{x}$-paths in each subgraph G_i, it contains at least $\sum m_i$ many paths. Thus, $c = p - \sum m_i$ is nonnegative.

We construct a solution of ϕ as follows. If Q contains all x-paths in G_i, we set the boolean variable x_i of ϕ to false, otherwise to true. All remaining paths in Q (that are no x- or $\bar{x}$-paths in some G_i) must be paths containing some vertex c_k. Hence, they cannot contain vertices $\bar{x}_k^j$ in G_i if x_i is true or vertices x_k^j if x_i is false. Consequently, if there is a path containing c_k in Q, then the clause corresponding to such a vertex c_k is satisfied. The number of those remaining paths is exactly $c = p - \sum m_i$. Hence, at least c clauses are satisfied.

(b): We construct a solution of ψ as follows. If the boolean variable x_i of ϕ is true, we add all $\bar{x}$-paths in G_i to Q, otherwise we add all x-paths. Now suppose clause C_k is true. Then there is at least one satisfied literal in this clause. We may assume w.l.o.g., that the unnegated literal x_i in C_k evaluates to true. Since we added the $\bar{x}$-paths in subgraph G_i to Q, the c_k^i-path is vertex-disjoint to all paths already in Q, and we add it to Q. Obviously, we can add either path c_k^i or path $\bar{c}_k^i$ to Q for each satisfied clause C_k. The set Q created this way contains $p = c + \sum m_i$ many vertex-disjoint length-restricted paths with the required properties. $\square$

It is clear by these two lemmas, that optimal solutions for ϕ correspond to optimal solutions for ψ.

Back to the proof of Theorem 2.29. Using these two lemmata, we proof that the reduction given above is indeed an L–reduction.

Remember that, by definition of MAX-3-SAT, we have $r \leq \sum m_i \leq 3r$.

First we will show that $Opt(\psi) \leq \alpha \cdot Opt(\phi)$, for $\alpha = 7$. This can be seen easily: Due to lemma 2.31 we know that we may transform every solution of ψ into a solution with the special properties given in the lemma. Due to lemma 2.32 we know that such a special solution corresponds to a solution of ϕ. Of course, this also holds for an optimal solution of ψ, and thus $Opt(\psi) \leq Opt(\phi) + \sum m_i \leq Opt(\phi) + 3r$. By Lemma 2.28 we have $r \leq 2 \cdot Opt(\phi)$, and the claim follows.

Second we will prove that the gap to optimality is transformed linearly. If Q is a solution of ψ and x is the solution of ϕ as constructed in Lemma 2.32, then $|Opt(\phi) - c(x)| \leq |Opt(\psi) - c(Q)|$. We may choose $\beta = 1$. Due to Lemma 2.32, we know that $c(x) = c(Q) - \sum m_i \geq c(Q) - 3r$ and $Opt(\phi) \leq Opt(\psi) - \sum m_i$. Since $3r \geq \sum m_i$, it follows that $Opt(\phi) - c(x) \leq Opt(\psi) - c(Q)$. This concludes the proof of Theorem 2.29. □

Extracting the constants from the proof and from Theorem 2.13, we obtain the following corollary.

Corollary 2.33 *There is no polynomial time ϵ–approximation algorithm for* SOP *with $\epsilon < \frac{7}{37}$, unless $\mathcal{P} = \mathcal{NP}$.*

Corollary 2.34 *There is no polynomial-time approximation scheme for* SOP*, unless $\mathcal{P} = \mathcal{NP}$.*

Finally, let us present a special case of SOP which is polynomially solvable.

Theorem 2.35 SOP *is solvable in polynomial time for length bounds $l \leq 4$.*

We will present the proof given by Itai, Perl and Shiloach [IPS82]. The basic idea is the same as in the proof of Theorem 2.25.

Proof: Let $\phi = (G, d)$ be an instance of SOP. We show that finding a maximum number of (s,t)-paths is solvable in polynomial time.

Let S be the set of all neighbors of s, T be the set of all neighbors of t, and $A = S \cap T$.

The cases $l = 1$ and $l = 2$ are trivial. Again, we just have to select the direct edge (if it exists) and if $l = 2$ all the paths that contain s,t and exactly one of the nodes in A, too.

For $l = 3$ and $l = 4$ we reduce the problem to a maximum flow problem (or to the maximum bipartite matching for $l = 3$).

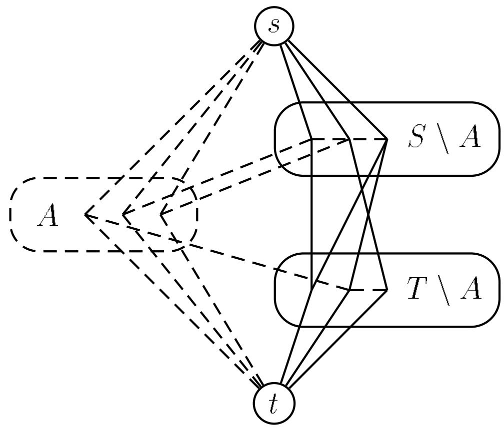


Figure 2.8: The reduced graph for $l = 3$

Lemma 2.36 *For $l = 3$ and $l = 4$ it is sufficient to consider graphs where $A = \emptyset$, there is no (s,t) edge, and S,T are stable sets.*

Proof [Lemma 2.36]: Obviously, every optimal solution contains all vertices $v \in A$. If the path that contains vertex $v \in A$ contains any other vertex, we may shorten this path without using new vertices. Thus, there is an optimal solution containing all paths (s,v,t) with $v \in A$. With the same argument we can eliminate all edges in $[S,S]$ and in $[T,T]$. If edge (s,t) exists, it is necessarily contained in an optimal solution. $\square$

In the following, we consider only graphs with $A = \emptyset$, there is no (s,t) edge, and S and T are stable sets.

In the case $l = 3$ (see Figure 2.8) we know that only vertices in $S \cup T$ are contained in a solution. We have a reduction from SOP to the maximum (s,t)-flow problem (or to the maximum bipartite matching problem), because the problem reduces to finding the maximum number of (s,t)-paths (or the maximum number of edges between S and T) in the graph where A and all edges in $[S,S]$ and in $[T,T]$ are removed.

Now let us focus on $l = 4$. Let B be the set of all vertices in $G \setminus (S \cup T)$, that have a neighbor in S and a neighbor in T (see Figure 2.9). It is easy to see that no vertex in $G \setminus (S \cup T \cup B)$ is contained in a solution. Without loss of generality we may assume that $G \setminus (S \cup T \cup B)$ is empty. Since all paths must be length bounded by 4, we can assume that B is a stable set—no path containing an edge in $[B,B]$ can be shorter than 5. If we place a new vertex $u_{(v,w)}$ on each edge (v,w) with $v \in S$ and $w \in T$, then all (s,t)-paths are

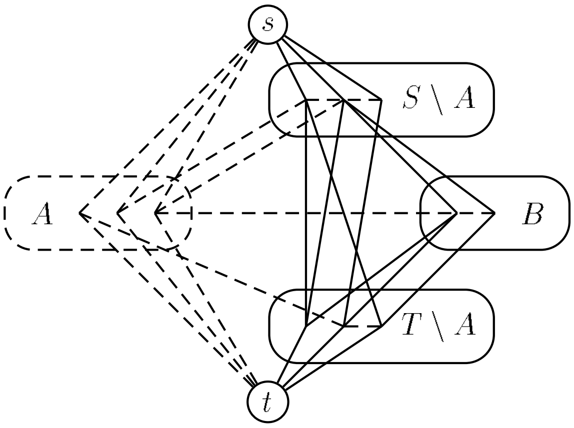


Figure 2.9: The reduced graph for $l = 4$

exactly of length 4. Our problem is now reduced to the problem of finding a maximum number of vertex-disjoint paths. $\square$

Chapter 3

Short Paths and Separating Vertex Sets

In this chapter we will present some results that are related to two of the central results in graph connectivity—Menger's Theorem and Whitney's Theorem. These theorems link the concepts of path-connectivity and cut sets in a graph.

3.1 Menger–like Results

Let $G = (V, E)$ be a graph containing the vertices s and t.

Theorem 3.1 (Menger's Theorem)

- *The maximum number of edge-disjoint (s, t)-paths in G equals the minimum number of edges in an edge (s, t)-cut.*
- *If s and t are nonadjacent, then the maximum number of vertex-disjoint (s, t)-paths is equal to the minimum cardinality of a vertex cut separating s and t.*

This theorem, which is due to Menger [Men27], can be regarded as a kind of duality theorem for graph connectivity. Unfortunately, it gives no hint on the length of the (s, t)-paths.

In the literature, there are a number of publications dealing with relations between length restricted disjoint paths and cut sets. We will present only those results, that are directly related to our work and that are useful for understanding the structure of the problems or for designing algorithms. We will not prove the theorems. Most of the proofs are technical and give no real insight into the problems. The reader is referred to the corresponding publications.

Lovàsz, Neumann-Lara, and Plummer presented in [LNLP78] some results about the relation between the size of articulation sets and the number of vertex-disjoint paths between two vertices that satisfy a length restriction (that is our single-demand case).

Let $A_l(s,t)$ be the maximum number of vertex-disjoint l-bounded (s,t)-paths in G and let $V_l(s,t)$ be the minimum number of vertices whose deletion destroys all these l-bounded (s,t)-paths.

Note that for $l = n = |V|$ we have Menger's theorem, and thus $A_n(s,t) = V_n(s,t)$. As already mentioned, this equality does not hold for arbitrary length bounds. But it is obvious, that "weak duality" holds.

Theorem 3.2 *For every $l \geq 0$ and all $s,t \in V$ we have $A_l(s,t) \leq V_l(s,t)$.*

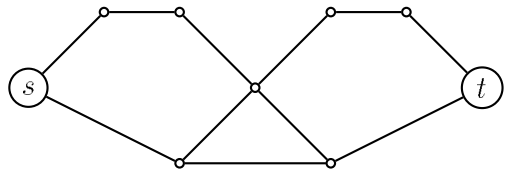


Figure 3.1: Counterexample for "strong duality"

From Figure 3.1 it can easily be seen, that equality does not hold in general. There is only one "vertex-disjoint" (s,t)-path of length $l \leq 5$, but every vertex set cutting all 5-bounded (s,t)-paths has to contain at least two vertices.

In the following we try to bound the gap from above.

It is clear that $V_l(s,t) \leq (l-1) \cdot A_l(s,t)$. Assume we have a set Q of vertex-disjoint l-bounded (s,t)-paths with $|Q| = A_l(s,t)$. Then the set containing all inner vertices of all paths in Q destroys all l-bounded (s,t)-paths. Otherwise Q is not maximal.

The following two theorems sharpen this simple relation. The first one gives a bound depending on the distance between s and t.

Theorem 3.3 *For every $l \geq 0$ and all $s,t \in V$ with $l \geq d(s,t)$ we have $V_l(s,t) \leq (l - d(s,t)+1) \cdot A_l(s,t)$, where $d(s,t)$ denotes the distance between s and t.*

The next theorem gives a bound depending only on l.

Theorem 3.4 *For every $l \geq 0$ and all nonadjacent $s,t \in V$ we have $V_l(s,t) \leq \lfloor \frac{n}{2} \rfloor A_l(s,t)$.*

By this theorem, it follows that $A_2(s,t) = V_2(s,t)$ and $A_3(s,t) = V_3(s,t)$ for all $s,t \in V$. This equality can also be shown for a length bound of $l = 4$, but Lovàsz et. al. pointed out that already for $l = 5$ there are graphs with $V_5(s,t) = 2\,A_5(s,t)$.

Theorem 3.5 *For length bounds $l \leq 4$ the equality $V_l(s,t) = A_l(s,t)$ holds.*

Thus, the "Menger Theorem for length restricted paths" is true for exactly those length bounds, for which SFP is solvable in polynomial time.

Lovàsz et. al. conjectured that the bound given in Theorem 3.4 can be improved to a factor of $\lfloor \sqrt{\frac{n}{2}} \rfloor + 1$.

3.2 Whitney–like Results

A graph $G = (V,E)$ is said to be k-connected, if $|V| > k$ and $G - X$ is connected for any $X \subseteq V$ with $|X| \leq k-1$. This is equivalent to saying that there are at least k vertex-disjoint paths between any pair of vertices in G. This result is due to Whitney [Whi32].

Theorem 3.6 (Whitney's Theorem)

- *A graph $G = (V,E)$ with $|V| > k$ is k-connected if and only if there are k vertex-disjoint paths between each pair of vertices.*
- *A graph $G = (V,E)$ is k-edge-connected if and only if there are k edge-disjoint paths between each pair of vertices.*

As in Menger's theorem, nothing is said about the length of the paths that connect these pairs of vertices.

In this section we present the results of Faudree, Ordman, Schelp, Jacobson and Tuza [FJO+87] related to this topic.

Let G be a simple graph with n vertices, l be a length bound with $0 < l \leq n-1$, and k be the diversification parameter with $0 < k \leq n-1$. We denote by $P_{l,k}(G)$ the property that between each pair of vertices $s, t \in V$ there are at least k vertex-disjoint paths of length at most l.

The first two theorems state that very large density or very large connectivity of a graph G ensure that G has property $P_{l,k}$.

Theorem 3.7 *For a graph $G = (V, E)$ and integers $l \geq 2$ and $k \geq 1$ such that at least one the following conditions holds, property $P_{l,k}$ is satisfied.*

(i) *G has at least $\binom{n-1}{2} + k$ edges.*

(ii) *G has a minimum degree of $\lfloor \frac{n+k}{2} \rfloor$.*

Theorem 3.8 *If G is $(\frac{n-k}{l} + k)$-connected, then property $P_{l,k}$ is satisfied.*

If G is already known to be k-connected the degree condition can be relaxed.

Theorem 3.9 *If G is k-connected and G has minimum degree exceeding $\lfloor \frac{n-k+2}{\lfloor (l+4)/3 \rfloor} \rfloor + m - 2$, then property $P_{l,k}$ is satisfied.*

Chapter 4

Algorithmic Approaches to the Single-Demand Problems

In this chapter we present the algorithms we implemented for the single-demand problems. Due to the results of Chapter 2, we cannot expect exact or optimal solutions (except in some special cases) in polynomial time. Furthermore, we have shown that even an approximation within reasonable bounds is impossible in the general weighted case. Hence, we cannot give performance guarantees for the solutions computed by our heuristics, except in the special cases we mentioned in the last chapter. Nevertheless, our heuristics perform very well in practice, as shown by the results in the appendix.

Before presenting the algorithms, we have to introduce a little more notation. Since standard network flow techniques apply to directed graphs, we implemented the algorithms described in this chapter on digraph data structures. The corresponding directed versions of the single-demand problems are defined as follows. For a graph $G = (V, E)$ and a demand $d = (s, t, k, l)$, with $s, t \in V$, $1 \le l \le |V| - 1$, and $1 \le k \le |V| - 1$, let $D = (V, A)$ be the associated digraph of G and $\overrightarrow{d} = (s, t, k, l)$ be the directed demand associated with d. For each cost function $c : E \to \mathbb{N}$ on the edges of G let $\overrightarrow{c} : A \to \mathbb{N}$ be the associated cost function on the arcs of D, that is $\overrightarrow{c}(v, w) := c(v, w)$ for all $(v, w) \in A$. For notational convenience we denote $d := \overrightarrow{d}$ and $c := \overrightarrow{c}$ in the directed versions of the problems, too. This does not conflict with the undirected case.

Let $D = (V, A)$ be a digraph and $d = (s, t, k, l)$ be a directed demand in D. As in the directed case, a solution (of a directed single-demand problem) is a collection of vertex-disjoint l-bounded directed (s, t)-paths in D. A solution is feasible, if it contains at least k paths. For each of the undirected single-demand problems we define the corresponding directed single-demand problem in the natural way: In the directed version of SFP the aim is to find a feasible solution, i.e., a collection of vertex-disjoint l-bounded directed (s,t)-paths, etc. For simplicity, we use the acronyms SOP, SFP, and WSP for the directed versions of the problems, too.

Since paths are vertex-simple, it is obvious that each (s,t)-path in G corresponds to a directed (s,t)-path in D containing the same vertices in the same order, and vice versa. It is easy to see that there is a 1–1 correspondence between collections of vertex-disjoint l-bounded (s,t)-paths in G and collections of vertex-disjoint l-bounded directed (s,t)-paths in D. Hence, the undirected single-demand problems are equivalent to the directed single-demand problems on the associated graphs, demands and cost functions.

In Section 4.1 we present the preprocessing and problem reduction steps performed in the beginning to reduce the size of the problems.

A special algorithm for the polynomially solvable one path case, i.e., where the diversification is $k = 1$, is presented in Section 4.2.

In Section 4.3 we present a class of heuristics that can be characterized as flow heuristics. These heuristics use standard max-flow and min-cost-flow techniques to compute solutions for the single-demand problems. To apply these techniques we have to relax the single-demand problems by dropping the path length restrictions. After running these flow heuristics, which may create path collections violating the length restriction, we apply several postprocessing methods to obtain solutions that satisfy the length restriction. These postprocessing methods are described in Section 4.3.5.

Another approach, based on augmenting a given solution path by path until the desired diversification is reached, is presented in Section 4.4. This approach uses a depth-first-search strategy to augment a given solution, such that, given a collection of paths that satisfy the length restriction, each of the paths in the new (augmented) collection satisfies the length restriction, too. We present the original algorithm proposed by Perl and Ronen [PR84] and our modifications to cope with some problems of this approach.

In Section 4.5 we present the exact enumeration algorithms we implemented. Finally, in Section 4.6 we present the entire heuristic frameworks for the single-demand problems that describe how the various heuristics work together. Of course, our frameworks solve the polynomially solvable cases described in Chapter 2 to optimality.

4.1 Preprocessing

In the beginning of each computation various tests are carried out in order to reduce the size of the problem in terms of considered edges and vertices. These tests can be categorized into two types: **demand reduction** and **deletion** of edges and vertices.

In the demand reduction tests we try to fix some (s,t)-paths that are contained in every optimal solution. These tests apply only to the unweighted problems SFP and SOP.

Let $\phi = (D, d)$ be the given instance of (the directed version of) SOP, where $d = (s, t, k, l)$ is an (s,t)-demand in D, or let $\psi = (D, d)$ be the corresponding given instance of SFP. Let S and T denote the sets of neighbors of s and t, respectively, i.e., $S := \{v \in V \mid (s,v) \in \delta(s) \, or \, (v,s) \in \delta(s)\}$ and $T := \{v \in V \mid (t,v) \in \delta(t) \, or \, (v,t) \in \delta(t)\}$. Define $U := S \cap T$.

The demand reduction tests are based on the following two simple observations:

(i) Each optimal solution for ϕ contains the direct arc (s,t), if it exists.

(ii) There is an optimal solution for ϕ that contains all paths (s, u_i, t), $u_i \in U$.

The proof of these two observations was given in Lemma 2.36. Analogously, if there is a feasible solution for ψ, then there is a feasible solution of ψ that contains the arc (s,t), if it exists, and all paths (s, u_i, t) with $u_i \in U$. According to this observation, in the demand reduction tests we compute all paths of length $l \leq 2$, delete the edges and internal vertices of these paths from D, and decrease the diversification parameter k of the demand d by the number of paths found ($|U| + 1$ iff $(s,t) \in A$, $|U|$ otherwise). For an illustration see Figure 4.1—the paths selected in the demand reduction tests are drawn bold.

In the second type of tests, the so called deletion test, we remove everything from D, that cannot be contained in a solution of a demand problem. The first test of this type is based on an observation proved in Lemma 2.36:

(i) In the unweighted case it is sufficient to consider graphs where S and T are stable sets.

(ii) There is an optimal solution for ϕ, such that no path in this solution contains two vertices in S or two vertices in T.

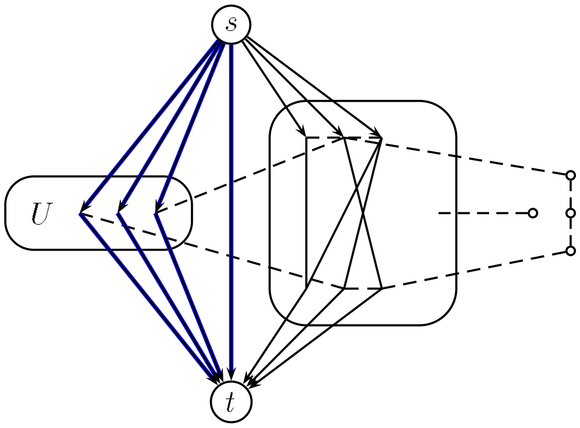


Figure 4.1: Problem reduction for SFP and SOP

Analogously, if there is a feasible solution of ψ, then there is a feasible solution of ψ, where no path contains two vertices in S or two vertices in T. Hence, we can delete all edges in $[S, S]$ and all edges in $[T, T]$. Furthermore, since we are working on digraphs, we can remove all arcs $a \in [V \setminus \{s\}, S] \cup [T, V \setminus \{t\}]$. Thus, S can be "entered" from s only and T can be "left" to t only. This deletion test applies only to the unweighted problems SFP and SOP.

The other three deletion tests apply to all single-demand problems. First of all, we can delete all arcs in $\delta^+(t)$ and all arcs in $\delta^-(s)$.

The second test is the length test: Let $d_1(u, v)$ denote the distance between the nodes u and v in D, that is the length of a shortest (u, v)-path with respect to the number of arcs. We can delete vertex v from D if $d(s, v) + d(v, t) > l$. We can delete arc $a = (v, w)$ from D if $d(s, v) + 1 + d(w, t) > l$. The distances $d(s, v)$ and $d(v, t)$ can be computed simultaneously for all $v \in V$ applying a shortest path tree algorithm two times, once for root s and once for root t.

The last deletion test is the so called degree test. It is obvious, that we can remove all vertices $v \in V \setminus \{s, t\}$ with $|\delta^-(v)| = 0$ or $|\delta^+(v)| = 0$. Additionally, we can remove all $v \in V \setminus \{s, t\}$ for which $|\delta^-(v)| = |\delta^+(v)| = 1$ and the arcs in $\delta^-(v)$ and $\delta^+(v)$ are antiparallel.

The last two tests, namely the length test and the degree test, are repeated as long as some vertex or some arc can be deleted, because the distances and degrees may change after a deletion.

For an illustration of the deletion test see Figure 4.1. The arcs and the vertices removed from D are drawn dashed.

Preprocessing

```
if a = (s,t) ∈ D then                                      // remove direct arc
  D := D − a, k := k − 1
for all v ∈ S ∩ T do                                       // remove length 2 paths
  D := D − v, k := k − 1
for all a ∈ [V \ {s}, S] ∪ [T, V \ {t}] ∪ δ⁻(s) ∪ δ⁺(t) ∪ {(t,s)} do
  D := D − a                                               // remove unnecessary arcs
repeat
  compute d(s,v) and d(v,t) for all v ∈ V
  for all v ∈ V \ {s,t} do
    if d(s,v) + d(v,t) > l then                            // length test for vertices
      D := D − v
    if |δ⁺(v)| = 0 or |δ⁻(v)| = 0 then                     // degree tests for vertices
      D := D − v
    if |δ⁺(v)| = |δ⁻(v)| = 1 and δ⁺(v), δ⁻(v) are antiparallel then
      D := D − v
  for all a = (v,w) ∈ A do
    if d(s,v) + 1 + d(w,t) > l then                        // length test for edges
      D := D − a
while some vertex or edge deleted
```

Figure 4.2: Preprocessing for SFP or SOP

The complete preprocessing for SFP or SOP is shown in Figure 4.2.

4.2 The One Path Case

If the diversification parameter is $k = 1$, the unweighted problem SFP can be solved by a shortest path algorithm, for example Dijkstra's labeling algorithm [Dij59]. If the shortest path satisfies the length restriction, we have a feasible solution. Otherwise no path can satisfy the length restriction.

The weighted case is not that easy. Nevertheless, also the weighted problem WSP is polynomially solvable for diversification $k = 1$. The idea is to modify Dijkstra's shortest path algorithm. We use the fact that all arc lengths are equal (they are 1).

Let $\phi = (D, d, c)$ be an instance of WSP, where $D = (V, A)$ is a digraph, $d = (s, t, k, l)$ is a demand in D with diversification $k = 1$, and c is a nonnegative cost function on the arcs of D. In the following we will interpret c as a second

Min-Cost-Length-Restricted-Path

$U := V \setminus \{s\}$ // *Initialize*
$R := N^+(s)$
$d_i(v) := \begin{cases} 0 & \text{if } v = u, \\ c_e & \text{if } a = (s,v) \in A, \\ \infty & \textit{otherwise} \end{cases}$
while $R \neq \emptyset$ // *There are reachable vertices*
  find $v \in R$ and $1 \leq i \leq l$ with // *markable vertex*
   (i) $d_i(v) \leq d_j(w)\ \forall w \in R,\ 1 \leq j \leq l$
   (ii) $d_i(v) = d_j(w) \Rightarrow j \geq i,\ \forall w \in R,\ 1 \leq j \leq l$
  $U := U \setminus \{v\}$ // *mark*
  $R := R \setminus \{v\}$
  for all $w \in N^+(v) \cap U$ **do** // *update neighbors*
   for $j = i + 1$ **to** $j \leq l$ **do**
    $d_j(w) := \min\{d_{j-1}(w), d_j(w), d_{j-1}(v) + c_{(v,w)}\}$
    $p_i(w)$ according to *argmin* from above
   if $d_l(w) < \infty$ **then**
    $R := R \cup \{w\}$

Figure 4.3: The Min-Cost-Length-Restricted-Path algorithm

arc length function. Let l be the given length bound ($1 \leq l \leq n-1$). As Dijkstra's labeling algorithm, the Min-Cost-Length-Restricted-Path algorithm performs a number of iterations. In each iteration a vertex is marked and the distance labels of its neighbors are updated. Unlike in Dijkstra's algorithm, we have a distance label and a predecessor arc for each path length bound i, $1 \leq i \leq l$, assigned to every vertex. For all $v \in V$, $1 \leq i \leq l$, let $d_i(v)$ be the length of a shortest path from s to v with respect to c using at most i arcs in the current iteration of the algorithm. Let U denote the set of all unmarked vertices and let $R := \{v \in U \mid \exists j :\ 1 \leq j \leq l,\ d_j(v) < \infty\}$ the set of all reachable unmarked vertices in the current iteration. Let $N^+(v)$ denote the set of all directed neighbors of v, i.e., all $w \in V$ with $(v,w) \in \delta^+(v)$. In the algorithm we keep two l-dimensional arrays for each vertex: the distance labels $d_i(v)$ and the predecessor arcs $p_i(v)$ for each path length bound i, $1 \leq i \leq l$. In each iteration, we mark a vertex v with the smallest distance $d_i(v)$ from u (with respect to c). We mark the vertex, where this distance is obtained for the minimal path length bound i among the unmarked vertices with this smallest distance for some path length bound j. The distance labels $d_j(w)$ and the predecessors $p_j(w)$ are updated for each of its neighbors $w \in N^+(v)$ and for each path length bound j, $i < j \leq l$. Repeatedly applying this procedure, we compute for each vertex $v \in V$ the length (weight) of the shortest (with resp. to c) length bounded path and a set of l shortest path trees to reconstruct the paths. After the last iteration $d_l(v)$ is the length (weight) of a shortest (s, v)-path with no more than l arcs. Backtracking the predecessor arcs of the appropriate path length bounds, the path itself can be constructed.

The complete Min-Cost-Length-Restricted-Path algorithm is shown in Figure 4.3. Since this algorithm works exactly like Dijkstra's labeling algorithm, except the fact we have to update the distances and predecessors for each path length bound for each neighboring vertex, the time complexity of this algorithm is in $O(n^3)$.

The correctness of the algorithm can be seen analogously to Dijkstra's algorithm. In each iteration all distance labels are correct for the marked vertices. Since we mark an unmarked vertex with the smallest distance and the fewest arcs in the corresponding path and all paths corresponding to vertices in R at some path length bound can only contain already marked vertices as intermediate vertices, the selected vertex has correct distance labels and predecessors after the marking.

4.3 Flow Techniques

In this section we present several heuristics which are based on flow techniques. As we have seen in Chapter 2, the path length restriction makes the single-demand problems computationally difficult. But if this restriction is dropped, the problems become easy.

4.3.1 Interlacing Paths and Vertex-Disjoint Augmentation

Using flow techniques, many path problems can be modeled by integer flows in digraphs. We interpret each (s, t)-path as an (s, t)-flow of value 1 on the arcs of the path. Overlapping of paths results in higher flow values on the overlapping arcs. If two paths contain a common vertex, then the flow on this vertex is larger than 1. Path disjointness conditions can be modeled by capacities on the arcs and the vertices.

We can formulate the vertex disjointness conditions of the single-demand problems in various ways. The simplest way is to impose vertex capacities of 1 on all vertices $v \in V \setminus \{s,t\}$. We do not need to impose arc capacities, because the arc-disjointness constraints are satisfied automatically if the vertex capacity constraints are fulfilled. It is obvious, that each integer (s,t)-flow with flow value of k that satisfies these vertex capacity constraints can be decomposed into k (s,t)-path-flows of value 1 and some circle-flows of value 1, such that these paths and cycles are vertex-disjoint.

Another way to model the vertex disjointness conditions is the so called vertex-splitting, where we introduce auxiliary arcs and vertices. Each vertex $v \in V$ is replaced by two vertices v_i and v_o that are connected by an arc $a_v = (v_i, v_o)$. Each arc $(w,v) \in A$ is replaced by an arc (w_o, v_i) with the same length. The length of arc $a_v = (v_i, v_o)$ is 0 for all $v \in V$. Note that each of the vertices now has either an indegree or an outdegree of 1. We impose arc capacities of 1 on all arcs. Now it is easy to see, that integer (s,t)-flows of value k correspond to collections of k (s,t)-paths and some cycles, where the paths and cycles are vertex-disjoint.

In the following we will define the special version of augmenting paths for vertex-disjoint (s,t)-path problems (that correspond to 0/1 (s,t)-flow problems with vertex capacities of 1), called interlacing paths. We use the first formulation of the vertex disjointness constraints, where vertex capacities of 1 are imposed. This notation is due to Suurballe [Suu74].

Definition 4.1 (Signed path) *Let $p = (v_0, \pm a_1, v_1, \pm a_2, \ldots, v_l)$ be a sequence of vertices $v_0, ldots, v_l \in V$ and signed arcs $a_1 \ldots, a_l \in A$. We call p a signed (v_0, v_l)-path if the following conditions hold:*

(i) For each arc $a_i \in p$ with a positive sign v_{i-1} is the tail of a_i and v_i is the head of a_i.

(ii) For each arc $a_i \in p$ with a negative sign v_{i-1} is the head of a_i and v_i is the tail of a_i.

(iii) No arc occurs multiple times in p.

Definition 4.2 (Interlacing) *Let $D = (V, A)$ be a digraph, $s, t \in V$, and S be a subgraph of D consisting of vertex-disjoint (s,t)-paths. A signed (s,t)-path p interlaces S if the following two conditions hold:*

(i) An arc $a \in p$ is in S if and only if it has a negative sign.

(ii) A vertex $v \in p$ is in S if and only if it is incident to an arc with negative sign.

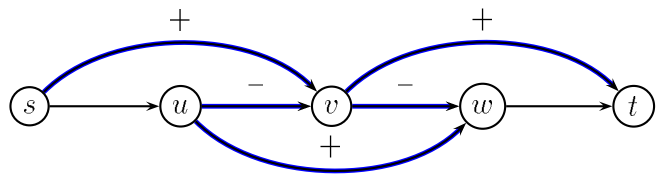


Figure 4.4: Interlacing

Note that a signed path need not be vertex-simple. Nevertheless, we use the term “path”, because interlacing paths are (vertex-simple) paths if the vertex-splitting model is used to formulate the vertex-disjointness conditions. For an example of an interlacing path see Figure 4.4, where S is the subgraph consisting of the path (s, u, v, w, t) and $p = (s, {}_{+}(s, v), v, {}_{-}(u, v), u, {}_{+}(u, w), w, {}_{-}(v, w), v, {}_{+}(v, t), t)$ interlaces S.

The augmentation of a collection of k vertex-disjoint (s, t)-paths to a collection of $k + 1$ vertex-disjoint (s, t)-paths works exactly like the standard flow augmentation. The following theorem is a reformulation of the corresponding flow augmentation theorem.

Theorem 4.3 (Augmentation) *Let $D = (V, A)$ be a digraph, $s, t \in V$, and Q be a collection of k vertex-disjoint (s, t)-paths in D. Let S be the subgraph of D consisting of the paths in Q and let p be an interlacing (s, t)-path for S in D. Define $S' := S \Delta p$ (where Δ denotes the symmetric difference). Then S' consists of $k + 1$ vertex-disjoint (s, t)-paths in D.*

It is easy to see (by transformation to the standard flow problems), that we have strong duality between the number of vertex-disjoint (s, t)-paths and the cardinality of (s, t)-vertex-cuts.

We can create a collection of k vertex-disjoint (s, t)-paths by starting with the empty collection and augmenting with interlacing paths, analogously to the flow augmentation with additional flow on augmenting paths. Given nonnegative arc costs, we can compute such a collection with minimal total costs. Analogously to the min-cost-flow augmentation, we have to augment the current solution with a shortest interlacing path (with respect to the costs). Since in each augmentation step we increase the number of paths by 1 and the maximal number of vertex-disjoint (s, t)-paths is bounded by $n - 1$, we have to perform at most $n - 1$ many augmentations.

The interlacing paths needed in each augmentation step can be found in polynomial time: If we use the vertex-splitting model, we can use a generic shortest path algorithm for arbitrary arc lengths to find an interlacing path. Given a collection Q of vertex-disjoint (s,t)-paths, we first reverse each arc contained in a path in Q and negate its length. Then we apply the shortest path algorithm to find a shortest augmenting (s,t)-path, which is also a shortest interlacing (s,t)-path for Q. In our implementation we neither use vertex-splitting nor reverse the arcs in Q. We use the arc length reduction described in Section 4.3.2 to guarantee nonnegative arc lengths in each iteration and a modified version of Dijkstra's labeling algorithm which does the vertex-splitting implicitly (see Algorithm Min-Length-Interlacing 4.5). Similarly to Dijkstra's labeling algorithm, Min-Length-Interlacing computes a shortest interlacing path tree. From this tree we can reconstruct the shortest interlacing (s,v)-path for each $v \in V$.

The entire Min-Total-Length-Paths algorithm is shown in Figure 4.6. It computes a collection Q of k vertex-disjoint (s,t)-paths of minimal total length with respect to the arc lengths c. The augmentation is performed using a minimum length interlacing path computed by the Min-Length-Interlacing algorithm (see Figure 4.5). The paths can be reconstructed at the end by a simple path-decomposition of the subgraph S (that consists of exactly those paths). Since our algorithms do the vertex-splitting implicitly, we need some additional data. The vector $\mathbf{l}$ contains for each vertex $v \in V$ the length $l(v)$ that the arc (v_i, v_o) would have in the vertex-splitting model. For each $v \in V$ the values $d_i(v)$ and $d_o(v)$ are the distances that the vertices v_i and v_o would have from s_i in the vertex-splitting model. After each augmentation we reduce the arc lengths $\mathbf{c}$ and the internal "vertex lengths" $\mathbf{l}$ by the feasible potential given in $\bar{\mathbf{d}}_\mathbf{i}$ and $\bar{\mathbf{d}}_\mathbf{o}$ (see Section 4.3.2). The distances $\mathbf{d_i}$ and $\mathbf{d_o}$ are updated in each iteration by Min-Length-Interlacing.

Finally, let us give a short description of the Min-Length-Interlacing algorithm. If $Q \neq \emptyset$, we have to assume that c and l are the reduced arc length function and the reduced internal vertex lengths as computed by Min-Total-Length-Paths, respectively. Otherwise c may be any nonnegative arc length function and l must be 0 for all vertices and the algorithm reduces to Dijkstra's shortest path algorithm. Let U denote the set of unmarked vertices and $R := \{v \in U \,|\, d(v) < \infty\}$ the set of reachable unmarked vertices. Let $N^+(v) := \{u \in V \,|\, (v,u) \in \delta^+(v)\}$. For each vertex $v \in V$ we have two distances $d_i(v)$ and $d_o(v)$ that correspond to the distances the vertices v_i and v_o would have in the vertex-splitting model from s_i. Furthermore, we have a predecessor arc $p(v)$for each $v \in V$ to reconstruct the interlacing. In each iteration we mark the vertex v with the smallest distance $d_i(v)$ and

Min-Length-Interlacing

```
U := V \ {s}                                             // init
R := N⁺(s) \ {v ∈ V | v ∈ Q}
d(v) := { 0    if v = s,
        { c_a  if a = (s, v) ∈ A and a ∉ Q,
        { ∞    otherwise
p(v) undefined for all v ∈ V
while R ≠ ∅ and t ∈ U
  find v ∈ R with d_i(v) ≤ d_i(w) ∀w ∈ R
  U := U \ {v}                                           // mark vertex
  R := R \ {v}
  if v ∉ Q then                                          // free vertex
    d_o(v) := d_i(v) + l(v)
    for all w ∈ N⁺(v) ∩ U do                             // normal neighbor update
      if d_o(v) + c_(v,w) < d_i(w) then
        d_i(w) := d_o(v) + c_(v,w)
        p(w) := (v, w)
        R := R ∪ {w}
  if v ∈ Q then                                          // backtrack on path
    let u be the predecessor of v on the path in Q
    d_o(u) := d_i(v) + c_(u,v)
    if d_i(v) + c_(u,v) + l(u) < d_i(u) then             // update for u
      d_i(u) := d_i(v) + c_(u,v) + l(u)
      p(u) := (u, v)
      R := R ∪ {u}
    for all w ∈ (N⁺(u) ∩ U) with w ∉ Q do                // local neighbor update
      if d_o(u) + c_(u,w) < d_i(w) then
        d_i(w) := d_o(u) + c_(u,w)
        p(w) := (u, w)
        R := R ∪ {w}
for all v ∈ V do                                         // feasible potential
  d̄_i(v) := min{d_i(v), d_i(t)}
  d̄_o(v) := min{d_o(v), d_i(t)}
create the interlacing p by backtracking the predecessors p_v
```

Figure 4.5: Min-Length-Interlacing algorithm

update the distances and predecessors of its neighbors (or of the neighbors of its predecessor). If v is not contained in Q, we use the standard neighbor update method of Dijkstra's algorithm. Note that the neighbors of v are neighbors of v_o in the vertex-splitting model—we have to set the correct

Min-Total-Length-Paths

$Q := \emptyset$
$\mathbf{l} := \mathbf{0}$ *// internal node lengths*
$\bar{\mathbf{d}}_\mathbf{i}, \bar{\mathbf{d}}_\mathbf{o} := \mathbf{0}$ *// v_i and v_o potentials*
for $i = 1$ **to** k **do**
  Min-Length-Interlacing$(Q, c) \rightarrow p$ *// interlacing, $\bar{\mathbf{d}}_\mathbf{i}, \bar{\mathbf{d}}_\mathbf{o}$ are modified*
  Length-Reduction$(\mathbf{c}, \mathbf{l}, \bar{\mathbf{d}}_\mathbf{i}, \bar{\mathbf{d}}_\mathbf{o})$
  if $p = \emptyset$ **then**
    exit "There are no k vertex-disjoint paths."
  augment Q by p *// augment path collection*

Figure 4.6: Min-Total-Length-Paths algorithm

distance $d_o(v)$ before updating the neighbors. Since we cannot "cross" a path in Q in a single vertex, we have to backtrack this path. If vertex v is contained in a path $p_i \in Q$, we first backtrack to its predecessor vertex u on p_i and update its distances and predecessors if necessary. Then we update the distances and predecessors of all neighbors of u (except those on the path p_i) without marking vertex u. Again, we have to do consider the pseudoarcs (v_i, v_o) from the vertex-splitting model before updating u or its neighbors. After the last iteration the shortest interlacing (s,t)-path can be reconstructed by backtracking the predecessor arcs of each vertex. These predecessors build the shortest interlacing path tree rooted at s. Note that vertices in Q have to be treated according to the "local" neighbor update done in the construction of the tree.

4.3.2 Length Reduction

As mentioned in the previous section, we do a length reduction after in each augmentation step after computing the interlacing path. With this technique we can use the fast Min-Length-Interlacing algorithm (for nonnegative lengths). Without this reduction method we would have to use a (slower) shortest interlacing path algorithm for arbitrary lengths.

Let $D = (V, A)$ be a digraph and $c : A \rightarrow \mathbb{R}$ be a length function on the arcs of D. We call a function $\pi : V \rightarrow \mathbb{R}$ a **feasible potential** (with respect to c) if $\pi(v) - \pi(u) \leq c(u, v)$ for every arc $(u, v) \in A$.

Theorem 4.4 *If $c : A \rightarrow \mathbb{R}$ is nonnegative, then there exists a feasible potential $\pi : V \rightarrow \mathbb{R}$. Furthermore, if c is integer, then there is an integer*

Length-Reduction

for all $a = (u, v) \in A$ **do** // *arc length reduction*
  if $a \notin Q$ **then**
    $c(a) := c(a) + d_o(u) - d_i(v)$
  else
    $c(a) := c(a) - d_o(u) + d_i(v)$
for all $v \in V$ **do** // *internal vertex length reduction*
  if $v \notin Q$ **then**
    $l(v) := l(v) + d_i(v) - d_o(v)$
  else
    $l(v) := l(v) - d_i(u) + d_o(v)$

Figure 4.7: Length-Reduction method

feasible potential.

Theorem 4.4 guarantees only the existence of a feasible potential. The next theorem is of more practical importance.

Theorem 4.5 *Let s be a vertex in V and $c : A \to \mathbb{R}$ be a nonnegative cost function. For all $v \in V$ denote by $d(v)$ the distance from s to v with respect to c. Then $d : V \to \mathbb{R}$ is a feasible potential.*

Suppose we are given a shortest path tree with respect to c rooted in s. Let $d(v)$ denote the length of a shortest (s, v)-path. Due to Theorem 4.5, d is a feasible potential with respect to c. For each arc $(u, v) \in A$ we define a new arc length function $c'(u, v) := c(u, v) + \bar{d}(u) - \bar{d}(v)$. Obviously, the new arc lengths have the following properties:

Observation 4.6

(i) *c' is nonnegative,*

(ii) *for all $a \in A$ that are contained in the shortest path tree (rooted in s and with respect to c) we have $c'(a) = 0$,*

(iii) *the length of directed cycles does not change, i.e., $c'(h) = c(h)$ for all directed cycles h in D, and*

(iv) *for all directed (s, t)-paths p we have $c'(p) = c(p) - d(t)$.*

Using this arc length reduction, we can apply a shortest path algorithm for nonnegative arc lengths to compute a shortest interlacing path (if the vertex-splitting model introduced in the previous section is used). Due to property (ii), the new arc lengths are $c'(a) = 0$ for all arcs a contained in the interlacing path created. Thus, the "reversed" lengths $-c'(a)$ are nonnegative, too. Since only arcs contained in the interlacing path are reversed, the new arc lengths $c'(a)$ are nonnegative for all arcs $a \in A$.

As mentioned above, in our implementation we do not reverse any arc. The reversion is done implicitly in the Min-Length-Interlacing algorithm. Hence, we have to modify the arc length reduction method described in this section, such that the arc lengths and the internal "node lengths" (that are the lengths of the arcs (v_i, v_o) in the vertex-splitting model) are reduced by the potential returned by Min-Length-Interlacing in d_i and d_o. The complete reduction used in our algorithms is shown in Algorithm 4.7.

4.3.3 Primal and Dual Versions

Applying Min-Total-Length-Paths with different arc length functions and different diversification parameters we obtain various heuristics. There are three cases of special interest for our single-demand problems, one primal and two dual methods.

The first version is the Min-Cost-Paths heuristic. It can be used for the weighted problem WSP. Given an instance $\phi = (D, d, c)$ of WSP, we interpret the cost function $c : A \to \mathbb{N}$ as arc lengths. Applying Min-Total-Length-Paths with these arc lengths, we obtain an algorithm that solves the problem of finding k vertex-disjoint (s,t)-paths at minimal cost This is the relaxation we get from WSP by dropping the path length restriction. If we run this heuristic with the diversification parameter k given in the demand we have a dual heuristic—the sum of the costs of the resulting (unrestricted) vertex-disjoint paths is a lower bound for the sum of the costs of any set of k many length-restricted vertex-disjoint paths. If all paths in the collection computed by this heuristic satisfy the length restriction, then this collection is optimal.

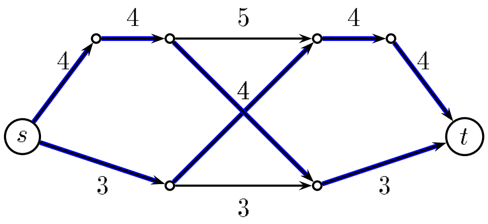
The second version, due to Suurballe [Suu74], is to set all arc lengths to 1. Applying Min-Total-Length-Paths with these arc lengths, we obtain an algorithm that finds a collection of k vertex-disjoint (s,t)-paths with the minimal total number of arcs. Note that this is no relaxation of any of the original problems. We just replaced the path length restriction with the objective to minimize the total number of arcs used. If we run this heuristic with the diversification parameter k given in the demand, we have a "primal" method. In practice the path collections returned by this heuristic are much better than arbitrary collections of (s,t)-paths. If we apply this method without a diversification parameter, we have a dual method that computes the maximal number of vertex-disjoint (s,t)-paths. Obviously, this number is an upper bound for the maximal number of length restricted vertex-disjoint (s,t)-paths. The last two heuristics can be applied for the unweighted problems SOP and SFP and for the weighted problem WSP. Nevertheless, in the weighted case we use the Min-Cost-Paths heuristic first, because the versions of Suurballe's heuristic do not take the arc costs into account.

4.3.4 Perturbation

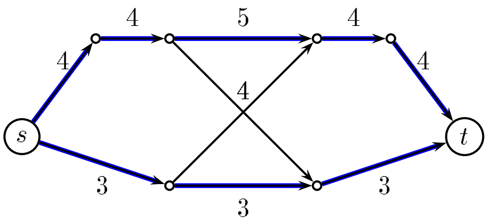
To avoid extensive postprocessing, we perturbate the arc lengths before applying Min-Total-Length-Paths in the Min-Cost-Paths heuristic or in Suuballe's heuristic.

The first perturbation method can be used for both flow heuristics—the Min-Cost-Paths heuristic and Suurballe's heuristic. It modifies the arc lengths in such a way, that arcs "close" to s and t, i.e., arcs that may lie on short (s,t)-paths (with respect to the number of arcs in the path) are preferred to arcs "far away" from s and t. For each pair of vertices $u, v \in V$ let $d_1(u,v)$ be the distance between u and v with respect to the arc length function $c = \mathbf{1}$. For each arc $a = (u,v) \in A$ we define $\alpha_a := d_1(s,u) + 1 + d_1(v,t)$. (In many cases this is the length of a shortest (s,t)-path containing arc a.) We define the perturbation $\epsilon_a := M^{\alpha_a - n}$, where $M := |A| + 1$ and $n = |V|$. The perturbated arc lengths are defined as $c_a^\epsilon := c_a + \epsilon_a$. Note that $\epsilon_a < 1$ for all $a \in A$, because no (s,t)-path contains more than $n-1$ arcs. Even more, we have $\sum_{a \in A} \epsilon_a < 1$. It follows, that for each collection Q of vertex-disjoint (s,t)-paths we have $\sum_{p \in Q} c^\epsilon(p) < \sum_{p \in Q} c(p) + 1$. Let Q and Q' be two collections of vertex-disjoint (s,t)-paths with $\sum_{p \in Q} c(p) = \sum_{p \in Q'} c(p)$. Suppose there is a length l, $1 \le l \le n-1$, such that there is a path $p' \in Q'$ containing an arc a' with $\alpha_{a'} \ge l$, but for all arcs a contained in the paths in Q we have $\alpha_a < l$. Then, due to the exponential perturbation, we have $\sum_{p \in Q} c^\epsilon(p) < \sum_{p \in Q'} c^\epsilon(p)$. In other words, this perturbation method does not change the length order of path collections with different total lengths, but prefers collections where the maximal number of arcs contained in a path is small among the collections of the same total length.

It is easy to see, that in the example in Figure 4.8 for each arc $a \in A$ the



(a) Collection Q



(b) Collection Q'

Figure 4.8: Perturbation of arc lengths: α's and two collections Q, Q'

value α_a is exactly the length of the shortest path containing this arc. If $c_a = 1$ for all arcs $a \in A$, then we have $\sum_{p \in Q} c^\epsilon(p) < \sum_{p \in Q'} c^\epsilon(p)$.

It is clear, that the α's can be computed very fast by building the shortest path tree rooted in s and the reverse shortest path tree "rooted" in t.

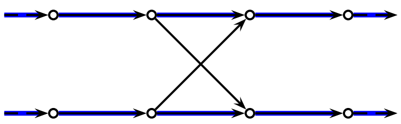
The second perturbation method works similar. It is used for the weighted problem WSP before applying Suurballe's heuristic. In this case, given two collections of vertex-disjoint (s,t)-paths with the same total number of arcs, we want to prefer the collection with the smaller total costs. Again, if the numbers of arcs contained in the two path collections are different, the order of the collections with respect to the arc lengths should not change.

Let $C := \sum_{a \in A} c_a + 1$. For each arc $a \in A$ define $\epsilon_a := c_a / C$. Obviously, we have $\sum_{a \in A} \epsilon_a < 1$. The new perturbated arc lengths are defined as $c_a^\epsilon := 1 + \epsilon_a$. It is easy to see, that for any collection Q of vertex-disjoint (s,t)-paths we have $c^\epsilon(Q) = \sum_{a \in Q} 1 + 1/C \cdot \sum_{a \in Q} c_a$. Consequently, this perturbation has the needed properties.

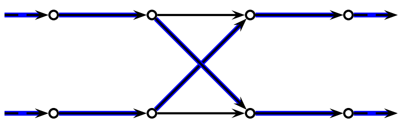
4.3.5 Postprocessing

After running the flow heuristics, we may end up with path collections that violate the length restriction or that have immense overall costs. With the postprocessing techniques described in this section, we try to improve the results of the flow heuristics to obtain feasible and/or better solutions.

The postprocessing methods can be splitted into two groups: methods we apply to reduce the maximal length of the paths in a collection and methods



(a) p_1 and p_2



(b) q_1 and q_2

Figure 4.9: Cross-over q_1,q_2 of p_1,p_2

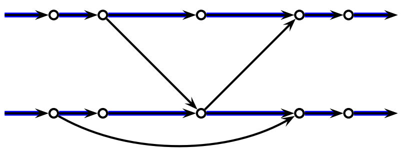
to reduce the overall costs of a feasible solution.

Let Q be a collection of vertex-disjoint (s,t)-paths computed by a flow heuristic. In our postprocessing algorithms we repeatedly apply local exchange methods. All exchange methods we present in this section modify only two paths in one iteration, w.l.o.g. these paths are $p_1, p_2 \in Q$. Let $p_1 = s, v_1, \ldots, t$ and $p_2 = s, w_1, \ldots, t$. We use two basic techniques in the exchange methods—cross-overs and substitutions.

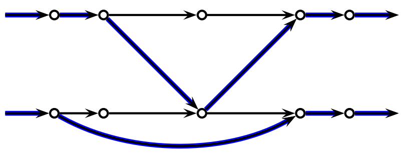
In the cross-over methods we compute all possible cross-overs of the two paths. A cross-over of p_1 and p_2 is a set of two other paths q_1 and q_2, such that $q_1 = s, v_1, \ldots, v_i, w_{j+1}, \ldots, t$ and $q_2 = s, w_1, \ldots, w_j, v_{i+1}, \ldots, t$ (see Figure 4.9).

If we use this method to reduce the maximal path length, we exchange the paths p_1 and p_2 in Q with the cross-over that minimizes the maximal path length. To obtain a feasible solution it only makes sense to apply this method to pairs of paths, that contain one long path (which violates the length restriction) and one short path. Otherwise the maximal path length of all cross-overs violates the length bound, too. If we apply this technique in the postprocessing for an instance of WSP, we take the costs of the cross-over as a second objective when selecting the cross-over we exchange with. If the cross-over method is used to reduce the costs of a feasible solution, i.e. all paths already satisfy the length restriction, we exchange the two paths p_1 and p_2 with a cross-over, that also satisfies the length restriction and that minimizes the costs among all those cross-overs and the original paths.

The substitution method works a little different. For any r with $1 \leq r \leq n-2$ an r-substitution of p_1 by p_2 is a set of two other vertex-disjoint paths q_1 and q_2, such that $q_1 = s, v_1, \ldots, v_i, w_h, \ldots, w_{h+r-1}, v_j, \ldots, t$—we substitute the subpath $v_i, \ldots, v_j$ of p_1 with a subpath $v_i, w_h, \ldots, w_{h+r-1}, v_j$ of p_2. In other words, we substitute a sequence of (arbitrary many) vertices in p_1 with a sequence of r vertices from p_2. The path q_2 just has to be vertex-disjoint to



(a) p_1 and p_2



(b) q_1 and q_2

Figure 4.10: 1-Substitution of p_1 by p_2 and resulting p_1,p_2

q_1 and to all paths in $Q \setminus \{p_1, p_2\}$. Such a path q_2 can be computed after the substitution, if it exists, using a shortest path algorithm in the digraph obtained from D by removing q_1 and all paths in $Q \setminus \{p_1, p_2\}$.

If we use this method to reduce the maximal path length, we only consider those substitutions, in which we substitute a subpath of r' vertices in a longest path of Q with a subpath of r vertices of some other path in Q with $r' > r$. If we find such a substitution, where both resulting paths q_1 and q_2 are shorter than the old longest path, we exchange the two original paths by the new paths. In case the substitution method to reduce the costs of a given feasible solution, we can substitute any subpath by any other subpath, as long as the length restriction is not violated by the resulting paths. We exchange the paths, if the new cost are smaller than the old ones. Although this exchange method is very expensive (in terms of time), it still is polynomially bounded. There are $O(k^2)$ pairings, $l-2$ different exchange lengths, and for each exchange length at most $O(n^3)$ possible substitutions.

4.4 Perl & Ronen's Heuristic

The heuristic presented in this section incorporates distance considerations into flow techniques. After each augmentation all constructed paths have to satisfy the length restriction. Originally, the algorithm is designed to solve SOP. Nevertheless, it also can be used to solve SFP.

Let $\phi = (D, d)$ be an instance of SOP, where $D = (V, A)$ is a digraph and $d = (s, t, k, l)$ is a directed demand in D. An optimal solution of SOP is a collection containing a maximum number of vertex-disjoint l-bounded paths.

The heuristic proposed by Perl and Ronen [PR84] contains two basic steps:

(1) find an initial solution Q for ϕ,

(2) repeated application of an augmentation algorithm for Q.

One basic advantage of this algorithm is that after every iteration we have a feasible solution. Thus, when solving the feasibility problem SFP, where we have to find "only" k vertex-disjoint l-bounded (s,t)-paths, we can stop the heuristic if this number of paths is reached.

There are many ways to find an initial solution. We could start with $Q = \emptyset$ or with any other solution. For example, we could use a collection of vertex-disjoint (s,t)-paths computed by one of the flow heuristics (see Section 4.3) and eliminate all paths that violate the length restriction to obtain an initial solution Q. Another way is to compute an initial solution, a so called "bounding solution", is presented in the following section. One advantage using this method is that problems with length bounds $l \leq 4$ are solved to optimality already in step (1) of the algorithm.

4.4.1 Creating Bounding Solutions

A solution is called an l-bounding solution if no l-bounded (s,t)-path which is vertex-disjoint to all paths in the solution exists in D. In other words, there is no l-bounded (s,t)-path in the digraph constructed by removing all arcs and vertices (except s and t) in the solution from D.

Such an l-bounding solution can be computed very easily. The algorithm proposed by Perl and Ronen creates a bounding solution by repeatedly applying a max-flow algorithm.

Let Q denote the bounding solution we are constructing and let $d_1(s,t)$ denote the distance (with respect to the arc length function $c = \mathbf{1}$) between s and t in D. For each $\bar{l} = 1, \ldots, l$ let $D_{\bar{l}}$ denote the subdigraph of D, that consists of those arcs and vertices that lie on an (s,t)-path of length at most $\bar{l}$. Obviously, $D_{\bar{l}} = \emptyset$ for all $\bar{l} < d_1(s,t)$. For each length level $\bar{l} = d_1(s,t), \ldots, l$ we do the following. First, we compute the maximal number of vertex-disjoint (s,t)-paths with length equal to l in D. This is done by applying a max-flow algorithm in D_l. Then these paths are added to our bounding solution and removed from the digraph D. It is clear, that after these steps there is no $\bar{l}$-bounded (s,t)-path in the remaining digraph. After the $\bar{l}$-th iteration the solution Q is $\bar{l}$-bounding. The complete algorithm is shown in Figure 4.11.

Bounding Solution

```
l̄ = d(s,t)                                              // start with distance
Q = ∅
repeat                                          // loop all different distance levels
   construct D_l̄ (f.e. bredth-first-search)
   max-flow in D_l̄: → collection Q_l̄ of vertex-disjoint l̄-bounded paths
   remove all arcs and vertices (except s and t) in Q_l̄ from D
   Q := Q ∪ Q_l̄
   l̄ := d(s,t)                                           // distance increases
until l̄ > l
```

Figure 4.11: Bounding Solution algorithm

Using this technique we can solve the problems with length bound $l \leq 4$ to optimality. This is easy to see. The steps in the (constructive) proof of Theorem 2.35 correspond 1-1 to the steps performed by the algorithm. The reductions in Lemma 2.36 are done implicitly: In the algorithm we start with a path length level of $\bar{l} = d_1(s,t)$ and increase it step by step after removing the $\bar{l}$-bounded paths found.

With the same technique we can solve the metric WSP for length bounds $l \leq 4$ to optimality. First we have to compute the set Q' of all 2-bounded (s,t)-paths. These paths are vertex-disjoint. Then we apply a min-cost-flow algorithm in the digraph D_l obtained from D by removing all arcs and vertices in Q' and all arcs and vertices, that do not lie on any path of length of at most l. We obtain the path collections $Q_1, Q_2, \ldots, Q_j$, such that each Q_i is the solution returned by the min-cost-flow algorithm after the i-th iteration, i.e., Q_i contains i many vertex-disjoint l-bounded (s,t)-paths in D_l with minimal costs. If we "fill" each Q_i with the $k-i$ cheapest paths from Q', we get a set of solutions including an optimal feasible solution, if there exists a feasible solution. The correctness of the algorithm follows directly from the proof of Theorem 2.25.

4.4.2 The Original Augmentation

Since the augmentation algorithm proposed by Perl and Ronen is very technical, we present only the main ideas of this algorithm. For a detailed description the reader is referred to [PR84].

Given a solution Q_k of ϕ with k vertex-disjoint l-bounded (s,t)-paths, the augmentation algorithm tries to compute a new solution Q_{k+1} containing $k+1$ such paths. A vertex $v \in V$ is called free if it is not contained in any path in Q_k. The main idea is to accept only such an interlacing path (see Definition 4.2), for which the paths created in the augmentation step are l-bounded. To find such an interlacing path, the algorithm applies a depth-first-search, starting from vertex s and searching for vertex t. Free vertices are preferred to nonfree ones in such a way, that the nonfree neighbors of a vertex are considered only after all free vertices have been considered. Each free vertex is labeled by its distance from s along the current search path. The algorithm is based on the assumption that if in the previous visit of vertex v the search path failed to reach t, then it will fail again, except the distance from s to v is less than it was in the previous try. Hence, a vertex is only visited by the depth-first-search, if its new distance from s is less than its previous distance from s. The depth-first-search is performed as long as the last vertex in the search path has a free neighbor, that is reachable within the length bound. Although the algorithm performs very well in practice, the assumption made above is wrong; it is only needed to bound the time spent in the depth first search. thus, the assumption guarantees that the time complexity of the algorithm is polynomially bounded.

Let $p = s, v_1, \ldots, v_i$ be the current search path. Assume all free reachable neighbors of v_i have already been considered, and there is a nonfree neighbor w of v_i. Let $q = s, w_1, \ldots, (w_j = w), \ldots, t$ be the (s,t)-path in Q_k, that contains w. If the distance from s to v_i in p plus the distance from w_j to t in q is less than l, then the algorithm performs a match between p and q. This match generates a new l-bounded (s,t)-path $q' = s, v_1, \ldots, v_i, w_j, \ldots, t$ and a new search path $p' = s, w_1, \ldots, w_{j-1}$. After the match the depth-first-search continues with the search path p'. Note that we have to backtrack the new search path p' in the depth-first-search. Perl and Ronen backtrack this path only until w_1 is reached. They do not "restart" at s after a match is done.

If a match was not successful, i.e., we could not find a solution with $k+1$ paths, we have to backtrack from this match. To do this, we have to keep the search state (vertex labels, paths in the current solution, etc.) before performing a match. To prevent an endless rematching among some paths, we also attach to each vertex a list of forbidden matches. When doing a match for the paths p and q, as explained above, we add two forbidden match identifiers to w_j's forbidden match list: one to prevent the match we actually do and one to prevent the rematch from the other side that could be done in the next step. A match is identified by the length of the search path that is matched and the two end vertices of the search paths before and

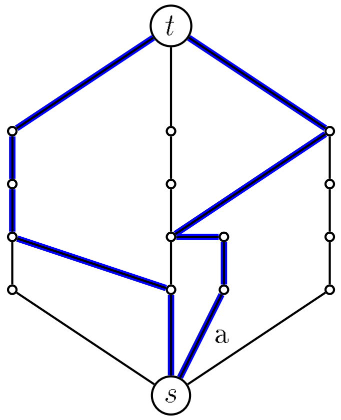


Figure 4.12: Solution by unmodified heuristic

after the match.

The time complexity of the augmentation algorithm can be determined by adding the complexity of the depth-first-search and the matching routines. Due to the assumption made for the search paths above, the depth-first-search will take at most $O(l|A|)$. Since there are at most $l|V||A|$ matches performed and each match needs $O(l)$ time to update the distances on the affected paths, the complete augmentation algorithms runs in $O(l^2|V||A|)$.

4.4.3 Modifications

There are some problems in the augmentation algorithm proposed by Perl and Ronen that can easily be fixed.

Let Q_k be a solution of ϕ and Q_{k+1} be the augmented solution created by the original augmentation algorithm. Let $S_k := \delta(s) \cap \{a \in A \,|\, a \in Q_k\}$, $T_k := \delta(t) \cap \{a \in A \,|\, a \in Q_k\}$, $S_{k+1} := \delta(s) \cap \{a \in A \,|\, a \in Q_{k+1}\}$ and $T_{k+1} := \delta(t) \cap \{a \in A \,|\, a \in Q_{k+1}\}$. Then we have $S_k \subset S_{k+1}$ and $T_k \subset T_{k+1}$. In other words, this algorithm never removes any outgoing arc of s or any incoming arc of t from a solution. This may block the augmentation procedure. An example with length bound is $l = 5$ is given in Figure 4.12. The algorithm cannot augment the given solution, because any solution that contains arc a has at most 2 paths.

We can use a simple preprocessing to fix this problem. Let k^* be at least the maximal number of vertex-disjoint l-bounded paths that may occur in our computation. (We could choose $k^* := n - 1$ to be on the safe side.) We

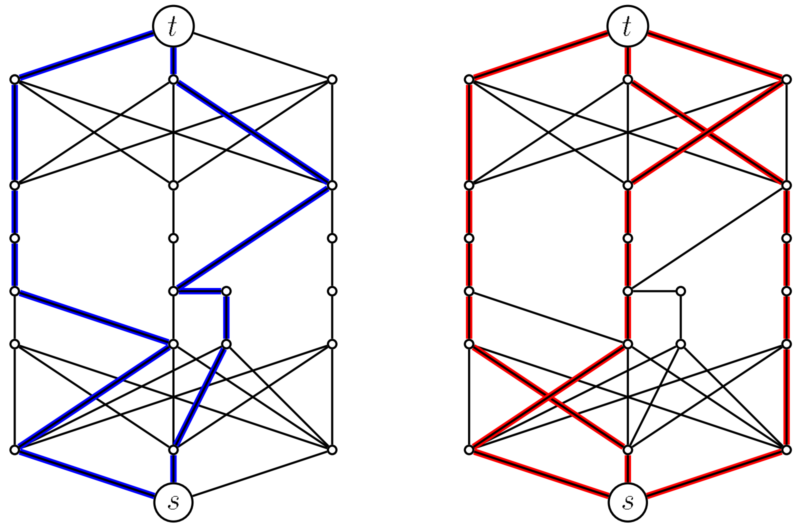


Figure 4.13: Modified graph and augmented solution

split the vertices s and t, such that there are k^* copies of s named $s_1, \ldots, s_{k^*}$ and k^* copies of t named $t_1, \ldots, t_{k^*}$. For each arc (s, v) we now have the arcs (s_i, v), and analogously for t. Then we add two new auxiliary vertices s and t and new auxiliary arcs (s, s_i) and (t_i, t) for $1 \leq i \leq k^*$. Finally, we increase the length bound l by 2, because all (s,t)-paths in the new digraph are exactly 2 arcs longer than before.

If we apply the original augmentation algorithm of Perl and Ronen to the modified digraph and demand, the outgoing arcs of s and the incoming arcs of t don't have to be removed from a solution to increase the number of paths. We now have these auxiliary vertices inbetween (see Figure 4.13).

But there is still another problem of the original approach that can be fixed, although not in such a nice way: The augmentation method is not symmetric. If we redirect the demand and, thus, change the direction of the depth-first-search, we may find a better solution, although we did not find one in the original direction. For an example see Figure 4.13 again. If we exchange s and t (reverse the demand), we find no augmentation of the given solution. The problem behind is that a certain match can be done only after another match is done. In our implementation we run the algorithm in both directions if necessary. Unfortunately, this does not eliminate the problem of the match order in general.

4.5 The Exact Enumeration Algorithms

In this section we present the last class of algorithms for the single-demand problems. As we have seen in the last sections, none of the polynomial algorithms can solve these problems exact or to optimality, except in some special cases. Thus we decided to implement exact algorithms for the single-demand problems. Although these algorithms are not polynomial, they are very fast in practice if we use certain speedup techniques.

The exact algorithms work in a branch-and-bound like way. In the following we describe the algorithm to solve SFP.

Let $\phi = (D, d)$ be an instance of SFP, where $D = (V, A)$ is a digraph and $d = (s, t, k, l)$ is a directed demand in D. We denote the current solution, that is a collection of vertex-disjoint l-bounded (s, t)-paths, by Q. The algorithm is a repeated depth-first-search strategy to find vertex t within the length bound of l. We start with $Q = \emptyset$. The vertices in the current search path of the depth-first-search and the vertices in Q are labeled, except vertex t. We call a vertex free if it is not labeled, i.e., it is neither contained in the current solution Q nor in the search path. Vertex t is always free. In the depth-first-search we consider only the free neighbors of the current vertex v. To speed up the algorithm we also label all these free neighbors of v before entering the next level of the depth-first-search. Thus, we avoid to roundabout a single arcs by a path. For an illustration see Figure 4.14, where the search path after 3 steps of the depth-first-search is shown. The filled vertices are labeled.

If the depth-first-search reaches vertex t within the length bound of l, the new (s, t)-path is added to Q. All vertices not contained in Q are unlabeled and the depth-first-search restarts in s. If the repeated depth-first-search fails to find the desired number of vertex-disjoint l-bounded paths, we have to backtrack from the last path selection. We remove the path from Q that was added last and continue at the point where we added this path. In order to use the neighbor labeling method described above effectively, we have to restore the old vertex label when backtracking from a path selection. Another "trick" to avoid time-wasting operations is to stop restarting the depth-first-search if there are only $k - |Q| - 1$ free vertices in the neighborhood of s left. Obviously, in this case the repeated depth-first-search cannot find $k - |Q|$ additional (s, t)-paths. If this algorithm finds a solution Q of k vertex-disjoint l-bounded paths at some point, we can stop, otherwise such a solution does not exist. The complete algorithm is shown in Figure 4.15.

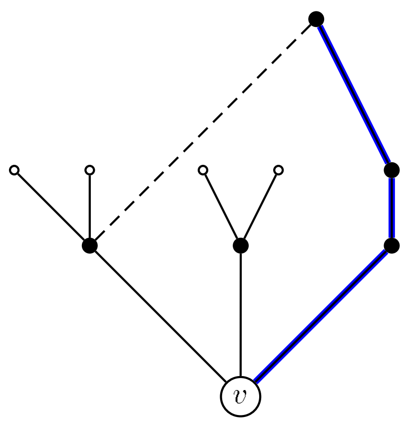


Figure 4.14: Avoiding roundabouts by neighbor labeling

The exact algorithms for SOP and WSP are similar. But, since these two problems are optimization problems we cannot stop when a feasible solution is found. We have to keep a best solution Q^*. The algorithms work branch-and-bound like, that is, they stop searching for a solution in a certain direction if they cannot find a better solution in this direction. The exact algorithm for WSP, for example, does the following: Let $p = s, \ldots, v$ denote the current search path. Before entering a vertex in the depth-first-search, we sum the total costs of the current solution $c(Q)$, the costs of the current search path $c(p)$, the costs of the shortest path (with respect to c) between v and t denoted by $d(v, t)$ and $k - |Q| - 1$ times the costs of a shortest (s, t)-path (with respect to c) denoted by $d(s, t)$. Obviously, this sum is a lower bound for the costs of any solution that might be found in this search-direction by the repeated depth-first-search. The distances $d(v, t)$ can be computed once before each depth-first-search or after each change of the vertex labels, to take these new labels into account. The order in which the neighbors of the current vertex v are considered is defined by these cost bounds. Unfortunately, we cannot use the neighbor labeling method described above to speed up the algorithm in the nonmetric case. In this case a roundabout of an arc may result in smaller costs. Nevertheless, in the metric case it can be used as in the unweighted case.

Similar techniques are used in the exact enumeration algorithm for SOP.

SFP-**Exact**

```
Q = ∅, i = 0
unlabel all vertices
p = s                                                          // search path
i = 0                                               // current depth in the dfs
recursive-search-paths(p, Q, i)

// recursive repeated depth-first-search: p: path, Q: solution, i: current length
function recursive-search-paths(p, Q, i)
  v := head(p)
  if l = i and v ≠ t then                                            // to far
    return no solution found
  if v = t then                                               // found new path
    Q := Q ∪ p, p = s
    unlabel all vertices not in Q
    if |Q| = k then
      exit have a solution: Q
    if |{free unconsidered neighbors of s}| < k − |Q| then
      exit infeasible
    recursive-search-paths(p, Q, 0)                           // restart dfs in s
    remove p from Q                                    // recursive search failed
    restore previous marking
    return no solution found
  else
    N := {free neighbors of v}
    label all vertices w ∈ N
    for all w ∈ N do
      append arc (v, w) to p
      recursive-search-paths(p, Q, i + 1)
    unlabel all vertices w ∈ N
  return no solution found
```

Figure 4.15: Exact algorithm for SFP

4.6 The Entire Algorithms

In this last section of the chapter we want to present the entire algorithmic frameworks for single-demand problems. These frameworks describe how the various heuristics work together to solve the single-demand problems.

Generally, we run the heuristics in the same order as they were explained in this chapter. First of all, the preprocessing steps are performed to reduce

SOP-**Framework**

if $l \leq 4$ **then**
  return BOUNDING SOLUTION
PREPROCESSING reduce D and $d \rightarrow\ Q_{pre}$
PERTURBATE arc lengths $1 \rightarrow c^{\epsilon}$
MIN-TOTAL-LENGTH-PATHS with lengths c^{ϵ}:$\rightarrow$ collection Q_f of maximal number k^u of vertex-disjoint (s,t)-paths
if all $p \in Q_f$ satisfy length restriction **then**
  return $Q_f \cup Q_{pre}$ // *we are optimal*
POSTPROCESSING $Q_f\ \rightarrow\ Q'_f$
if all $p \in Q'_f$ satisfy length restriction **then**
  return $Q'_f \cup Q_{pre}$ // *we are optimal*
BOUNDING SOLUTION and REPEATED AUGMENTATION $\rightarrow\ Q_{bp}$.
if $|Q_{bp}| = k^u$ **then**
  return $Q_{bp} \cup Q_{pre}$ // *we are optimal*
SOP-EXACT with lower bound $|Q_{bp}| \rightarrow\ Q_{ex}$
return $Q_{ex} \cup Q_{pre}$

Figure 4.16: SOP-Framework

the size of the graph and/or fix some paths. If the problem belongs to one of the polynomially solvable special cases, we run the appropriate exact algorithm. Otherwise we apply the flow heuristics using the perturbation techniques described in Section 4.3.4. These heuristics are very fast and we hope they create sufficiently good solutions. In practice, especially when used to solve the single-demand subproblems in the multi-demand framework, they do. If the solutions generated by the flow heuristics do not satisfy the length restriction or are not cost-minimal, we run the postprocessing methods described in Section 4.3.5. In the third step we create a bounding solution and apply the modified augmentation algorithm we described in Section 4.4.3 if necessary. Although this algorithm is designed for unweighted problems only, we also use it in the weighted case to obtain a feasible solution. The costs of this solution can be used as an upper bound in the next step. Fourth and last we run the appropriate exact enumeration algorithm. Of course, this is only done if no feasible or optimal solution was found in the previous steps. For the optimization problems SOP and WSP we use the bounds obtained from the previous solutions. In Figures 4.16 and 4.17 the complete frameworks for SOP and WSP are shown. The framework for SFP is the same as for SOP, except we run the SFP-versions of the heuristics and stop if any solution was found.

WSP-Framework

if $l \leq 4$ and metric **then**
  return BOUNDING SOLUTION
if $k = 1$ **then**
  return MIN-COST-LENGTH-RESTRICTED-PATH
PREPROCESSING reduce D, if metric also reduce d
PERTURBATE arc costs $c \to c^{\epsilon}$
MIN-TOTAL-LENGTH-PATHS with lengths c^{ϵ} and diversification $k \to$ collection Q_{f1} of k vertex-disjoint (s,t)-paths with minimal costs c^{l} or $Q_{f1} = \emptyset$ if not exists
if $Q_{f1} = \emptyset$ **then**
  return infeasible, there are no k paths
if all $p \in Q_{f1}$ satisfy length restriction **then**
  return $Q_{f1} \cup Q_{pre}$ *// we are optimal*
POSTPROCESSING $Q_{f1} \to Q'_{f1}$
if all $p \in Q'_{f1}$ satisfy length restriction **then**
  we have a "good" solution, set $c^u := min\{c^u, c(Q'_{f1})\}$
PERTURBATE arc lengths $1 \to c^{\epsilon}$ (using either costs or distances to s and t)
MIN-TOTAL-LENGTH-PATHS with lengths c^{ϵ} and diversification $k \to$ collection Q_{f2} of k vertex-disjoint (s,t)-paths with minimal total number of arcs
if all $p \in Q_{f2}$ satisfy length restriction **then**
  we have a "good" solution, set $c^u := min\{c^u, c(Q_{f2})\}$
POSTPROCESSING $Q_{f2} \to Q'_{f2}$
if all $p \in Q'_{f2}$ satisfy length restriction **then**
  we have a "good" solution, set $c^u := min\{c^u, c(Q'_{f2})\}$
BOUNDING SOLUTION and REPEATED AUGMENTATION $\to Q_{bp}$
if $|Q_{bp}| = k$ **then**
  we have a "good" solution, set $c^u := min\{c^u, c(Q_{bp})\}$
WSP-EXACT with upper bound $c^u \to Q_{ex}$
return Q_{ex}

Figure 4.17: WSP-Framework

Chapter 5

Algorithmic Approaches to the Multi-Demand Problems

In this chapter we will present the heuristics and frameworks we implemented for the multi-demand problems MFP and WMP. All of these algorithms are based on the techniques we developed for the single-demand problems in the previous chapter.

5.1 The Framework for MFP

As already mentioned in Section 1.2.1, the multi-demand feasibility problem MFP is decomposable into its associated single-demand feasibility problems. Each algorithm that solves MFP has to solve these single-demand problems somehow.

In our MFP–Framework we solve these single-demand problems simultaneously after some preprocessing tests are done. Let $\phi = (G, D)$ be the given instance of MFP, where $G = (V, E)$ is a (simple finite) graph and D is a set of demands in G. We define the minimal length bound $L := \min_{d \in D}\{l^d\}$ and the maximal diversification parameter $K := \max_{d \in D}\{k^d\}$. For real-world data the length bounds are equal for all associated single-demand problems, because they are based on the underlying hardware in the communication network. Often, the diversification parameter is given as a global survivability parameter, and hence it is the same for all of the associated single-demand problems, too.

```
MFP-Framework
if |E| ≥ binom(n-1, 2) + K or min_{v∈V}{|δ(v)|} ≥ ⌊(n + K)/2⌋ then
  return "true"
for all d ∈ D do
  PREPROCESSING for d
  FLOW HEURISTICS for d
if for any d "flow value" < k^d then
  return "false"
remove all satisfied demands from D
for all d ∈ D do
  BOUNDING SOLUTION and REPEATED AUGMENTATION for d
remove all satisfied demands from D
for all d ∈ D do
  EXACT ENUMERATION for d
remove all satisfied demands from D
if D = ∅ then
  return "true"
else
  return "false"
```

Figure 5.1: MFP–Framework

In the preprocessing we apply two tests based on the theorems in Section 3.2: If G has at least $\binom{n-1}{2} + K$ edges or a minimum degree of $\lfloor (n+K)/2 \rfloor$, then the solution of ϕ is "true" (see Theorem 3.7).

If none of these tests gives a positive answer, we have to solve each of the associated single-demand feasibility problems. For each of these problems we apply the SFP–Framework described in Section 4.6. We change the order of the computations made to reduce the overall processing time. First, we apply only the Preprocessing and the flow heuristics for each of the associated single-demand problems. We can stop without applying the time-consuming Repeated Augmentation heuristic or the SFP–Exact method if for at least one of the demands even the maximal number of paths (without a length bound) is less than the required diversification. In the second step we apply the Bounding Solution and Repeated Augmentation heuristics for each single-demand problem, for which we have no feasible solution yet. In the last step we apply the algorithm SFP–Exact for the demands that are still not satisfied. Solving the associated single-demand problems simultaneously in this way is much faster than solving them sequentially (if the given instance is not feasible), although we have to keep the current graph state (the preprocessed graph and the solutions already created) for each demand. For machines with only few RAM this memory vs. speed trade-off may propose a sequential processing of the demands.

If we have demands between all pairs of vertices in G, then we can apply two additional tests based on the Theorems 3.8 and 3.9 in the preprocessing. In this case we know the global connectivity of the graph after applying the flow heuristics for each demand. Since this case does not occur in practice, we omitted these tests in Figure 5.1, where the entire framework for MFP is shown.

5.2 The Algorithms for WMP

In this section we present a decrease heuristic framework, an exact algorithm based on this framework, and an increase heuristic framework for the weighted multi-demand problem. Remember that the weighted multi-demand problem is not decomposable into single demand problems.

Let $\phi = (G, D, c)$ be the given instance of WMP, where $G = (V, E)$ is a graph, D is a set of demands in G, and c is a nonnegative cost function on the edges of G. The aim is to select a subset $S \subseteq E$ at minimal cost, such that each demand $d \in D$ can be satisfied in the subgraph (V, S).

5.2.1 The Decrease Heuristic Framework and the Exact Algorithm

In the Decrease Heuristic Framework we first construct a feasible solution S (or start with a given feasible solution S). Then we try to remove edges from S. Various criteria for selecting the edge to remove are implemented. Furthermore, we have some strategies which of the single-demand algorithms have to be applied in which phase of the algorithm.

The general method works as follows. In the first step we solve all single-demand problems independently in G using the WSP–Framework described in Section 4.6. We can stop if one of the single-demand problems is not solvable—the multi-demand problem cannot be solvable, too. Analogously to the MFP–Framework, this processing is done simultaneously for all demands to avoid unnecessary operations. Our initial solution S consists of all edges in E that are contained in a solution of a single-demand problem.

Decrease Heuristic Framework

```
solve WSP for all d ∈ D simultaneously → Q^d
  if for at least one d ∈ D not solvable then
    return "not solvable"
S := {e ∈ E | e ∈ Q^d for some d ∈ D}
M = ∅                                              // unremovable edges
while S \ M ≠ ∅
  select edge e ∈ S \ M
  D' := {demands affected by the removal of e}
  for all d ∈ D' do
    p_i^d :=path in Q^d containing e
    AUGMENTATION of Q^d \ p_i^d in S \ e: → Q̄^d
    if augmentation failed then
      SINGLE-DEMAND FRAMEWORK for d in S \ e: → Q̄^d
    if |Q̄^d| < k^d then
      M := M ∪ {e}, break loop
    else
      Q^d := Q̄^d
  if e ∉ M then
    S := S \ {e}
return S
```

Figure 5.2: Decrease Heuristic Framework for WMP

In the second step we select a candidate edge $e \in S$ for removal and solve the associated multi-demand feasibility problem in $(V, S \setminus \{e\})$. We don't have to solve the complete single-demand feasibility problem for each demand. Only demands, whose current solution contains the edge e are "rerouted". If all demands can be satisfied in $(V, S \setminus \{e\})$ we remove edge e from S and continue, otherwise e is marked as "unremovable". The heuristic algorithm stops if no more edge can be removed from S. The complete Decrease Heuristic Framework is shown in Figure 5.2.

The exact method works analogously, except that we start with $S = E$ and we include a backtracking step. Every time we can remove an edge e from S, we first consider the case where e is removed from S (as in the Decrease Heuristic Framework), and then we consider the case where e is marked as "unremovable" (although it is removable). The sum of the edges marked as "unremovable" is used as a lower bound for the costs of the solutions we might find in the current direction in the search tree.

For each demand $d \in D$ we denote by Q^d the current solution of the single-demand problem associated with demand d, i.e., the collection of vertex-disjoint l^d-bounded (s^d, t^d)-paths in (V, S) used to satisfy d. For each edge $e \in E$ let $f(e)$ be the current "flow" on this edge, i.e., $f(e) := |\{d \in D \mid e \in Q^d\}|$. We say that demand d (or its routing Q^d) is affected by the removal of edge e if Q^d contains e.

In the following we describe the criteria for selecting the edge to remove:

1. **first edge:** We select the edge $e \in S$ with minimal index.
2. **maximal cost:** We select the edge $e \in S$ with $c(e) = \max\{c(e') \mid e' \in S\}$, that is the edge, whose removal decreases the overall costs at most.
3. **minimal flow:** We select the edge $e \in S$ with $f(e) = \min\{f(e') \mid e' \in S\}$, that is the edge, whose removal "destroys" the fewest routings.
4. **maximal cost flow ratio:** We select the $e \in S$ with $c(e)/f(e) = \max\{c(e')/f(e') \mid e' \in S\}$, that is the edge, that has the largest costs per demand routed through ratio.
5. **minimal max diversification affected:** We select the edge $e \in S$ with $max\{k^d \mid d : e \in Q^d\} = \min\{max\{k^d \mid d : e' \in Q^d\} \mid e' \in S\}$, that is the edge, for which the maximal diversification of the demands affected by the deletion of this edge is minimized. (We assume that demands with a larger diversification are harder to satisfy than demands with small diversification.)
6. **maximal min length affected:** We select the edge $e \in S$ with $min\{l^d \mid d : e \in Q^d\} = \max\{min\{l^d \mid d : e' \in Q^d\} \mid e' \in S\}$, that is the edge, for which the minimal length bound of the demands affected by the deletion of this edge is maximized. (We assume that demands with a lower length bound are harder to satisfy than demands with large length bound.)

Another parameter that affects the behavior of the algorithm is which of the single-demand algorithms are used to solve the associated single-demand problems in the rerouting step of our framework. In this step we solve the associated single-demand feasibility problem for all demands, whose current solution (routing) contains the edge we want to remove. We use the methods for the unweighted problem—the costs are considered when creating an initial solution and in the edge selection criteria.

In the following we describe how the rerouting problem is solved. It is easy to see, that for an affected demand d exactly one path p_i^d in Q^d is interrupted if edge e is removed from S. Hence, $Q^d \setminus \{p_i^d\}$ is a collection of $k^d - 1$ many vertex-disjoint l^d-bounded (s^d, t^d)-paths in $(V, S \setminus \{e\})$. We can apply the Augmentation heuristic described in Section 4.4 to get a new solution of k^d such paths in $(V, S \setminus \{e\})$. If this method fails to reroute the interrupted path, we may apply the complete SFP–Framework presented in Section 4.6 to construct a new solution in $(V, S \setminus \{e\})$ without using the remaining paths.

If we apply only some of the algorithms used in the single-demand framework in the rerouting step, we obtain several variants of this method. For example, we can use only the Preprocessing, the flow heuristics, and the Bounding Solution and Repeated Augmentation methods. It is obvious, that a solution computed using only these algorithms in the rerouting method is a feasible solution of the original problem. On the other hand, there may be better feasible solutions we cannot find with this method. But we can reduce the processing time not using the exact enumeration algorithm. Nevertheless, we implemented both versions of the rerouting, with and without using SFP–Exact. In the latter case we can choose to continue with the version using also SFP–Exact, if no more edges can be deleted according to the first rerouting test.

5.2.2 The Increase Heuristic Framework

In the Increase Heuristic Framework we start with the empty set $S := \emptyset$ (or with $S := \{e \in E \,|\, c_e = 0\}$) and add edges to S until all demands can be satisfied in the subgraph (V, S).

The algorithm consists of three phases: the connectivity phase, in which we create a subgraph S that satisfies the desired connectivity requirements without the given length bounds, the feasibility phase, in which we add edges to S until the given demands are satisfied (with length bounds), and the decrease heuristics phase, in which we apply one of the decrease heuristics presented in the previous section to eliminate unnecessary edges.

Let $K := \max_{d \in D}\{k^d\}$. For each k with $1 \leq k \leq K$ we define $D_k := \{d \in D \,|\, k^d \geq k\}$. In the connectivity phase we do K augmentation steps, such that after the k-th augmentation we have at least k vertex-disjoint (s^d, t^d)-paths in (V, S) for each demand $d \in D_k$.

The k-th augmentation step works as follows. First, for all edges $e \in S$ we set $c_e = 0$. Thus, in each step we take only the additional costs of the augmentation into account. Then we apply the Min-Cost-Paths algorithm for

Increase Heuristic Framework

$S := \{e \in E \mid c_e = 0\}$
$c'_e := c_e$ for all $e \in E$
for $k = 1$ **to** K **do** // *phase 1*
  for $d \in D_k$ **do**
    MIN-COST-PATHS for k and c' between s^d and $t^d \to Q^d$
    $S := S \cup \{e \in Q^d\}$
    $c'_e := 0$ for all $e \in Q^d$
for $d \in D$ **do** // *phase 2*
  BOUNDING SOLUTION and REPEATED AUGMENTATION for d in S
  if not feasible **then**
    EXACT ENUMERATION for d in G with costs $c' \to Q^d$
    $S := S \cup \{e \in Q^d\}$
    $c'_e := 0$ for all $e \in Q^d$
DECREASE HEURISTIC FRAMEWORK // *phase 3*

Figure 5.3: Increase Heuristic Framework for WMP

each $d \in D_k$ in the graph G (see Section 4.3.3). After applying Min-Cost-Paths for a demand, we immediately add the edges in the returned path collection by the algorithm to S and set their costs to zero. Since we use only flow techniques and since edges that are already in S cause no additional costs, edges not is S are considered only if the required number of paths cannot be obtained even by long paths. Hence, this technique tends to generate long paths. For demands, whose length bound is 4 or less, we use the Bounding Solution algorithm instead of the Min-Cost-Paths algorithm. Thus, the paths computed for these demands satisfy the length restriction and we don't have to consider these demands in the second phase of our algorithm.

In the second phase we add some more edges to S, such that finally all demands can be satisfied in (V, S) (within the given length bounds). We start with S as computed in the connectivity phase. For each $d \in D$ we first apply the Bounding Solution and Repeated Augmentation heuristics in (V, S) to determine whether d can be satisfied. If d cannot be satisfied in (V, S) we apply the SFP–Exact algorithm for d in G. We add all edges in the returned path collection to S and set their costs to zero.

In the last phase we apply a decrease heuristic with S as initial solution to remove edges that are no longer needed. The complete heuristic (when no demand d has a length bound $l^d \leq 4$) is shown in Figure 5.3.

Varying the order, in which the demands are considered in the augmentation phase and in the feasibility phase, one may obtain a number of heuristics. In our implementation we consider the demands in order of their length bounds. This way we hope to cope with the problem, that the paths found in the first phase are longer than the given length bound.

Appendix A

Computational Results

The algorithms described in Chapter 4 and in Chapter 5 are implemented in C++ using the standard template library. The code can be used with every compiler that compiles the ANSI/ISO standard c++ library. We compiled the code with GNU g++, version 2.7.2, for solaris 2.5. For the computations we used a "SUN Ultra 1".

In the first set of tables we want to present results of the Decrease Heuristic Framework and the Increase Heuristic Framework. We applied both methods to various real-world data of **e-plus** Mobilfunk GmbH. We defined the cost of an edge to be the euclidean distance between its ends. Due to a nondisclosure agreement, we rescaled the values of the solutions (by a constant factor).

Instances			
Name	Vertices	Edges	Demands
MSC96	11	34	24
MSC98	14	39	82
MSC2000	17	62	106
E-all	237	as stated	as stated

Since the graphs in MSC96, MSC98 and MSC2000 are quite "small" as instances for the single demand problems, we constructed "random network graphs" to test our single demand frameworks. The graphs used for the tests shown in the appendix have been constructed by taking the positions of all vertices in the **e-plus** network (E-all). The edges are placed in the graph with length-decreasing probability. Thus, these graphs are "real world telecommunication network like", meaning that they are locally high connected but globally low connected.

Decrease and Increase Heuristic Framework results									
k	l	*d1*	*d2*	*d3*	*d4*	*d5*	*d6*	*best i*	*opt*
MSC96									
2	3	–	–	–	–	–	–	–	–
2	5	38.91	31.94	33.07	31.89	38.91	38.91	30.42	25.34
2	∞	34.43	20.29	26.13	19.54	34.43	34.43	19.54	19.54
3	3	–	–	–	–	–	–	–	–
3	5	45.79	41.44	44.64	45.82	45.79	45.79	41.50	
3	∞	45.81	40.16	44.64	40.08	45.81	45.81	41.11	37.13
MSC98									
2	3	–	–	–	–	–	–	–	–
2	5	53.90	35.57	42.94	38.22	53.90	53.90	37.86	
2	∞	38.03	20.73	38.61	28.15	38.03	38.03	23.46	
3	3	–	–	–	–	–	–	–	–
3	5	57.57		50.35	47.60	57.57	57.57	51.91	
3	∞	54.70	40.09	49.74	39.77	54.70	54.70	39.54	
MSC2000									
2	3	–	–	–	–	–	–	–	–
2	5	70.02	46.5	58.47	47.21	70.02	70.02	44.10	
2	∞	50.27	23.04	49.84	31.66	50.27	50.27	25.97	
3	3	–	–	–	–	–	–	–	–
3	5	79.94	64.65	69.77	58.97	79.94	79.94	61.88	
3	∞	81.04	43.22	62.89	50.52	81.04	81.04	47.44	

In the following table we present some detailed information about the tests that were done with the Decrease Heuristic Framework and the Increase Heuristic Framework. For each of the used single demand algorithms the number of calls is shown.

Decrease and Increase Heuristic Framework results									
Heuristic	Costs	Time	Pre.	Suu.	MCP.	Post.	Bou.	Aug.	Ex.
MSC96, $l = 3$, $k = 2$									
d1	—	0:0	0	0	0	0	1	0	0
MSC96, $l = 5$, $k = 2$									
d1	38.92	0:0.34	110	65	24	0	0	86	0
d2	31.95	0:0.28	81	45	24	0	0	57	0
continued on next page									

continued from previous page

Heuristic	Costs	Time	Pre.	Suu.	MCP.	Post.	Bou.	Aug.	Ex.
d3	33.08	0:0.31	92	57	24	0	0	68	0
d4	31.89	0:0.29	85	49	24	0	0	61	0
d5	38.92	0:0.39	110	65	24	0	0	86	0
d6	38.92	0:0.37	110	65	24	0	0	86	0
i no d	39.54	0:0.15	6	6	54	12	30	24	6
i d1	31.39	0:0.35	62	53	54	12	30	80	6
i d2	30.42	0:0.38	68	60	54	12	30	86	6
i d3	31.26	0:0.34	51	48	54	12	30	69	6
i d4	30.42	0:0.35	59	54	54	12	30	77	6
i d5	31.39	0:0.36	62	53	54	12	30	80	6
i d6	31.39	0:0.36	62	53	54	12	30	80	6
e2	25.34	381:52	9.3e5	6.8e5	24	37	34	9.3e5	0
MSC96, $l = \infty$, $k = 2$									
d1	34.43	0:0.37	126	83	24	0	0	102	0
d2	20.29	0:0.47	139	105	24	0	0	115	0
d3	26.14	0:0.32	111	76	24	0	0	87	0
d4	19.54	0:0.53	140	108	24	0	0	116	0
d5	34.43	0:0.38	126	83	24	0	0	102	0
d6	34.43	0:0.37	126	83	24	0	0	102	0
i no d	19.54	0:0.05	0	0	48	0	0	0	0
i d1	19.54	0:0.16	11	11	48	0	0	11	0
i d2	19.54	0:0.17	11	11	48	0	0	11	0
i d3	19.54	0:0.17	11	11	48	0	0	11	0
i d4	19.54	0:0.18	11	11	48	0	0	11	0
i d5	19.54	0:0.17	11	11	48	0	0	11	0
i d6	19.54	0:0.17	11	11	48	0	0	11	0
e2	19.54	22:17	1.3e5	1.1e5	24	0	0	1.3e5	0
MSC96, $l = 3$, $k = 3$									
d1	-	0:0	0	0	0	0	1	0	0
MSC96, $l = 5$, $k = 3$									
d1	45.79	0:0.55	136	92	24	0	0	112	0
d2	41.44	0:0.49	120	87	24	0	0	96	0
d3	44.64	0:0.46	109	77	24	0	0	85	0
d4	45.82	0:0.43	108	71	24	2	2	84	0
d5	45.79	0:0.57	136	92	24	0	0	112	0
d6	45.79	0:0.55	136	92	24	0	0	112	0

continued on next page

continued from previous page

Heuristic	Costs	Time	Pre.	Suu.	MCP.	Post.	Bou.	Aug.	Ex.
i no d	54.09	0:0.15	2	2	74	4	16	14	2
i d1	41.50	0:0.52	91	89	74	4	16	103	2
i d2	41.50	0:0.50	81	76	74	4	16	93	2
i d3	41.50	0:0.45	64	62	74	4	16	76	2
i d4	41.50	0:0.48	75	71	74	4	16	87	2
i d5	41.50	0:0.55	91	89	74	4	16	103	2
i d6	41.50	0:0.55	91	89	74	4	16	103	2
MSC96, $l = \infty$, $k = 3$									
d1	45.81	0:0.52	140	96	24	0	0	116	0
d2	40.16	0:0.52	142	108	24	0	0	118	0
d3	44.64	0:0.43	116	84	24	0	0	92	0
d4	40.08	0:0.46	118	84	24	0	0	94	0
d5	45.81	0:0.52	140	96	24	0	0	116	0
d6	45.81	0:0.53	140	96	24	0	0	116	0
i no d	42.18	0:0.08	0	0	72	0	0	0	0
i d1	41.11	0:0.43	64	61	72	0	0	64	0
i d2	41.11	0:0.41	67	63	72	0	0	67	0
i d3	41.16	0:0.35	45	43	72	0	0	45	0
i d4	41.16	0:0.33	46	43	72	0	0	46	0
i d5	41.11	0:0.44	64	61	72	0	0	64	0
i d6	41.11	0:0.43	64	61	72	0	0	64	0
e2	37.13	355:50	1.9e6	1.7e6	24	0	0	1.9e6	0
MSC2000, $l = 3$, $k = 2$									
d1	-	0:0.01	0	0	0	0	2	0	0
MSC2000, $l = 5$, $k = 2$									
d1	70.03	0:3.17	586	342	106	0	0	480	0
d2	46.52	0:3.92	666	488	106	1	1	560	0
d3	58.47	0:2.63	476	290	106	0	0	370	0
d4	47.21	0:2.96	517	345	106	0	0	411	0
d5	70.02	0:3.2	586	342	106	0	0	480	0
d6	70.02	0:3.24	586	342	106	0	0	480	0
i no d	50.56	0:0.69	7	5	219	10	75	70	5
i d1	48.25	0:1.75	169	157	219	10	75	232	5
i d2	44.10	0:2.23	252	242	219	11	76	315	5
i d3	44.10	0:1.88	184	176	219	10	75	247	5
i d4	44.10	0:2.03	223	211	219	11	76	286	5

continued on next page

continued from previous page

Heuristic	Costs	Time	Pre.	Suu.	MCP.	Post.	Bou.	Aug.	Ex.
i d5	48.25	0:1.79	169	157	219	10	75	232	5
i d6	48.25	0:1.84	169	157	219	10	75	232	5
MSC2000, $l = \infty$, $k = 2$									
d1	50.27	0:4.09	786	548	106	0	0	680	0
d2	23.04	0:12.56	1213	1062	106	0	0	1107	0
d3	49.84	0:2.7	546	360	106	0	0	440	0
d4	31.66	0:3.81	695	529	106	0	0	589	0
d5	50.27	0:4.19	786	548	106	0	0	680	0
d6	50.27	0:4.15	786	548	106	0	0	680	0
i no d	32.22	0:0.32	0	0	212	0	0	0	0
i d1	25.97	0:2.3	241	241	212	0	0	241	0
i d2	25.97	0:3.42	334	333	212	0	0	334	0
i d3	25.97	0:2.29	233	232	212	0	0	233	0
i d4	25.97	0:3.31	333	332	212	0	0	333	0
i d5	25.97	0:2.38	241	241	212	0	0	241	0
i d6	25.97	0:2.37	241	241	212	0	0	241	0
MSC2000, $l = 3$, $k = 3$									
d1	-	0:0.01	0	0	0	0	2	0	0
MSC2000, $l = 5$, $k = 3$									
d1	79.94	0:5.17	792	598	106	3	3	686	0
d2	64.65	0:4.66	645	500	106	1	1	539	0
d3	69.77	0:3.95	587	420	106	0	0	481	0
d4	58.97	0:4.96	722	568	106	0	0	616	0
d5	79.94	0:5.24	792	598	106	3	3	686	0
d6	79.94	0:5.31	792	598	106	3	3	686	0
i no d	82.86	0:1.01	5	4	323	8	71	67	4
i d1	67.25	0:3.77	370	356	323	8	71	432	4
i d2	61.88	0:4.45	466	452	323	8	71	528	4
i d3	64.56	0:3.77	354	351	323	8	71	416	4
i d4	63.55	0:3.74	349	345	323	8	71	411	4
i d5	67.25	0:3.93	370	356	323	8	71	432	4
i d6	67.25	0:3.85	370	356	323	8	71	432	4
MSC2000, $l = \infty$, $k = 3$									
d1	81.04	0:4.41	753	559	106	0	0	647	0
d2	43.22	0:6.53	911	774	106	0	0	805	0
d3	62.89	0:3.77	629	461	106	0	0	523	0

continued on next page

continued from previous page

Heuristic	Costs	Time	Pre.	Suu.	MCP.	Post.	Bou.	Aug.	Ex.
d4	50.52	0:4.53	707	555	106	0	0	601	0
d5	81.04	0:4.59	753	559	106	0	0	647	0
d6	81.04	0:4.58	753	559	106	0	0	647	0
i no d	50.10	0:0.6	0	0	318	0	0	0	0
i d1	48.09	0:2.36	157	157	318	0	0	157	0
i d2	47.44	0:2.33	149	149	318	0	0	149	0
i d3	47.44	0:2.3	152	151	318	0	0	152	0
i d4	47.44	0:2.38	164	164	318	0	0	164	0
i d5	48.09	0:2.34	157	157	318	0	0	157	0
i d6	48.09	0:2.43	157	157	318	0	0	157	0

In the next table we want to present some results of our WSP-Framework. In the tests nearly all demands could be solved to optimality or the nonexistance of a feasible solution could be shown already in the min-cost-flow step. Although more than 10.000 tests were done, we present only a few results that show the typical behavior of the algorithm.

WSP-Framework results								
k	l	$\|Q_{f1}\|$	$c(Q_{f1})$	$c(Q'_{f1})$	$c(Q_{f2})$	$c(Q'_{f2})$	$c(Q_{bp})$	$c(Q)$
e-plus MSC—data								
all demands solved already in min-cost-flow								
E-all, 500 edges, random demands								
3	6	0	0	-	-	-	-	-
3	11	3	21.35	-	-	-	-	-
4	5	1	6.24	-	-	-	-	-
4	6	4	17.45	-	21.17	21.17	18.75	18.46
4	7	4	15.61	-	-	-	-	-
5	6	5	27.03	-	-	-	-	-
5	18	3	31.91	-	-	-	-	-
5	19	3	9.08	-	-	-	-	-
6	8	4	26.55	-	-	-	-	-
7	12	5	18.94	-	-	-	-	-
E-all, 600 edges, random demands								
2	6	1	9.77	-	-	-	-	-
3	7	3	23.60	-	24.30	24.30	30.52	24.30
4	6	4	23.64	-	25.40	25.40	27.40	23.84
5	8	2	13.68	-	-	-	-	-

continued on next page

continued from previous page

k	l	$\|Q_{f1}\|$	$c(Q_{f1})$	$c(Q'_{f1})$	$c(Q_{f2})$	$c(Q'_{f2})$	$c(Q_{bp})$	$c(Q)$
6	11	5	39.34	-	-	-	-	-
6	19	6	46.84	-	-	-	-	-
7	10	1	9.74	-	-	-	-	-
8	11	4	30.82	-	-	-	-	-
8	14	6	47.17	-	-	-	-	-
8	16	3	19.01	-	-	-	-	-
10	19	6	31.12	-	-	-	-	-
E-all, 800 edges, random demands								
2	20	2	6.91	-	-	-	-	-
3	18	3	25.15	-	-	-	-	-
4	7	4	20.25	-	-	-	-	-
5	6	5	37.78	-	40.40	40.40	54.43	38.28
6	10	2	11.96	-	-	-	-	-
6	15	6	52.16	-	-	-	-	-
8	7	4	33.39	-	-	-	-	-
8	18	4	30.24	-	-	-	-	-
9	14	7	33.24	-	-	-	-	-
9	15	4	34.34	-	-	-	-	-

In the next table some results of the SOP-Framework are shown. Again, we present only a few test results that show the typical behavior of the framework—almost every instance could be solved by the flow heuristics to optimality.

SOP-Framework results							
Time	$\|Q_{Pre}\|$	$\|Q_f\|$	$l_{max}(Q_f)$	$l_{max}(Q'_f)$	$\|Q_{Bou}\|$	$\|Q_{pb}\|$	k_{opt}
E-all, 400 edges, random demands with l=5							
0:0.04	0	0	0	0	0	0	0
0:0.04	0	2	5	0	0	0	2
0:0.06	0	1	3	0	0	0	1
0:0.08	1	6	5	0	0	0	6
0:0.06	1	4	5	0	0	0	4
E-all, 400 edges, random demands with l=7							
0:0.06	0	1	7	0	0	0	1
0:0.1	0	6	7	0	0	0	6
0:0.06	0	2	7	0	0	0	2
0:0.08	0	3	6	0	0	0	3

continued on next page

continued from previous page

Time	$\lvert Q_{Pre}\rvert$	$\lvert Q_f\rvert$	$l_{max}(Q_f)$	$l_{max}(Q'_f)$	$\lvert Q_{Bou}\rvert$	$\lvert Q_{pb}\rvert$	k_{opt}
0:0.06	0	3	6	0	0	0	3
E-all, 400 edges, random demands with l=10							
0:0.04	0	0	0	0	0	0	0
0:0.12	0	3	6	0	0	0	3
0:0.18	1	7	6	0	0	0	7
0:0.08	0	3	6	0	0	0	3
0:0.1	0	2	5	0	0	0	2
E-all, 600 edges, random demands with l=5							
0:0.98	0	7	7	7	0	0	6
0:0.08	0	2	4	0	0	0	2
0:0.08	0	3	5	0	0	0	3
0:0.08	0	2	5	0	0	0	2
0:0.1	0	4	5	0	0	0	4
E-all, 600 edges, random demands with l=7							
0:0.18	0	6	7	0	0	0	6
0:0.14	0	4	5	0	0	0	4
0:0.18	0	4	5	0	0	0	4
0:0.08	0	2	5	0	0	0	2
0:0.14	0	4	6	0	0	0	4
E-all, 600 edges, random demands with l=10							
0:0.36	1	9	7	0	0	0	9
0:0.16	0	3	6	0	0	0	3
0:0.14	0	1	5	0	0	0	1
0:0.28	0	6	6	0	0	0	6
0:0.32	1	7	5	0	0	0	7
E-all, 1000 edges, random demands with l=5							
0:0.22	0	8	4	0	0	0	8
0:0.2	0	9	5	0	0	0	9
0:0.14	0	3	4	0	0	0	3
0:0.2	0	8	5	0	0	0	8
0:0.24	0	9	4	0	0	0	9
E-all, 1000 edges, random demands with l=7							
0:0.42	0	6	5	0	0	0	6
0:0.26	0	4	6	0	0	0	4
0:0.66	0	10	5	0	0	0	10
0:0.48	0	6	5	0	0	0	6

continued on next page

continued from previous page							
Time	$\lvert Q_{Pre}\rvert$	$\lvert Q_f\rvert$	$l_{max}(Q_f)$	$l_{max}(Q'_f)$	$\lvert Q_{Bou}\rvert$	$\lvert Q_{pb}\rvert$	k_{opt}
0:0.54	0	7	5	0	0	0	7
E-all, 1000 edges, random demands with l=10							
0:0.42	2	10	4	0	0	0	10
0:0.4	0	6	5	0	0	0	6
0:0.56	0	7	5	0	0	0	7
0:0.98	0	12	5	0	0	0	12
0:0.6	0	5	5	0	0	0	5
E-all, 3000 edges, random demands with l=5							
0:2.44	8	34	4	0	0	0	34
0:1.5	0	13	4	0	0	0	13
0:1.86	0	17	4	0	0	0	17
0:2.64	1	25	4	0	0	0	25
0:1.5	5	21	3	0	0	0	21
E-all, 3000 edges, random demands with l=7							
0:2.06	7	23	4	0	0	0	23
0:1.64	3	14	4	0	0	0	14
0:2.48	0	16	4	0	0	0	16
0:3.6	0	24	4	0	0	0	24
0:2.46	3	21	4	0	0	0	21
E-all, 3000 edges, random demands with l=10							
0:1.72	1	13	4	0	0	0	13
0:4.72	1	28	4	0	0	0	28
0:2.48	0	14	4	0	0	0	14
0:2.3	4	19	4	0	0	0	19
0:2.88	0	18	4	0	0	0	18

Index

Bibliography

[AGW96] D. Alevras, M. Grötschel, and R. Wessäly, *A network dimensioning tool*, Tech. report, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1996.

[AW96a] D. Alevras and R. Wessäly, *Cost–efficient network synthesis from leased lines*, Tech. report, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1996.

[AW96b] D. Alevras and R. Wessäly, *Rerouting lost communication paths*, Tech. report, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1996.

[BC94] Daniel Bovet and Pierluigi Crescenzi, *Introduction to the theory of complexity*, Pretice Hall International, 1994.

[Dij59] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numer. Math. **1** (1959), 269–271.

[FJO+87] R. J. Faudree, M. S. Jacobson, E. T. Ordman, R. H. Shelp, and Z. Tuza, *Mengers theorem and short paths*, The Journal of Combinatorial Mathematics and Combinatorial Computing **2** (1987), 235–253.

[GMS95] M. Grötschel, C.L. Monma, and M. Stoer, Handbooks in Operations Research and Management Science, Network Models, ch. 10, Design of Survivable Networks, North-Holland, 1995.

[IPS82] A. Itai, Y. Perl, and Y. Shiloach, *The complexity of finding maximum disjoint paths with length constraints*, Networks **12** (1982), 277–286.

[LNLP78] L. Lovàsz, V. Neumann-Lara, and M. Plummer, *Mengerian theorems for paths of bounded length*, Periodica Mathematica Hungaria **9** (1978), 269–276.

[Men27] K. Menger, *Zur allgemeinen kurvertheorie*, Fund. Math. **10** (1927), 96–115.

[OM87] P. Orponen and H. Mannila, *On approximation preserving reductions: Complete problems and robust measures*, Tech. report, Department of Computer Science, University of Helsinki, 1987.

[Pap94] C. H. Papadimitriou, *Computational complexity*, Addison–Wesley, 1994.

[PR84] Y. Perl and D. Ronen, *Heuristics for finding a maximum number of disjoint bounded paths*, Networks **14** (1984), 531–544.

[PY88] C. H. Papadimitriou and M. Yannakakis, *Optimization, approximation and complexity classes*, Proc. 20th ACM Symp. on the theory of computing (1988), 229–234.

[ST95] D. B. Shmoys and È. Tardos, *Computational complexity*, Handbook of Combinatorics (R. L. Graham, M. Grötschel, and L. Lovàsz, eds.), vol. 2, North-Holland, 1995, pp. 1599–1645.

[Suu74] J. W. Suurballe, *Disjoint paths in a network*, Networks **4** (1974), 125–145.

[Whi32] H. Whitney, *Non-separable and planar graphs*, Trans. Amer. Math. Soc. **34** (1932), 339–362.