ε-Relaxation and Auction Algorithms for the Convex Cost Network Flow Problem

by

Lazaros C. Polymenakos

Professional Diploma in Electrical and Computer Engineering
National Technical University of Athens, Greece (1989)
S.M., Massachusetts Institute of Technology (1991)

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

© Massachusetts Institute of Technology 1995. All rights reserved.
$\varepsilon$–Relaxation and Auction Algorithms for the Convex Cost Network Flow Problem

by

Lazaros C. Polimenakos

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 1995, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

The problem considered is the convex cost network flow problem; we propose auction and $\varepsilon$–relaxation algorithms for its solution. These methods stem from the $\varepsilon$–relaxation and auction algorithms that have already been developed for the linear cost problem. The new methods operate by doing an approximate coordinate ascent, where neither the primal nor the dual cost necessarily improves in a particular iteration. What motivates these methods is their good performance for the linear programming case. Our analysis shows that the $\varepsilon$–relaxation and auction methods for the convex cost problem have polynomial complexity, making them attractive for practical applications. Computational experimentation verifies this claim.

Thesis Supervisor: Dimitri P. Bertsekas
Title: Professor of Electrical Engineering and Computer Science
Acknowledgments

I would first like to thank my thesis advisor, Dimitri P. Bertsekas, for his guidance and support; without his help anything resembling this work would not have been possible. I also thank the other thesis committee members, Robert Gallager and John Tsitsiklis for their many suggestions and advises. Special thanks are also due to professor Paul Tseng of the University of Washington, Seattle, for being an unofficial reader of this work.

On the personal side, I would like to thank my parents Christos and Chrysoula for their tremendous support and forbearance, my brother Thanasis for always believing in me and my grandmother Anastasia for her complete love and affection. I should also mention my friends and colleagues at LIDS along with my close friends outside the lab for the collaborations, for the stimulating discussions and for the pleasurable breaks from the everyday grind.

Finally, I would like to thank professor and director of LIDS Sanjoy Mitter for offering his help to me on a variety of occasions and professor Mitchell Trott for his friendly and humorous attitude towards lab life.
Dedication

This work is dedicated to my family: my parents Christos and Chrysooula for all their sacrifices and support throughout the years, my brother Thanasis for always believing in me and my grandmother Anastasia for her love and affection. It is especially dedicated to the memory of my grandfather Athanasios Papachristopoulos, a mathematician, historian and writer, a man of keen insight, my mentor.
Contents

1 Introduction ........................................... 7
   1.1 Problem Definition ................................. 7
   1.2 Outline and Overview .............................. 13

2 Convex Functions and Monotropic Programming .......... 15
   2.1 Definitions and Notation .......................... 15
   2.2 The Convex Cost Network Flow Problem ............ 18
      2.2.1 Assumptions and Notation .................... 19
      2.2.2 Elementary Vectors ........................... 22

3 Overview of Methods for the Convex Cost Problem .... 24
   3.1 The Fortified Descent Algorithm ................... 25
   3.2 The Relaxation Algorithm ......................... 27
   3.3 The Minimum Mean Cost Cycle Canceling Method ... 30
   3.4 The Tighten and Cancel Method ................... 33

4 The $\epsilon$-Relaxation Method ......................... 36
   4.1 $\epsilon$—Complementary Slackness ................ 36
   4.2 The $\epsilon$—Relaxation Algorithm ................ 40
   4.3 Complexity Analysis ................................ 53
   4.4 The Reverse and Combined $\epsilon$—Relaxation Algorithms 58
   4.5 Computational Results ............................. 66

5 The Generic Auction Algorithm .......................... 73
5.1 The Algorithm .................................................. 73

6 The Auction/Sequential Shortest Paths Algorithm 84
  6.1 The Algorithm ............................................... 84
  6.2 The Reverse and Combined ASSP Algorithm .............. 92
  6.3 Computational Results ..................................... 97

7 Conclusion ...................................................... 104
References .......................................................... 106
Chapter 1

Introduction

1.1 Problem Definition

The subject of this report is the development of a class of algorithms solving the minimum cost network flow problem for convex separable cost functions. The minimum cost network flow problem is a constrained optimization problem where a cost function is minimized subject to linear constraints which correspond to conservation of flow conditions. In particular, consider a directed graph $G$ with a set of nodes $\mathcal{N} = \{1, \ldots, N\}$ and a set of arcs $\mathcal{A} \subset \mathcal{N} \times \mathcal{N}$, with cardinality $|\mathcal{A}| = A$. Let $x_{ij}$ be the flow of the arc $(i,j) \in \mathcal{A}$. The minimum cost network flow problem is then defined as:

\[
\begin{align*}
\text{minimize} & \quad f(x) = \sum_{(i,j) \in \mathcal{A}} f_{ij}(x_{ij}) \\
\text{subject to} & \quad \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i
\end{align*}
\]

where each $f_{ij} : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ is a convex function. Further assumptions on the convex cost functions will be given in later sections.

Despite its imposing formal description, the problem that we consider arises from concepts of the everyday life. For example, a city street map provides a good vi-
ualization of a network. Each arc (street) of the network is associated with a cost function which specifies how much it costs to ship a certain amount of a goods along that arc. The cost may represent either time or actual monetary cost incurred by moving along the arc. This cost may depend on the length of the arc, its traffic conditions, restrictions of the size of vehicles allowed on it, etc. Evidently, more traffic on an arc incurs larger cost, and there may be an upperbound on the amount of traffic an arc can handle. Furthermore, the traffic on one arc affects the traffic on other arcs. Now, consider the problem faced by the people of any city around the world: how to commute in such a network so as to minimize the total delay experienced. A different problem would be to try to ship certain amounts of a single or a variety of products (eg. clothes) from their respective points of origin (factories), to geographically dispersed destinations (retail stores) through this network. Each origin has a certain capability of producing the product and each destination has a certain customer demand that must be met. Such problems are easily formulated as a minimum cost network flow problem with a linear or convex separable cost function. Similarly, many problems in urban and economic planning, scheduling, routing, and manufacturing, can be modeled as minimum cost network flow problems.

In addition to its practical importance, the minimum cost network flow problem is alluring from an algorithmic perspective. Many network optimization problems like the transportation problem, the assignment problem, and the shortest path problem, can be formulated as special cases of the minimum cost network flow problem. It provides, therefore, a solid setting for devising and testing new algorithms and theories which may then be applied to a variety of problems and to other areas of optimization. Many algorithms in linear programming and combinatorial optimization have their origins in ideas applied to the minimum cost network flow problem.

We adopt the algorithmic perspective in this thesis. In particular, we focus on the development of new algorithms for the minimum cost network flow problem with convex separable cost. These algorithms may then be applied to problems with special structure like the transportation problem. We expect that the convergence, and complexity analysis of our algorithms and their testing on minimum cost network flow.
problems will provide a comprehensive assessment of the efficiency and applicability of the algorithms to various real world problems.

In view of its practical and algorithmic importance, it should not be surprising that the minimum cost network flow problem has attracted a lot of attention in the optimization literature. In particular, the special case of a linear cost function has been analyzed by many researchers over the years and many efficient algorithms exist. Each of the developed algorithms exploits a specific property of the underlying combinatorial structure of the problem to reach the optimal solution. The associated bibliography runs in the thousands and a comprehensive survey is neither possible nor intended. We identify, however, some of the basic ideas shared by most algorithms. This will also facilitate the development and understanding of algorithms for the convex cost case.

There are three main classes of algorithms that solve the linear cost network flow problem: primal methods, dual methods and auction methods. A comprehensive analysis of these types of methods can be found in the textbooks [Ber91] and [AMO92]. In the sequel we will discuss briefly the key algorithmic ideas of representative methods from each category. The primal methods proceed by pushing flow around negative cost cycles, thus reducing the primal cost with every iteration. Such a procedure is often called canceling of the negative cost cycle. Different rules for selecting the negative cost cycles on which to push flow result in different algorithms with different complexity and practical behavior. For example, the network simplex method which is considered among the fastest in practice, cancels cycles with the minimum negative cost. The reader is referred to a number of references that analyze the simplex method and study its behavior: A first, specialized version of the simplex for network problems appeared in [Dan51] and was extended in [Dan63]. The generic negative cost cycle canceling idea was given in [Kle67] and the strongly feasible trees implementation in [Cun76]. A large number of empirical studies of various pivoting rules have been made over the years. The textbook [BJS90] discusses a variety of pivot rules under which the network simplex method has polynomial running time. Further references can be found in [BJS90] and [Ber91]. In contrast to the simplex
method, Weintraub’s method (originally proposed for the non-linear cost problem in [W74] and subsequently analyzed for the linear cost problem in [BT89]) pushes flow along those negative cost cycles which lead to maximum improvement of the primal cost. Finally, primal methods with very good complexity bounds are the Minimum Mean Cost Cycle Canceling method proposed in [GoT89], the Tighten and Cancel method introduced in [GoT89], and the Minimum Ratio Cycles Canceling method analyzed in [WZ91].

Dual methods proceed by performing price changes along directions of dual cost improvement. Different choices of the direction of dual cost improvement lead to different algorithms. Because of the close connection of the primal and the dual problem, many of the primal methods can be modified to work for the dual problem. For example, the dual simplex method was analyzed in [HeK77] and [JeB80]. Dual methods that come from their primal counterparts have not really become popular in practice. There are methods, however, developed specifically for the dual problem. Such a method is the relaxation method proposed by Bertsekas, initially for the assignment problem in [Ber81], and generalized later to the min-cost problem in [Ber82a] (see also [Ber91]). This method has been shown to have very good practical performance.

Another popular dual technique is the so-called primal-dual method. It differs from the relaxation in the choice of direction of dual cost improvement: The primal-dual method chooses directions that provide the maximal rate of dual cost improvement whereas the relaxation method computes directions with only a small number of non-zero elements. The flows obtained in the course of the algorithm are not primal feasible and the primal-dual algorithm proceeds toward primal feasibility. This algorithm is also known in the literature as the successive shortest path algorithm, [Je58], [I60]. A popular generalization of this algorithm is the out-of-kilter algorithm, analyzed in [FoF62], [Min60].

Finally, auction methods operate by changing prices along coordinate directions in a similar fashion as dual coordinate ascent methods but the coordinate step is perturbed by $\epsilon$. These methods do not necessarily improve the primal or the dual cost.
at a particular iteration. However, they are guaranteed to make progress eventually and terminate with an optimal solution. These methods were first introduced in the context of the assignment problem [Ber79] but have since been extended to a variety of linear network problems. Such methods for the network flow problem are the $\epsilon$-relaxation, the generic auction (proposed in [BC91]), and the auction/sequential shortest path (proposed by Bertsekas in [Ber92]) algorithms. The $\epsilon$-relaxation algorithm was first proposed in [Ber86a] and [Ber86b]. Various implementations improving its worst-case complexity can be found in [BE87], [BE88], [BeT89], [Gol87a], [CoT90]. The complexity analysis of $\epsilon$-scaling first appeared in [Gol87a].

A class of network flow problems are those involving cost functions that are not linear but are instead convex and separable. Such problems belong to the area of monotropic optimization. Monotropic (meaning turning in one direction) refers to the monotonicity of the derivatives of convex functions. Monotropic programming problems contain all optimization problems with a separable, convex cost function and linear constraints. Therefore, monotropic programming contains the linear programs and the network programs as special cases and is itself a special case of general convex programs. One reason that monotropic programs are interesting is that nonlinear cost functions do arise in practice. Even if we could approach such problems with a series of linear approximations, it would be difficult computationally to replace the given problem by a series of complicated linear programs with possibly a lot of additional variables and constraints.

From a conceptual standpoint, an analysis of the conditions and assumptions that govern the solution of a monotropic program may be difficult to formulate if the original problem is replaced by piecewise linear approximations. Furthermore, monotropic programming has a rich duality theory which encompasses and extends the duality theory of linear programs. Duality provides valuable insights for the structure of the problem and for the meaning of the conditions characterizing its solution. Finally, the underlying combinatorial structure (given by the linear constraints) is analogous to that for linear network programs (structures like trees, paths, and cuts). Many methods developed for monotropic programs exploit this combinatorial structure to
solve the problem efficiently. It is not surprising, therefore, that primal and dual methods developed for the linear minimum cost flow problem have been translated to similar methods for its convex cost counterpart. For an overview of the generalization of some primal and dual methods to the convex cost network flow problem, the reader is referred to [Roc84]. The relaxation method was extended to the strictly convex and the mixed convex cost problems in [BHT87], whereas the Minimum Mean Cost Cycle Canceling method, and the Tighten and Cancel method were extended in the recent work of [KMc93]. An interesting, recent result is that the convex cost network flow problems can be solved in polynomial time, as shown in [KMc93].

Finally, we note that there have been other approaches to dealing with the convex cost network flow problem. One such approach, is to use efficient ways to reduce the problem to an essentially linear problem by linearization of the arc cost functions; see [Mey79], [KaM84], [Roc84]. Another approach is to use differentiable unconstrained optimization methods, such as coordinate descent [BHT87], conjugate gradient [Ven91], or adaptations of other nonlinear programming methods [HaH93], [Hag92]. Finally there are methods solving this problem which are based on the theory of Lagrange multipliers and have been shown to be efficient for certain types of problems as well as easily parallelizable. Specifically, we refer to proximal point methods and the alternating direction methods which have been explored recently ([EcF94], [NZ93]) for their potential for parallel computation. The proximal minimization algorithm has been studied extensively through the years. It was introduced in [Mar70] and is intimately related to the method of multipliers. Reference [Ber82] treats the subject extensively and contains a large number of references on the subject. The alternating direction method was proposed in [GIM75] and [GaM76] and further developed in [Ga79]. Reference [BeT89] has an educational analysis on the subject and contains many related references. The reader is also referred to [Ec89], [EcB92] and the subsequent work, [Ec93], which explored the potential of the method for parallel computation.
1.2 Outline and Overview

Auction algorithms have not been extended previously to the convex cost network flow problem. One of the goals of this work is to propose such extensions, analyze them and place them in perspective with other methods. Extensive testing of the $\epsilon-$relaxation and the auction algorithms for a variety of linear cost problems have shown the efficiency of these methods. The reader is referred to [BE88] [Ber88], [BC89a], [BC89b]. These references contain comparisons with other popular algorithms. For brief but indicative comparisons the reader is also referred to the textbook [Ber91].

Based on the $\epsilon-$complementary slackness conditions formulated in [BHT87], we extend the linear programming algorithms of [BE88], [BC91] and [Ber91] to the convex cost network flow problem. In particular, we propose the $\epsilon-$relaxation algorithm for the convex cost network flow problem, we prove its validity, and we derive unexpectedly favorable complexity bounds. The theory we develop becomes the basis for a variety of methods that utilize the ideas of $\epsilon-$relaxation. In particular, we proceed to generalize the Generic Auction and the Auction/Sequential Shortest Path algorithms for the convex cost problem and we estimate their complexity bounds. All these methods share a common framework which can be cast into a general auction algorithmic model containing the auction and $\epsilon-$relaxation algorithms for the linear minimum-cost flow problem. Computational experimentation verified the intuition that these algorithms outperform existing algorithms for difficult problems.

This report is organized as follows: In Chapter 2 we review some basic notions from convex analysis that will be needed in the development of our algorithms. We also define formally the problem that we are solving, and put forward the assumptions that we will be making throughout the thesis. In Chapter 3 we make an overview of some primal and dual methods solving the convex cost network flow problem. The overview is restricted to methods that bear some similarity with the new methods that we are proposing and will provide a basis to make comparisons. In Chapter 4 we extend the notion of $\epsilon-$Complementary Slackness to the convex cost network flow problem and we present the first contribution of this thesis, the development the
\(\epsilon\)-relaxation algorithm. We derive a favorable complexity bound for the \(\epsilon\)-relaxation algorithm under some assumptions. We also address the practical question of the efficiency of the algorithm by providing computational results on a variety of test problems and making comparisons with existing algorithms. Based on our analysis of the \(\epsilon\)-relaxation algorithm, we develop in Chapter 5 a generic auction algorithm for the problem, which contains the \(\epsilon\)-relaxation algorithms as a special case along with other variations like the network auction. In Chapter 6 we develop a different algorithmic idea, the Auction/Sequential Shortest Path algorithm and we present computational results that assess the efficiency of the algorithm. Chapter 7 presents conclusions and some possible topics for further research.
Chapter 2

Convex Functions and Monotropic Programming

This chapter lays the theoretical foundations of this thesis, and has two purposes. First it gives an overview of some fundamental points of the theory of convex analysis and introduces the basic notation that we will be using throughout this report. Second, it formulates the convex cost network flow problem and puts forward the basic assumptions that we will be making in order to solve it.

2.1 Definitions and Notation

We review some basic notions of the theory of convex functions which will be needed in the development of our algorithms. For a more complete exposition on the theory of convex functions in the context of monotropic programming, the reader is referred to [Roc84] and [Roc70] which have been the main sources for the definitions and theorems we present in this section.

For a function $f : \mathbb{R}^m \to (-\infty, \infty]$, we define its effective domain as

$$\text{dom} f = \{ x \in \mathbb{R}^m \mid f(x) < \infty \}.$$
We say that $f$ is convex if and only if

$$f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y), \quad \forall \alpha \in (0, 1), \forall x, y \in \mathbb{R}^m,$$

and strictly convex if the above inequality is strict for $x \neq y$.

If a convex function is defined on an interval we can extend its definition to the whole real line by setting $f$ equal to $+\infty$ for points outside the interval. A convex function is proper if $\text{dom} f \neq \emptyset$, and

$$f(x) > -\infty, \quad \forall x \in \mathbb{R}^m.$$

A function is lower semicontinuous if for every $x \in \mathbb{R}^m$ we have

$$f(x) \leq \liminf_{i \to \infty} f(x^i),$$

for every sequence $x^1, x^2 \ldots$ in $\mathbb{R}^m$ such that $x^i$ converges to $x$. Similarly, a function is upper semicontinuous if for every $x \in \mathbb{R}^m$ we have

$$f(x) \geq \limsup_{i \to \infty} f(x^i),$$

for every sequence $x^1, x^2 \ldots$ in $\mathbb{R}^m$ such that $x^i$ converges to $x$. The significance of lower semicontinuity for a proper convex function has to do with the limiting behavior of the function at the endpoints of its effective domain. Lower semicontinuity ensures that the proper convex function is closed in the sense that if its value at a finite endpoint of its effective domain is finite, then it agrees with the natural limit value as the endpoint is approached from the interior of the effective domain. Figure 2.1.1 illustrates the concepts of lower and upper semicontinuity.

A convex function is subdifferentiable at $x \in \text{dom} f$ if there exists $d \in \mathbb{R}^m$ such that

$$f(x + y) - f(x) \geq d^Ty, \quad \forall y \in \mathbb{R}^m,$$

and $d$ is called the subgradient of $f$ at $x$. The set of all subgradients $d$ of $f$ at $x$ are
Figure 2.1.1: The graphs on the left shows a lower semicontinuous function $f : \mathbb{R} \rightarrow (-\infty, \infty]$. The graph on the right shows an upper semicontinuous convex function.

called the subdifferential of $f$ at $x$, which is denoted by $\partial f(x)$.

The conjugate function of a convex function $f$ is denoted by $g$ and is given by

$$ g(t) = \sup_x \{ t^T x - f(x) \}. $$

If $f$ is a proper lower semicontinuous convex function then $g$ is a proper lower semicontinuous convex function ([Roc84]). This property is especially useful in the development of dual algorithms for the convex cost network flow problems, since the primal and the dual problem are cast in the same framework.

Finally, for a proper convex function $f : \mathbb{R} \rightarrow (-\infty, \infty]$, we define its right derivative $f^+(x)$ and left derivative $f^-(x)$ at a point $x \in \text{dom} f$ as

$$ f^+(x) = \lim_{y \downarrow x} \frac{f(y) - f(x)}{y - x}, \quad f^-(x) = \lim_{y \uparrow x} \frac{f(x) - f(y)}{x - y}, $$

(see Figure 2.1.2).

The right and left derivatives of a convex function are non-decreasing functions. Furthermore, the right derivative is right continuous and upper semicontinuous at every interior point of the $\text{dom} f$; the left derivative is left continuous and lower semicontinuous at every interior of $\text{dom} f$. A proof for these properties can be found
Figure 2.1.2: The left and right derivatives of a convex function $f : \mathbb{R} \rightarrow (-\infty, \infty]$.

in [BeT89, pp. 643]. These properties will prove crucial in the development of the algorithms in later chapters. We also follow the convention that if $x > y$, $\forall y \in \text{dom } f$ then $f^+(x) = f^-(x) = \infty$ and if $x < y$, $\forall y \in \text{dom } f$ then $f^+(x) = f^-(x) = -\infty$. Then it follows that if $x \geq y$ then $f^-(y) \leq f^-(x)$, $f^-(y) \leq f^+(x)$, and $f^-(x) \leq f^+(x)$.

For $x \in \text{dom } f$ we also define the directional derivative of $f$ at $x$ in the direction $d$ by

$$f'(x; d) = \inf_{\lambda > 0} \frac{f(x + \lambda d) - f(x)}{\lambda}.$$

### 2.2 The Convex Cost Network Flow Problem

This section reviews some of the basic notions related to the problem that we address in this thesis. In particular, we state the convex cost network flow problem formally and present its dual. We also introduce the assumptions and some basic notation that we will be using throughout this report. Then we describe the notion of elementary vectors which is important for the development of algorithms. Sources for the material in this section were [Roc84], [Roc70] and [BT89].
2.2.1 Assumptions and Notation

We consider the network flow problem involving a directed graph $G$ with a set of nodes $\mathcal{N} = \{1, \ldots, N\}$ and a set of arcs $\mathcal{A} \subset \mathcal{N} \times \mathcal{N}$, with cardinality $|\mathcal{A}| = A$. Let $x_{ij}$ be the flow of the arc $(i, j) \in \mathcal{A}$. The problem that we consider is known in the literature as the optimal distribution problem [Roc84, p. 328] and is of the form:

$$\begin{align*}
\text{minimize} & \quad f(x) = \sum_{(i,j) \in \mathcal{A}} f_{ij}(x_{ij}) \\
\text{subject to} & \quad \sum_{\{j|(i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i) \in \mathcal{A}\}} x_{ji} = s_i
\end{align*}$$

where each $f_{ij} : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ is a proper lower semicontinuous convex function. We will denote the effective domain of each function $f_{ij}$ by

$$C_{ij} = \{\xi \in \mathbb{R} \mid f_{ij}(\xi) < \infty\}.$$ 

Since $f_{ij}$ is proper convex, we conclude that $C_{ij}$ is a nonempty interval, and we will denote its left and right endpoints by $l_{ij}$ and $u_{ij}$ respectively. We assume that the problem is feasible, i.e., there exists a flow vector $x$ satisfying the flow conservation constraints and its components $x_{ij}$ belong to $C_{ij}$, for all $(i, j) \in \mathcal{A}$. Clearly, the cost of such a flow vector is finite. Let us denote by $f_{ij}^+(\xi)$ and $f_{ij}^-(\xi)$ the left and right derivatives respectively of the function $f_{ij}$ at the point $\xi$. We make the following assumption:

**Assumption 1** For all $\xi \in C_{ij}$ we have $f_{ij}^+(\xi) > -\infty$ and $f_{ij}^-(\xi) < +\infty$, for all $(i, j) \in \mathcal{A}$.

Figure 2.2.1 illustrates the types of behavior exhibited at the end points of the effective domain that are avoided by imposing assumption 1. As a result, any feasible solution $x$ to our problem satisfies $f_{ij}^+(x) > -\infty$ and $f_{ij}^-(x) < +\infty$, for all $(i, j) \in \mathcal{A}$. Such a solution is called regularly feasible. Under our assumptions, all feasible solutions are regularly feasible; we conclude that if an optimal solution exists, it is
also regularly feasible. The importance of this assumption will become evident after the development of the dual problem.

The dual of the optimal distribution problem is the optimal differential problem which is defined in terms of node prices $p_i$, $i \in \mathcal{N}$, and in terms of the conjugate functions $g_{ij}$ of each $f_{ij}$. We will denote the effective domain of each function $g_{ij}$ by $D_{ij}$ and the left and right endpoints of $D_{ij}$ by $d_{ij}^-$ and $d_{ij}^+$ respectively. We define for each arc $(i,j) \in \mathcal{A}$ the price differential $t_{ij} = p_i - p_j$. The cost of the optimal differential problem (dual cost) $q(p)$ is then defined as

$$q(p) = -\sum_{i \in \mathcal{N}} p_is_i - \sum_{(i,j) \in \mathcal{A}} g_{ij}(t_{ij}).$$

The optimal differential problem is defined as the maximum of the dual cost with no constraints on the prices. Furthermore, the solutions of the optimal distribution problem and the optimal differential problem are closely related. This fact, together with the absence of constraints on the prices, makes the optimal differential problem appealing and motivates the algorithms that we will present. The following duality
result from convex analysis establishes the relationship of the solutions to the two problems:

**Proposition 2.2.1** Let $S$ be any subspace of $\mathbb{R}^n$, let $S^\perp$ be the orthogonal complement of $S$. Let $f$ be a proper, separable, convex and lower semicontinuous function and let $g$ be its conjugate. Consider the primal problem $\inf_{x \in S} f(x)$ and define its dual as $\inf_{t \in S^\perp} g(t)$. If the infimum of neither problem is $\infty$ then

$$\inf_{x \in S} f(x) = - \inf_{t \in S^\perp} g(t)$$

This proposition ensures that if the optimal solutions to the optimal differential and the optimal distribution problems are finite, then the solutions are equal.

A corollary of the above proposition is that if $x \in S$, $t \in S^\perp$ and $x \in \partial g(t)$ then $f(x) + g(t) = 0$. In the terminology of the linear programming theory, this says that if we have primal and dual feasibility and the complementary slackness conditions are satisfied then optimality is achieved. Complementary slackness in the context of the convex cost problem is equivalent to $x \in \partial g(t)$. We will use this result to prove convergence of our algorithms.

We also make the following assumption about the conjugate functions $g_{ij}$.

**Assumption 2** The conjugate convex function of each $f_{ij}$ defined by

$$g_{ij}(t_{ij}) = \sup_{x_{ij} \in C_{ij}} \{ t_{ij}x_{ij} - f_{ij}(x_{ij}) \} \quad (2.2.1)$$

is real valued, i.e., $-\infty < g_{ij}(t_{ij}) < +\infty$, for all $t_{ij} \in \mathbb{R}$.

We shall now discuss the necessity of the assumptions we have put forward. In particular, assumption (2) ensures the following properties: First, we have that for every $t_{ij}$ there exists a flow $x_{ij} \in C_{ij}$ attaining the supremum in (2.2.1) and

$$\lim_{|x_{ij}| \to +\infty} f_{ij}(x_{ij}) = +\infty.$$
Thus the cost function of the optimal distribution problem has bounded level sets and at least one optimal flow vector exists. This combined with the assumption made earlier that every solution to the optimal distribution problem is regularly feasible ensures the existence of an optimal solution for the optimal differential problem ([Roc84, p. 360]). Thus solving the optimal differential problem yields a solution for the optimal distribution problem. Furthermore, since the conjugate functions $g_{ij}$ are finite for all $t_{ij}$ we conclude that

$$\lim_{x_{ij} \to +\infty} f_{ij}^-(x_{ij}) = +\infty$$

and

$$\lim_{x_{ij} \to -\infty} f_{ij}^+(x_{ij}) = -\infty$$

(see [Roc84, p. 332]). This property will prove crucial in the development of algorithms for the convex cost network flow problem (especially in proving that the solution produced by the algorithm can be made arbitrarily close to the optimal solution). Finally, algorithms that operate by successive changes on the price vector will be very simple to initialize since for any choice of the initial price vector $p$ the dual cost will be finite.

### 2.2.2 Elementary Vectors

The primal and the dual methods solving the linear and convex cost problem try to find multi-coordinate cost improving directions. Intuition suggests that a small number of nonzero entries in the descent vector is desired for computational efficiency. This can be made precise by the notion of the *elementary vectors of a subspace*. We define the *signed support* of a vector $z \in \mathbb{R}^m$ as the union of the index sets $\{j \mid z_j > 0\}$ and $\{j \mid z_j < 0\}$, called the positive and negative support of $z$, respectively. The signed support of a vector $z'$ is contained in the signed support of a vector $z$ if and only if the positive support of $z'$ is contained in the positive support of $z$ and the negative support of $z'$ is contained in the negative support of $z$. It is *strictly contained*
if in addition the positive support of $z'$ is strictly contained in the positive support of $z$ or the negative support of $z'$ is contained in the negative support of $z$. Then an elementary vector of a subspace $S$ is defined as $z \in S$ such that $z \neq 0$ and there exists no other vector $z' \in S$ whose signed support is strictly contained in the signed support of $z$.

The importance of elementary vectors in the development of efficient algorithms for the network flow problem lies in two of their properties. One fundamental property of the elementary vectors is that, given any $x \in S$, it can be decomposed in a finite sum of elementary vectors, with their signed supports contained in the signed support of $x$ [Roc84]. A second property is that if our current solution of a monotropic programming problem is not optimal then there exists a direction of cost improvement which is an elementary vector [Roc84]. These properties are exploited in all algorithms that operate by successively improving the primal or the dual cost.

In the context of solving minimum cost network flow problems, the elementary vectors for the primal subspace are related to cycle flows, and the elementary vectors for the dual subspace are related to cut flows [Roc70]. Primal (dual) algorithms operate by finding successively cycles (cuts) with directional derivative that improves the primal (dual) cost and then a cost improving step is taken along that direction. In the following chapter we summarize a few well-known algorithms which operate in this fashion.
Chapter 3

Overview of Methods for the Convex Cost Problem

In this chapter we review a few algorithms which solve the convex cost network flow problem. A variety of methods exist and an extensive overview of these methods is beyond the scope of this presentation. The reader is referred to [Roc84] and [BeT89]. The methods that we review will become the basis for comparisons with the algorithms we develop. Furthermore, they are representative of various ideas utilized in the primal and dual algorithmic setting. We review four algorithms: The fortified descent algorithm [Roc84], the relaxation algorithm [BHT87], the minimum mean cost cycle canceling method [KMc93], and the tighten and cancel method [KMc93]. All these methods are related in the sense that they all relax in some way the complementary slackness conditions, and arrive at optimal solutions with costs within \( \varepsilon \) to the optimal. They differ, of course, significantly in the way these conditions are formulated and in the way the algorithms make progress toward the optimal solution. For some of the algorithms, we will give the results of complexity analysis as a measure of their efficiency. Since the cost functions are convex, it is not possible to express the size of the problem in terms of the problem data (e.g. flows and prices). A standard formulation that is amenable to complexity analysis techniques is to introduce an oracle that relates flows and costs. In particular, a standard oracle for the convex cost network flow problem would give us the following answers:
When given a flow $x_{ij}$, the oracle returns the cost $f_{ij}(x_{ij})$, the left derivative $f_{ij}^-(x_{ij})$ and the right derivative $f_{ij}^+(x_{ij})$.

When given a scalar $t_{ij}$, the oracle returns a flow $x_{ij}$ such that $f_{ij}(x_{ij}) \leq t_{ij} \leq f_{ij}^+(x_{ij})$.

Inquiries to such an oracle can be considered as simple operations. A complexity analysis of an algorithm for the convex cost network flow problem then becomes an estimate of the total number of simple operations performed by the algorithm.

### 3.1 The Fortified Descent Algorithm

One of the first algorithms to introduce the idea of finding a solution with cost that is within $\epsilon$ to the optimal was the fortified descent algorithm [Roc84]. For the optimal distribution problem the algorithm can be described as follows: We start with any feasible solution $x$ and try to solve the feasible differential problem (i.e., find a price vector such that each price is within its span interval) with span intervals

$$[d_{ij}^-(x), d_{ij}^+(x)] = [f_{ij}^-(x_{ij}), f_{ij}^+(x_{ij})],$$

where

$$f_{ij}^+ (x_{ij}) = \inf_{\zeta > x_{ij}} \frac{f_{ij}(\zeta) - f_{ij}(x_{ij}) + \delta}{\zeta - x_{ij}} \geq f_{ij}^+(x_{ij})$$

and

$$f_{ij}^- (x_{ij}) = \sup_{\zeta < x_{ij}} \frac{f_{ij}(\zeta) - f_{ij}(x_{ij}) + \delta}{\zeta - x_{ij}} \leq f_{ij}^-(x_{ij}).$$

Actually these are the slopes of the two lines depicted in the figure that follows (Figure 3.1.1).

If our current solution is not optimal then a cycle is obtained such that if flow is pushed, the cost decreases. The algorithm continues by finding descent directions along which the cost is improved if we augment the flow. For a complete description and proof of validity the reader is referred to [Roc84]. The cost of the solution differs
Figure 3.1.1: Illustration of the δ-bounds utilized by the fortified descent algorithm.

from the optimal by at most $\epsilon = \delta \cdot A$, where $A$ is the total number of arcs in the graph.

A similar algorithm can be applied to the dual problem. We start with a feasible solution $\pi$ and try to solve the feasible distribution problem (i.e., find a flow vector $x$ such that it satisfies the flow conservation equations and the capacity intervals) with capacity intervals defined by

$$[l_{ij}(x), u_{ij}(x)] = [g_{ij}^-(t_{ij}), g_{ij}^+(t_{ij})],$$

where

$$g_{ij}^+(x_{ij}) = \inf_{\zeta > t_{ij}} \frac{g_{ij}(\zeta) - g_{ij}(t_{ij}) + \delta}{\zeta - \xi} \geq g_{ij}^+(t_{ij})$$

$$g_{ij}^-(t_{ij}) = \sup_{\zeta < t_{ij}} \frac{f_{ij}(\zeta) - f_{ij}(t_{ij}) + \delta}{\zeta - t_{ij}} \leq g_{ij}^-(\zeta).$$

If our current solution is not optimal then a cut can be found and by changing the prices appropriately along the direction defined by the cut, a new price vector is obtained with improved cost.
3.2 The Relaxation Algorithm

The relaxation algorithm, introduced in [BHT87], also finds a direction of dual cost improvement and then takes a step along that direction which improves the dual cost as much as possible. It resembles the dual fortified descent algorithm since it utilizes bounds similar to the $\delta$-bounds and finds a solution with cost within $\epsilon$ of the optimal cost. The algorithm differs from the dual fortified descent because it attempts to find directions of dual cost improvement that are “cheap” in the sense that they contain a small number of non-zero elements. Such a direction can typically be computed quickly and often has only one nonzero element in which case only one node price coordinate is changed. Thus the algorithm often operates as a coordinate ascent algorithm.

It should be noted, however, that coordinate ascent directions cannot be used exclusively to improve the cost. In particular, an algorithm that tries to improve cost following coordinate directions exclusively, may terminate with the wrong answer. The reader is referred to [BeT89] where the issue is discussed extensively and a counter example is presented for a coordinate descent algorithm for the shortest path problem. Figure 3.2.2 shows a case where the surfaces of equal dual cost are piecewise linear and where any movement along a dual coordinate direction cannot improve the cost even though the current point is not the optimum. In order to ensure that the algorithm will solve the problem correctly, a multinode price change may be needed occasionally. To find such multinode directions of sufficient cost improvement and guarantee correct termination, the complementary slackness (CS) conditions that the flow vector $x$, and the price vector $p$ should satisfy are relaxed by $\epsilon$. We have:

$$f_{ij}(x_{ij}) - \epsilon \leq p_i - p_j \leq f_{ij}^+(x_{ij}) + \epsilon \quad \forall (i, j) \in A. \quad (3.2.1)$$

The above relation defines $\epsilon-$bounds on the flow vector; for each arc $(i, j) \in A$ the

---

1The need for multinode price increases in the context of the relaxation method was first discussed for the assignment problem in [Ber81]. Extension of the idea to the linear network flow problem is due to [Ber82a], and [BeT85].
The difficulty with following coordinate directions alone is illustrated here. At the indicated corner of the surfaces of equal dual cost, it is impossible to improve the dual cost by changing any single price coordinate.

The lower and the upper flow bound are respectively:

\[
I_{ij}^\varepsilon = \min \{ \xi \mid f_{ij}^+(\xi) \geq p_i - p_j - \varepsilon \}, \quad C_{ij}^\varepsilon = \max \{ \xi \mid f_{ij}^-(\xi) \leq p_i - p_j + \varepsilon \}. \tag{3.2.2}
\]

Thus \(\varepsilon\)-CS translates to:

\[
x_{ij} \in [I_{ij}^\varepsilon, C_{ij}^\varepsilon] \quad \forall (i, j) \in A. \tag{3.2.3}
\]

We observe that the \(\varepsilon\)-bounds of the relaxation method are similar to the \(\delta\)-bounds of the dual fortified descent method [Roc84], we discussed earlier. However, the \(\varepsilon\)-bounds are easier to calculate in practice.

Consider a set of nodes \(S\) and the sets of arcs

\[
[S, \mathcal{N} \setminus S] = \{(i, j) \mid i \in S, \ j \notin S\}, \quad [\mathcal{N} \setminus S, S] = \{(i, j) \mid j \in S, \ i \notin S\}.
\]
We define a vector \( u(S) \in \mathbb{R}^N \) as follows:

\[
 u_i(S) = \begin{cases} 
 -1 & \text{if } i \in S, \\
 0 & \text{if } i \notin S.
\end{cases}
\]

The directional derivative of the dual cost \( q \) at \( p \) in the direction of \( u(S) \) is defined as follows:

\[
 q'(p, u(S)) = \lim_{\Delta \to 0} \frac{q(p + \Delta u(S)) - q(p)}{\Delta} = \sum_{(i,j) \in \mathbb{N} \setminus S} c_{ij}^0 - \sum_{(i,j) \in [S, \mathbb{N} \setminus S]} l_{ij}^0 \tag{3.2.5}
\]

where \( c_{ij}^0, l_{ij}^0 \) are the \( \epsilon \)-bounds for \( \epsilon = 0 \). We refer to this directional derivative as \( -C_0(S, p) \). Consider now the price vector \( p' = p + \epsilon u(S) \). Then the directional derivative at \( p' \) along the direction \( u(S) \) is given by

\[
 q'(p', u(S)) = \lim_{\Delta \to 0} \frac{q(p' + \Delta u(S)) - q(p)}{\Delta} = \sum_{(i,j) \in \mathbb{N} \setminus S} c_{ij}^\epsilon - \sum_{(i,j) \in [S, \mathbb{N} \setminus S]} l_{ij}^\epsilon \tag{3.2.6}
\]

We refer to this directional derivative as \( -C_\epsilon(S, p) \). Since \( c_{ij}^0 \leq c_{ij}^\epsilon \) and \( l_{ij}^0 \geq l_{ij}^\epsilon \) we conclude that \( C_0(S, p) \geq C_\epsilon(S, p) \). Thus if for a set of nodes \( S \) we have that if \( C_\epsilon(S, p) > 0 \) then the direction \( u(S) \) is a direction of dual cost improvement. The relaxation iteration can be described as follows:

**Relaxation Iteration:**

**Step 1:** Use the labeling method of Ford-Fulkerson to find augmenting paths and for augmenting flow along them. If at some point the set \( S \) of labeled and scanned nodes satisfies \( C_\epsilon(S, p) > 0 \) then go to step 2.

**Step 2:** (Dual Descent Step). Determine \( \lambda^* \) such that

\[
 q(p + \lambda^* u(S)) = \min_{\lambda > 0} \{ q(p + \lambda u(S)) \}.
\]

Set \( p \leftarrow p + \lambda^* u(S) \) and update the \( \epsilon \)-bounds. Update \( x \) to be within the new bounds.
The analysis in [BHT87] shows that the stepsize in each dual ascent step is at least $\epsilon$ which is actually the main reason for the introduction of the $\epsilon$–CS conditions. Thus the $\epsilon$–CS conditions ensure that directions of sufficient cost improvement are chosen. This in turn ensures a lower bound on the amount of improvement of the dual cost per iteration and thus the algorithm terminates. Furthermore, the algorithm allows price changes on single nodes, thus interleaving coordinate ascent iterations with multinode price updates. The computational experiments reported in [BHT87] indicate that the algorithm is very efficient, especially, for ill-conditioned quadratic cost problems.

### 3.3 The Minimum Mean Cost Cycle Canceling Method

This method is an extension of the ideas in [GoT89] to the convex cost case. It was introduced in [KMc93], and is closely related to a more generic algorithm proposed by Weintraub in [W74], and later analyzed further for the linear cost case in [BT89].

It is based on the fact that if the flow vector $x$ is optimal, there exists no negative cost cycle at $x$, i.e., there exists no cycle such that an augmentation of flow along the cycle will decrease the cost. The intuitive description of the generic algorithm is as follows: Find a negative cost cycle and push flow around it so that the cost of the resulting flow $x'$ is less than that of $x$.

Weintraub’s and the Minimum Mean method differ in the choice of negative cost cycles to be canceled. The idea behind Weintraub’s algorithm is that if for a given flow vector $x$, we cancel the cycle that achieves the maximum decrease in the cost, then a polynomial upper bound on the amount of computation can be derived. However, finding such a cycle is an NP-complete problem. Instead, Weintraub’s algorithm finds a set of cycles to cancel such that the total decrease in cost is at least equal to the maximum possible decrease that canceling a single cycle can give. Such a set of cycles can be found relatively easily by solving an auxiliary linear bipartite matching problem.
The minimum mean cost cycle canceling method, chooses the cycle \( O = (F, B) \) (where \( F \) is the set of forward arcs and \( B \) the set of reverse arcs of the cycle) such that the cost per unit flow

\[
c(x, O) = \sum_{(i,j) \in F} f^+_i(x_{ij}) - \sum_{(i,j) \in B} f^-_i(x_{ij})
\]

is negative and the mean cost of \( O \),

\[
\bar{c}(x, O) = \frac{c(x, O)}{|O|}
\]

is as small as possible.

There are three key observations to be made:

- If \( O \) is a minimum mean cycle, then there exists a \( \delta > 0 \) such that if we set

\[
x'_{ij} = \begin{cases} 
  x_{ij} + \delta & \text{for } (i, j) \in F \\
  x_{ij} - \delta & \text{for } (i, j) \in B \\
  x_{ij} & \text{for } (i, j) \notin O 
\end{cases}
\]

then the total cost of the new flow vector \( x'_{ij} \) is strictly smaller than the cost of the flow vector \( x_{ij} \) (primal cost improvement). The quantity \( \delta \) can be computed as follows: Let \( \lambda = -\bar{c}(x, O) \). Choose \( \delta \) such that \( f^+_i(x_{ij} + \delta) < f^+_i(x_{ij}) + \lambda \) for all \( (i, j) \in F \) and \( f^-_i(x_{ij} - \delta) > f^-_i(x_{ij}) - \lambda \) for all \( (i, j) \in B \). Since the right and left derivatives are monotone, we have that

\[
f^{}_i(x_{ij} + \delta) < \delta f^+_i(x_{ij}) + \delta \lambda, \quad \forall (i, j) \in F
\]

\[
f^{}_i(x_{ij} - \delta) > \delta f^-_i(x_{ij}) - \delta \lambda, \quad \forall (i, j) \in B.
\]

Thus

\[
f(x') - f(x) < \delta c(x, O) + \delta |O| = 0.
\]
The absolute value of the minimum mean cost

$$\lambda(x) = \max\{0, - \min_{\text{cycles } \mathcal{O}} \bar{c}(x, \mathcal{O})\}$$

is the minimum possible value of $\epsilon$ such that there exists a price vector $p$ satisfying $\epsilon$-CS with $x$. Thus for the arcs in the minimum mean cycle $\mathcal{O}$, $\lambda(x)$-CS holds with equality, i.e., $p_i = p_j + f_{ij}^+(x_{ij}) + \lambda(x)$ for all $(i, j) \in F$, and $p_i = p_j - f_{ij}^-(x_{ij}) - \lambda(x)$ for all $(i, j) \in B$.

Canceling a minimum mean cycle does not increase the minimum mean cost of cycles of the resulting flow vector. Actually it is guaranteed that after at most $\psi = A - N + 1$ iterations $\lambda(x^{k+\psi}) \leq \left(1 - \frac{1}{2(N+1)}\right) \lambda(x^k)$.

Thus the algorithm would operate as follows: We create an auxiliary directed graph $\mathcal{D}(x)$ by introducing for each arc $(i, j) \in \mathcal{A}$ an arc $(i, j) \in \mathcal{D}(x)$ with cost $f_{ij}^+(x_{ij})$ and the reverse arc $(j, i) \in \mathcal{D}(x)$ with cost $f_{ij}^-(x_{ij})$. Then we find minimum mean directed cycles in $\mathcal{D}(x)$. It can be proved ([KMc93]) that this method requires $O(NA \log(\epsilon^0/\epsilon^t))$ iterations, where $\epsilon^0$ is the initial value of $\lambda(x)$ and $\epsilon^t$ is the value of $\lambda(x)$ upon termination. Each iteration needs about $O(NA)$ (see [K78]) time to find a minimum mean directed cycle. We note that although the complexity bound looks polynomial, it actually depends on the cost functions. In particular, given a feasible initial flow vector $\xi^0$ and and an initial price vector $p^0$, the value $\epsilon^0 = \lambda(x^0)$ depends on the cost functions. The authors refer to this complexity bound as polynomial but not strongly polynomial, pointing out the dependence on the cost functions.

It is interesting to see how the algorithm would operate for cuts instead of cycles, i.e., if we tried to solve the optimal differential problem ([KMc93]). We define an auxiliary graph $\mathcal{D}(x)$ with the same arcs as in $G$ and we define lower and upper flow bounds for each arc: $l_{ij} = g_{ij}(t_{ij})$ and $u_{ij} = g_{ij}^+(t_{ij})$, where $g_{ij}$ are the conjugate functions. Then a maximum mean cut $[S, N \setminus S]$ in $\mathcal{D}$ is the cut for which the quantity:

$$\left( \sum_{(i,j) \in [N \setminus S, S]} f_{ij}^-(x_{ij}) - \sum_{(i,j) \in [S, N \setminus S]} f_{ij}^+(x_{ij}) \right) / |\# \text{ arcs crossing } \mathcal{D}|$$

is maximized.
is maximized. Once we have such a cut, we compute a price increment such that

\[
g_{ij}^-(t_{ij} + \tau) \leq g_{ij}^+(t_{ij}) + \lambda = x_{ij} \quad \forall (i, j) \in [S, N \setminus S].
\]

\[
g_{ij}^+(t_{ij} - \tau) \geq g_{ij}^-(t_{ij}) - \lambda = x_{ij} \quad \forall (i, j) \in [N \setminus S, S].
\]

This algorithm has a lot of similarity with the relaxation algorithm and with the fortified descent algorithm of Rockafellar. The main differences lie in the selection of the cut and the lower and upper flow bounds employed. In particular, the cut chosen in the maximum mean cut algorithm is a cut for which \(\lambda - \text{CS}\) holds with equality for all the arcs involved. By contrast, the cut involved in the relaxation algorithm is any cut which leads to an ascent direction. For the relaxation algorithm to work, the flow bounds for the arcs have to be relaxed, since no effort is made to find a cut for which \(\epsilon - \text{CS}\) holds with equality. Sufficient descent, however, is guaranteed by the choice of the \(\epsilon - \text{perturbed}\) flow bounds.

### 3.4 The Tighten and Cancel Method

The tighten and cancel method [KMc93] is another primal method and is also based on a similar algorithm for the linear cost problem ([GoT89]). The algorithm operates in a similar fashion as the primal algorithms we have discussed so far: It finds and cancels negative cost cycles. The tighten and cancel method is based on an observation during the proof of the minimum mean method. Given a flow-price vector pair that satisfies \(\lambda - \text{Complementary slackness}\) (as defined above for the minimum mean cost cycle canceling method), we can define two types of arcs: Those which are Far (positively or negatively) from the tension \(t_{ij}\) and those which are Close (positively or negatively) to \(t_{ij}\). The equations satisfied by these categories of arcs are as follows:

Positively Far \(\iff\) \(f_{ij}^+(x_{ij}) + \lambda/2 < t_{ij} \leq f_{ij}^+(x_{ij}) + \lambda\) \hspace{1cm} (3.4.7)

Negatively Far \(\iff\) \(f_{ij}^-(x_{ij}) - \lambda \leq t_{ij} < f_{ij}^-(x_{ij}) - \lambda/2\) \hspace{1cm} (3.4.8)

Positively Close \(\iff\) \(t_{ij} \leq f_{ij}^+(x_{ij}) + \lambda/2\) \hspace{1cm} (3.4.9)
Positively Close  $\iff t_{ij} \geq f_{ij}^-(x_{ij}) - \lambda/2$  \hspace{1cm} (3.4.10)

Observe that no arc can be both positively and negatively far (because of the convexity of the cost functions $f_{ij}$). A key part of the proof of the complexity bound of the minimum mean cost cycle canceling algorithm ([KMc93]) is that once the minimum mean cost cycle contains an arc close to $t_{ij}$, then the value of $\lambda$ by which CS is violated, is decreased by a factor of $(1 - 1/2N)$. Thus the tighten and cancel algorithm ([KMc93]) does not attempt to cancel minimum mean cycles. Instead it cancels any cycle $O = (F, B)$ which consists of Positively (set $F$) and Negatively (set $B$) Far arcs. Every time a Far cycle is found, it is canceled, i.e., flow is incremented along the cycle by and amount $\delta > 0$ so that

$$f_{ij}^-(x_{ij} + \delta) \leq t_{ij} \leq f_{ij}^+(x_{ij} + \delta), \quad \forall (i, j) \in F.$$  \hspace{1cm} (3.4.11)

$$f_{ij}^-(x_{ij} - \delta) \leq t_{ij} \leq f_{ij}^+(x_{ij} - \delta), \quad \forall (i, j) \in B.$$  \hspace{1cm} (3.4.12)

In effect this operation brings at least one arc in $O$ close to $t_{ij}$. Such an arc will not be included in the subsequent cancel operations. Thus at most $O(A)$ of such cancellations can occur. Observe that each cancellation decreases the primal cost since

$$f_{ij}^+(x_{ij}) + \lambda/2 < t_{ij}, \quad \forall (i, j) \in F$$

$$f_{ij}^-(x_{ij}) - \lambda/2 > t_{ij}, \quad \forall (i, j) \in B.$$  \hspace{1cm} (3.4.11)

Since the sum of tensions along a cycle is zero it can be seen that

$$\sum_{(i,j) \in F} f_{ij}^+(x_{ij}) - \sum_{(i,j) \in B} f_{ij}^-(x_{ij}) < -|O|\lambda/2 < 0.$$  \hspace{1cm} (3.4.12)

Once all Far cycles have been canceled, the remaining graph involving Far arcs can be shown to be acyclic ([KMc93]). The tension of all these arcs can be modified as follows: Increase the tensions $t_{ij}$ of all Positively Far arcs $(i, j)$, by $k_{ij}\lambda/2(N + 1)$ ($1 \leq k_{ij} \leq N$, $\forall (i, j) \in P_{far}$) and decrease the tension of all negatively far arcs by $k_{ij}\lambda/2(N + 1)$ ($1 \leq k_{ij} \leq N$, $\forall (i, j) \in N_{far}$). The tensions of the positively
(negatively) close arcs can be increased (decreased) by any amount $k_{ij}\lambda/2(N+1)$ ($0 \leq k_{ij} \leq N$). Thus the value of $\lambda$ by which CS is relaxed, decreases by at least $(1 - 1/2(N+1))$. This technique ensures that the CS conditions are relaxed by a smaller amount.

One way to obtain the increments for the tensions of the arcs is the following: Consider the graph containing all Positively Far arcs and the reverses of all Negatively Far arcs so that Far cycles are directed cycles. After all Far cycles have been canceled, the final graph is acyclic, and an acyclic node labeling exists (i.e., a node labeling $L$ such that for each arc $(i,j)$, we have $L_i < L_j$). Define $\Delta L_{ij} = L_j - L_i$ and the tensions are updated as follows: $t_{ij} \leftarrow t_{ij} + \Delta L_{ij}\lambda/2(N+1)$. The complexity of such an algorithm is shown ([KMc93]) to be $O(NA\log(N)\log(e^0/e^t))$, where $e^0$ is the initial value of $\lambda(x)$ and $e^t$ is the value of $\lambda(x)$ upon termination.
Chapter 4

The $\epsilon$-Relaxation Method

This chapter lays the theoretical foundation for this thesis and contains our first contribution. In particular, we extend the $\epsilon$-relaxation algorithm [Ber86a], [Ber86b] to the convex cost network flow problem. We first develop the notion of $\epsilon$-Complementary Slackness ($\epsilon$-CS) as a relaxed form of the standard Complementary Slackness conditions for our problem. $\epsilon$-CS will give us freedom in the way we can perform changes on flow-price pairs. Thus a variety of methods can be developed sharing the same framework as $\epsilon$-relaxation, i.e. maintaining $\epsilon$-CS but differing in the way the flow-price pairs are updated. Such algorithms will be described in later chapters.

4.1 $\epsilon$-Complementary Slackness

We consider the convex cost network flow problem we defined earlier.

$$\begin{align*}
\text{minimize} & \quad f(x) = \sum_{(i,j) \in A} f_{ij}(x_{ij}) \\
\text{subject to} & \quad \sum_{\{j|(i,j) \in A\}} x_{ij} - \sum_{\{j|(j,i) \in A\}} x_{ji} = s_i
\end{align*}$$

where each $f_{ij}$ is a closed proper lower semicontinuous convex function. The Complementary Slackness (CS) conditions for this problem require that the flow vector and
the price vector satisfy

\[ f_{ij}^{-}(x_{ij}) \leq p_i - p_j \leq f_{ij}^{+}(x_{ij}), \quad \forall (i, j) \in \mathcal{A}. \]

In other words a flow-price pair satisfies the CS conditions if the vector of price differentials belongs to \( \partial f(x) \). For each arc \((i, j) \in \mathcal{A}\) the set of flow-price differential pairs satisfying the CS conditions define the characteristic curve associated with arc \((i, j)\),

\[ \Gamma_{ij} = \{(x_{ij}, t_{ij}) \in \mathbb{R}^2 \mid f_{ij}^{-}(x_{ij}) \leq t_{ij} \leq f_{ij}^{+}(x_{ij})\}. \]

The characteristic curve provides a convenient way to visualize the CS conditions and the way certain algorithms operate. Since there are no restrictions for the price vector, the standard duality result that we presented in the introduction takes the following form for our problem: If we have a flow-price vector pair \((x, p)\) such that \(x\) is a feasible solution to the optimal distribution problem and the pair \((x, p)\) satisfies the CS conditions, then \(x\) is a solution to the optimal distribution problem and \(p\) is a solution to the optimal differential problem. Therefore, a general approach to
solving the optimal differential problem would be to start from any flow-price vector pair satisfying the CS conditions and move toward optimality of the flow vector \( x \) by successively obtaining new flow-price vector pairs satisfying the CS conditions. Thus the CS conditions dictate the allowable changes we can perform on flows and prices. As shown in Figure 4.1.1 the characteristic curve can be quite complicated and following it as we strive for optimality could prove difficult. For this reason we introduce the \( \epsilon \)-CS conditions.

We define \( \epsilon \)-Complementary Slackness (\( \epsilon \)-CS) as a relaxed version of the CS conditions. In particular, the flow vector and the price vector satisfy \( \epsilon \)-CS if and only if

\[
f_{ij}^{-}(x_{ij}) - \epsilon \leq p_i - p_j \leq f_{ij}^{+}(x_{ij}) + \epsilon, \quad \forall (i, j) \in A.
\]

It is helpful to visualize the \( \epsilon \)-CS conditions. As shown in Figure 4.1.2, \( \epsilon \)-CS defines an \( \epsilon \)-cylinder around the characteristic curve. This is shown as a shaded region in the graph. All flow-price differential pairs in the shaded region satisfy the \( \epsilon \)-CS conditions. Intuition suggests that if we find a flow-price vector pair \((x, p)\) such that \( x \)
is a feasible solution to the optimal distribution problem, and the pair \((x, p)\) satisfies the \(\epsilon\)-CS conditions with \(\epsilon\) small, then \(x\) should be “close” to the optimal solution of the optimal distribution problem. This intuition, which will become precise in the analysis, motivates the algorithms we develop. Furthermore, the introduction of the \(\epsilon\)-CS conditions gives more freedom in the way we can move towards optimality as we discuss below.

Let us assume we are at a point on the characteristic curve. The \(\epsilon\)-CS conditions allow us to perform price rises and flow pushes without precisely following the characteristic curve. Starting from a point on the characteristic curve, we can follow any direction around that point and change the flow and price differential simultaneously until we either intersect the characteristic curve or the boundary of the \(\epsilon\)-cylinder defined by the \(\epsilon\)-CS conditions. The example in the figure 4.1.3 shows various ways

![Figure 4.1.3: Starting from any point on the characteristic curve (dark points) of arc \((i, j)\) a new point on the characteristic curve can be obtained in a variety of ways. The figure depicts a few such examples where the flow and price differential for arc \((i, j)\) are changed simultaneously according to some linear relation.](image)

that we can obtain a new flow-price differential pair on the characteristic curve if we start from a point on the characteristic curve and change the flows and price differ-
entials in a linear fashion. More complicated ways of changing the flow and price
differentials simultaneously may be used. Thus a variety of methods can be devised,
differing in the way we move in the allowable region defined by $\epsilon$–CS. The coordinate
directions are of interest to us primarily because we can change flows and prices
independently. The vertical direction changes the price differential without the need
to change the flow and the horizontal direction changes the flow without the need to
change the price differential. The methods that we develop in the next sections make
such independent flow and changes. In that respect they resemble the dual coordinate
ascent algorithms.

4.2 The $\epsilon$–Relaxation Algorithm

In this section we describe a new algorithm solving the convex cost network flow
problem based on the $\epsilon$–CS conditions introduced in the previous section. As in
section 4.1 we say that a flow-price vector pair satisfies the $\epsilon$–CS conditions if for all
$(i, j) \in A$ we have

$$p_i - p_j \in [f_{ij}^- (x_{ij}) - \epsilon, f_{ij}^+ (x_{ij}) + \epsilon].$$

Given a pair $(x, p)$ satisfying $\epsilon$–CS, we define for each node $i \in \mathcal{N}$ its push list as the
following set of arcs:

$$\left\{(i, j) \middle| \frac{\epsilon}{2} < p_i - p_j - f_{ij}^+(x_{ij}) \leq \epsilon\right\} \cup \left\{(j, i) \middle| -\epsilon \leq p_j - p_i - f_{ji}^-(x_{ji}) < -\frac{\epsilon}{2}\right\}.$$  

Figure 4.2.4 illustrates when an arc $(i, j)$ is in the push list of $i$ and when it is in the
push list of $j$. An arc $(i, j)$ (or $(j, i)$) of the push list of $i$ is said to be unblocked if
there exists a $\delta > 0$ such that

$$p_i - p_j \geq f_{ij}^+(x_{ij} + \delta),$$

(or $p_j - p_i \leq f_{ji}^-(x_{ji} - \delta)$, respectively). For an unblocked, push list arc, the supremum
of $\delta$ for which the above relation holds is called flow margin of the arc. The flow
Figure 4.2.4: A visualization of the conditions satisfied by a push list arc. The shaded area represents flow-price differential pairs corresponding to a push list arc.

margin of an arc $(i, j)$ and a $d$-flow push on $(i, j)$ is illustrated in Figure 4.2.5. An important property is the following:

**Proposition 4.2.1** The push list arcs of any node in $\mathcal{N}$ are unblocked.

**Proof:** Assume that for the arc $(i, j) \in \mathcal{A}$ we have

$$p_i - p_j < f_{ij}^+(x_{ij} + \delta),$$

for all $\delta > 0$. Since the function $f_{ij}^+$ is right continuous, we conclude that

$$p_i - p_j \leq \lim_{\delta \to 0} f_{ij}^+(x_{ij} + \delta)$$

which implies that

$$p_i - p_j \leq f_{ij}^+(x_{ij}).$$

Thus the arc $(i, j)$ cannot be in the push list of node $i$. A similar argument proves that an arc $(j, i)$ such that $p_j - p_i > f_{ji}^-(x_{ji} - \delta)$ for all $\delta > 0$, cannot belong to the push list of node $i$. We conclude, therefore, that the push list of any node $i \in \mathcal{N}$ contains only unblocked arcs. **Q.E.D.**
Thus, whenever we refer to an arc in the push list of some node, we know that the arc is unblocked. This observation will be of importance for the development of the algorithm. Let us now consider the set \( \mathcal{A}^* \) containing all push list arcs oriented in the direction of flow change. In particular, for each arc \((i, j)\) in the push list of a node \(i\) we introduce an arc \((i, j)\) in \(\mathcal{A}^*\) and for each arc \((j, i)\) of the push list of node \(i\) we introduce an arc \((i, j)\) in \(\mathcal{A}^*\). The set of nodes \(\mathcal{N}\) and the set \(\mathcal{A}^*\) define the \textit{admissible graph} \(G^* = (\mathcal{N}, \mathcal{A}^*)\). The notion of the admissible graph will prove useful in the analysis that follows.

We define the \textit{surplus} \(g_i\) of a node \(i\) as the net flow into \(i\):

\[
g_i = s_i + \sum_{(j | (j, i) \in \mathcal{A})} x_{ji} - \sum_{(j | (i, j) \in \mathcal{A})} x_{ij}.
\]

Clearly since the problem is feasible, the sum of the surpluses of all the nodes is zero. We will often refer to nodes with positive surplus as \textit{sources} and to nodes with negative surplus as \textit{sinks}. To solve the optimal differential problem we will start with a flow vector \(x\) that does not violate the capacity bounds of the arcs and a price vector \(p\) such that \((x, p)\) satisfy \(\epsilon\)-CS. We can perform two operations on a node \(i\) with positive surplus:
• A price rise on node $i$ increases the price $p_i$ by the maximum amount that maintains $\epsilon-CS$.

• A $\delta$-flow push along an arc $(i, j)$ (or along an arc $(j, i)$) consists of decreasing the surplus $g_i$ of node $i$ by an amount $\delta \leq g_i$ and increasing the flow of $(i, j)$ by $\delta$ (or decreasing the flow of $(j, i)$ by $\delta$, respectively). As a result the surplus $g_j$ of the opposite node $j$ increases by $\delta$.

We are now ready to describe a naive algorithm for the convex cost network flow problem. The algorithm successively obtains flow-price vector pairs that satisfy the $\epsilon$-CS conditions. A simple description is the following: We select a node $i$ with positive surplus and perform $\delta$-flow pushes along arcs of its push list. If no such arcs exist then we can perform a price rise. The algorithm will terminate once there are no nodes with positive surplus.

The above algorithm is naive in the sense that certain issues important to guarantee termination have been left out. We cannot hope for the algorithm to terminate unless we can guarantee that only a finite number of $\delta$-flow pushes can be performed. The naive algorithm, however, can lead to an infinite number of flow pushes because the choice of the flow $\delta$ in each $\delta$-push may be increasingly small, never leading to termination. Furthermore, if there are cycles in the admissible graph then examples can be devised where the number of $\delta$-pushes may be very large, leading to an inefficient algorithm. Such an example is found in [BT89, p. 378] for the case of linear arc costs. We repeat that example here in order to make our presentation complete.

In particular, consider the graph of Figure 4.2.6. All initial prices and all initial flows are set to zero. The feasible flow ranges and the costs of the arcs are shown in the figure. It is seen that for the particular choices of costs, initial prices and initial flows, $1$-CS is satisfied. However, there is a cycle in the admissible graph and the algorithm will push one unit of flow many times around that cycle leading to a large solution time. To avoid such situations we initialize the flow-price vector pair appropriately to guarantee the acyclicity of the admissible graph. We also define the way $\delta$-flow pushes are performed so that termination can be guaranteed. The resulting algorithm
Feasible flow range = [0, 1]
Cost = 2

Feasible flow range = [0,R]
Cost = -1

Feasible flow range = [0,R]
Cost = -1

Feasible flow range = [0,1]
Cost = 2

Figure 4.2.6: An example showing the importance of ensuring acyclicity of the admissible graph. Initially we choose $x = 0$ and $p = 0$ which do satisfy 1-CS. However the admissible graph has a cycle, namely 2-3-2. The algorithm will push one unit of flow $R$ times around the cycle 2-3-2 implying an $\Omega(R)$ solution time.

will be called $\epsilon-$relaxation and is described in detail below.

We begin our analysis by choosing the proper initialization for our algorithm. We set the flow-price vector pair so that the initial admissible graph $G^*$ is acyclic. In the analysis, we will show that, with this initialization, acyclicity of $G^*$ will be maintained throughout the algorithm. In particular, given an initial set of prices $\bar{p}$, we set the initial arc flow for every arc $((i, j)) \in A$ to

$$\bar{x}_{ij} = \sup \{ \xi \mid f_{ij}^+(\xi) \leq \bar{p}_i - \bar{p}_j - \frac{\epsilon}{2} \}.$$  

For this to be a valid initialization we must first make sure that $\epsilon$-CS is satisfied for every arc $(i, j) \in A$. As a result of the closedness assumption on the functions $f_{ij}$ and the definition of $\bar{x}_{ij}$ we conclude ([Roc84, p. 317]) that

$$f_{ij}(\bar{x}_{ij}) + \frac{\epsilon}{2} \leq \bar{p}_i - \bar{p}_j. \quad (4.2.1)$$
Thus the $\epsilon$–CS inequality $f_{ij}^+(\bar{x}_{ij}) - \epsilon \leq \bar{p}_i - \bar{p}_j$ is satisfied. We need to prove now that $\bar{p}_i - \bar{p}_j \leq f_{ij}^+(\bar{x}_{ij}) + \epsilon$. If $\bar{x}_{ij} = u_{ij}$ then $f_{ij}^+(\bar{x}_{ij}) = +\infty$ and the wanted inequality is satisfied trivially. If $\bar{x}_{ij} < u_{ij}$ then by the definition of $\bar{x}_{ij}$ and the fact that $f_{ij}^+$ is nondecreasing, we have that

$$f_{ij}^+(x_{ij}) > \bar{p}_i - \bar{p}_j + \frac{\epsilon}{2}, \quad \forall x_{ij} \geq \bar{x}_{ij}.$$ 

Since $f_{ij}^+$ is an upper-semicontinuous function, we conclude that

$$f_{ij}^+(\bar{x}_{ij}) \geq \lim_{x_{ij} \downarrow \bar{x}_{ij}} \sup_{x_{ij}} f_{ij}^+(x_{ij}) \geq \bar{p}_i - \bar{p}_j + \frac{\epsilon}{2}.$$  \hspace{1cm} (4.2.2)

Therefore, the $\epsilon$–CS inequality $\bar{p}_i - \bar{p}_j \leq f_{ij}^+(\bar{x}_{ij}) + \epsilon$, is also satisfied. Thus the initial flow-price vector pair satisfies the $\epsilon$–CS conditions. Furthermore, the above analysis (see Eq. (4.2.1), (4.2.2)) shows that the initial admissible graph is empty and thus acyclic. Based on this, we will later prove that the admissible graph remains acyclic throughout the algorithm and, as a consequence, the total number of flow pushes that can be performed by the algorithm is bounded.

We proceed now to describe the typical iteration of the $\epsilon$–relaxation algorithm. The algorithm will operate on nodes with positive surplus. We will often refer to this algorithm as the forward algorithm to differentiate it from an $\epsilon$–relaxation algorithm which operates on nodes of non-zero surplus that we will develop in a later section.

**Typical Iteration of the $\epsilon$–Relaxation Algorithm**

**Step 1:** Select a node $i$ with positive surplus; if no such node exists, terminate.

**Step 2:** If the push list of $i$ is empty then go to Step 3. Otherwise, pick an arc in the push list of $i$ and make a $\delta$–flow push towards the opposite node $j$, where

$$\delta = \min\{g_i, \text{flow margin of arc}\}.$$  

If the surplus of $i$ becomes zero, go to the next iteration; otherwise go to Step 2.
Step 3: Make a price rise at node i, i.e., increase the price \( p_i \) by the maximum amount that maintains the \( \epsilon \)-CS conditions. Proceed to a new iteration.

We make the following observations:

(1) The algorithm preserves \( \epsilon \)-CS and the prices are monotonically non-decreasing. This is evident from the initialization and Step 3 of the algorithm.

(2) Once the surplus of a node becomes nonnegative, it remains nonnegative for all subsequent iterations. The reason is that a flow push at a node \( i \) cannot make the surplus of \( i \) negative (cf. Step 2), and cannot decrease the surplus of neighboring nodes.

(3) If at some iteration a node has negative surplus, then its price must be equal to its initial price. This is a consequence of (2) above and the fact that price increases occur only on nodes with positive surplus.

To prove the termination of the algorithm we have to prove that the total number of price rises and \( \delta \)-flow pushes that the algorithm can perform is bounded. The following proposition shows that each price rise is bounded below by \( \frac{\epsilon}{2} \).

**Proposition 4.2.2** Each price increase is at least \( \frac{\epsilon}{2} \).

**Proof:** This can be seen as follows: First, we note that a price rise on a node \( i \) occurs only when its push list is empty. Thus for every arc \( (i,j) \in A \) we have that \( p_i - p_j \leq f_{ij}^+(x_{ij}) + \frac{\epsilon}{2} \), and for every arc \( (j,i) \in A \) we have \( p_j - p_i \geq f_{ij}^-(x_{ij}) - \frac{\epsilon}{2} \). We define the following set of positive numbers:

\[
S^+ = \{ p_j + f_{ij}^+(x_{ij}) + \epsilon - p_i : (i,j) \in A \} \\
S^- = \{ p_j - f_{ji}^-(x_{ji}) + \epsilon - p_i : (j,i) \in A \}
\]

Since the push list of \( i \) is empty we conclude that all the elements of \( S^+ \cup S^- \) are greater than \( \frac{\epsilon}{2} \). Then a price rise at \( i \) increases \( p_i \) by an increment \( \gamma = \min\{S^+ \cup S^-\} \geq \frac{\epsilon}{2} \).

Q.E.D.
The following lemma bounds the total number of price increases that the algorithm can perform on any node.

**Proposition 4.2.3 (Number of Price Increases).** Assume that for some $K \geq 1$, the initial price vector $p^0$ for the $\epsilon$-relaxation method satisfies $K\epsilon$-CS together with some feasible flow vector $x^0$. Then, the $\epsilon$-relaxation method performs at most $2(K+1)(N-1)$ price rises per node.

**Proof:** Consider the pair $(x, p)$ at the beginning of an $\epsilon$-relaxation iteration. Since the surplus vector $g = (g_1, \ldots, g_N)$ is not zero, and the flow vector $x^0$ is feasible, we conclude that for each source $s$ with $g_s > 0$ there exists a sink $t$ with $g_t < 0$ and a simple (i.e., having no cycles) path $H$ from $t$ to $s$ such that:

\begin{align*}
x_{ij} &> x_{ij}^0 \quad \forall \ (i, j) \in H^+ \quad (4.2.3) \\
x_{ij} &< x_{ij}^0 \quad \forall \ (i, j) \in H^- \quad (4.2.4)
\end{align*}

where $H^+$ is the set of forward arcs of $H$ and $H^-$ is the set of backward arcs of $H$.

(This can be seen from the Conformal Realization theorem ([Roc84] or [Ber91]) as follows. Consider the following problem: We replace the upper bounds on the flow by $u_{ij} - x_{ij}^0$, the lower bounds by $l_{ij} - x_{ij}^0$, the flow vector by $x^* = x - x^0$ and the $s_i$ by 0. The transformation leaves the surplus of every node unchanged and does not change the prices generated by the algorithm. Since $g \neq 0$, by the Conformal Realization Theorem, we can find a path $H$ from a sink $t$ to a source $s$ which conforms to the flow $x$, i.e., the flow on the path is positive for all $(i, j) \in H^+$ and the flow is negative for all $(i, j) \in H^-$. Thus equations (4.2.3) and (4.2.4) follow.)

From Eqs. (4.2.3) and (4.2.4), and the convexity of the functions $f_{ij}$, for all $(i, j) \in A$ we conclude that

\begin{align*}
f_{ij}^{-}(x_{ij}) &\geq f_{ij}^{+}(x_{ij}^0) \quad \forall \ (i, j) \in H^+ \\
f_{ij}^{+}(x_{ij}) &\leq f_{ij}^{-}(x_{ij}^0) \quad \forall \ (i, j) \in H^-
\end{align*}

(4.2.5)
Since the pair \((p, x)\) satisfies \(\epsilon\)-CS we conclude that

\[
p_i - p_j \in [f_{ij}^-(x_{ij}) - \epsilon, f_{ij}^+(x_{ij}) + \epsilon] \quad \forall (i, j) \in \mathcal{A}
\]  \hspace{1cm} (4.2.6)

Similarly, since the pair \((p^0, x^0)\) satisfies \(K\epsilon\)-CS we have

\[
p_i^0 - p_j^0 \in [f_{ij}^-(x_{ij}^0) - K\epsilon, f_{ij}^+(x_{ij}^0) + K\epsilon] \quad \forall (i, j) \in \mathcal{A}
\]  \hspace{1cm} (4.2.7)

Combining equations (4.2.5), (4.2.6), (4.2.7), we derive the following relations:

\[
p_i - p_j \geq p_i^0 - p_j^0 - (K + 1)\epsilon \quad \forall (i, j) \in H^+
\]  \hspace{1cm} (4.2.8)

\[
p_i - p_j \leq p_i^0 - p_j^0 + (K + 1)\epsilon \quad \forall (i, j) \in H^-
\]  \hspace{1cm} (4.2.9)

Summing the above equations over all arcs of the path \(H\) we get

\[
p_t - p_s \geq p_t^0 - p_s^0 - (K + 1)|H|\epsilon,
\]  \hspace{1cm} (4.2.10)

where \(|H|\) denotes the number of arcs of the path \(H\). We observed earlier that if a node is a sink at some time, then its price has remained unchanged since the beginning of the algorithm. Since the path is simple we have that \(|H| \leq N - 1\). Therefore, equation (4.2.10) yields

\[
p_s - p_s^0 \leq (K + 1)|H|\epsilon \leq (K + 1)(N - 1)\epsilon.
\]  \hspace{1cm} (4.2.11)

Since only nodes with positive surplus can increase their prices, and each price increase is at least \(\frac{\epsilon}{2}\) (due to the initialization of the price vector and the way prices rises occur) we conclude from equation (4.2.11) that the total number of price rises that can be performed for node \(s\) is at most \(2(K + 1)(N - 1)\). \textbf{Q.E.D.}

What is remarkable about this result is that the worst case performance of the algorithm is independent of the cost functions and our choice of \(\epsilon\). This result will be used later in this report to prove a particularly favorable complexity bound for the
algorithm.

Our next goal is to bound the number of flow pushes that can be performed between successive price increases. We first prove the acyclicity of the admissible graph.

**Proposition 4.2.4** The admissible graph remains acyclic throughout the algorithm.

**Proof:** The proof is similar to the one of [BE88] (see also [BeT89, p. 371]). We employ induction. Initially, the admissible graph $G^*$ is empty and the lemma is trivially satisfied. Assume that $G^*$ remains acyclic for all subsequent iterations up to the $m$–th iteration. We shall prove that after the $m$–th iteration $G^*$ remains acyclic. Clearly, after a flow push the admissible graph remains acyclic since it either remains unchanged, or some arcs are deleted from it. Thus we only have to prove that after a price rise at a node $i$, no cycle involving $i$ is created. We note that, after a price rise at node $i$, all incident arcs to $i$ in the admissible graph at the start of the $m$–th iteration are deleted and new arcs incident to $i$ are added. We claim that $i$ cannot have any incoming arcs which belong to the admissible graph. To see this note that just before a price rise at node $i$, we have

$$p_j - p_i - f^-(x_{ji}) \leq \epsilon, \quad \forall (j, i) \in A,$$

and since each price rise is at least $\epsilon/2$, we must have

$$p_j - p_i - f^-(x_{ji}) \leq \frac{\epsilon}{2}, \quad \forall (j, i) \in A,$$

after the price rise. Thus by Eq. (5), $(j, i)$ cannot be in the push list of node $j$. The case where $(j, i)$ belongs in the push list of $i$ is treated similarly, establishing the claim. Thus $(j, i)$ cannot be in $G^*$. Since $i$ cannot have any incoming incident arcs belonging to the admissible graph, no cycles are possible following a price rise at $i$.

Q.E.D.
We are now ready to bound the number of $\delta$–flow pushes that the algorithm can perform. Since the number of possible price increases is also bounded, the algorithm terminates. We first introduce some definitions that will facilitate the presentation. We say that a node $i$ is a predecessor of a node $j$ in the admissible graph if a directed path from $i$ to $j$ exists in $G^*$. Then node $j$ is a successor of $i$. Observe that flow is pushed towards the successors of a node and since $G^*$ is acyclic flow cannot be pushed from a node to any of its predecessors. A $\delta$–flow push along an arc is said to be saturating if $\delta$ is equal to the flow margin of the arc. A $\delta$–flow push is exhaustive if $\delta$ is equal to the surplus of the node. Observe that, as a result of Step 2 of the algorithm, a non-saturating $\delta$–flow push is necessarily exhaustive. Thus we have the proposition.

**Proposition 4.2.5** The number of $\delta$–flow pushes between successive price increases (not necessarily at the same node) performed by the algorithm is finite.

**Proof:** First we observe that flow changes cannot introduce new arcs in $G^*$. Furthermore, a saturating $\delta$–flow push along an arc removes the arc from the admissible graph. Thus the number of saturating $\delta$–flow pushes that can be performed is $O(A)$. It will thus suffice to show that the number of nonsaturating flow pushes that can be performed between saturating flow pushes is finite. Assume the contrary, that is, there is an infinite sequence of successive nonsaturating flow pushes with no intervening saturating flow push. Then, the surplus of some node $i^0$ must be exhausted infinitely often during this sequence. This can happen only if the surplus of some predecessor $i^1$ of $i^0$ is exhausted infinitely often during the sequence. Continuing in this manner we construct an infinite succession of predecessor nodes $\{i^k\}$. Thus some node in this sequence must be repeated, which is a contradiction since the admissible graph is acyclic. Q.E.D.

The above lemma completes the termination proof of the algorithm. Upon termination, we have that the flow-price vector pair satisfies $\epsilon$–CS and the flow vector is feasible. However, we have not yet obtained the best bound on the complexity of the algorithm. In Section 4.3 we will derive such a complexity bound for a particular
implementation of the algorithm.

Our next task is to estimate how close to optimal is the flow-price vector pair obtained upon termination of the $\epsilon-$relaxation algorithm. In particular, we show that the cost of the solution produced by the $\epsilon$-relaxation algorithm can be made arbitrarily close to the optimal cost by taking $\epsilon$ sufficiently small thus concluding the proof that our algorithm solves the optimal distribution problem. The proof is identical to the one in [BHT87]. We present two lemmas which are due to [BHT87].

**Proposition 4.2.6** Let $(x, p)$ satisfy $\epsilon-CS$ and $x$ be a feasible flow vector. Let $(\xi, p)$ satisfy CS $(\xi$ need not be a feasible vector). Then the primal cost $f(x)$ and the dual cost $g(p)$ satisfy the following relation

$$0 \leq f(x) + g(p) \leq \epsilon \sum_{(i,j) \in A} |x_{ij} - \xi_{ij}|.$$

**Proof:** Let $t_{ij} = p_i - p_j$ for all $(i, j) \in A$. Since the pair $(\xi, p)$ satisfies the CS conditions we have:

$$f_{ij}(\xi_{ij}) = \xi_{ij}t_{ij} - g_{ij}(t_{ij}), \quad \forall (i, j) \in A.$$

Take an arc $(i, j)$ such that $x_{ij} > \xi_{ij}$. Then the convexity of $f_{ij}$ implies

$$f_{ij}(x_{ij}) + (\xi_{ij} - x_{ij})f_{ij}^{-}(x_{ij}) \leq f_{ij}(\xi_{ij}) = \xi_{ij}t_{ij} - g_{ij}(t_{ij}).$$

Therefore

$$f_{ij}(x_{ij}) + g_{ij}(t_{ij}) \leq (x_{ij} - \xi_{ij})(f_{ij}^{-}(x_{ij}) - t_{ij}) + x_{ij}t_{ij}$$

$$\leq |x_{ij} - \xi_{ij}| \epsilon + x_{ij}t_{ij} \quad (4.2.12)$$

where the second inequality follows from $\epsilon$-CS. This inequality is obtained in a similar fashion when $x_{ij} \leq \xi_{ij}$. Therefore, inequality (4.2.12) holds for all arcs $(i, j) \in A$. From
the definition of the conjugate functions we have:

\[ x_{ij} t_{ij} \leq f_{ij}(x_{ij}) + g_{ij}(t_{ij}) \quad \forall (i,j) \in \mathcal{A}. \quad (4.2.13) \]

Since \( x \) is feasible we conclude that

\[ x_{ij} t_{ij} = \sum_{i \in A} p_i s_i. \quad (4.2.14) \]

Combining equations (4.2.12), (4.2.13), and (4.2.14) the result follows. \textbf{Q.E.D.}

The following proposition establishes that even if the flow bounds for the arcs may not be finite, the algorithm converges to the optimal solutions as \( \epsilon \) tends to zero. The proposition and its proof are identical to the one in \[BHT87\] and is listed here for completeness.

**Proposition 4.2.7** Let \( x(\epsilon) \) and \( p(\epsilon) \) denote any flow and price vector pair satisfying \( \epsilon \)-CS with \( x(\epsilon) \) feasible. Then \( f(x(\epsilon)) + q(p(\epsilon)) \to 0 \) as \( \epsilon \to 0 \).

**Proof:** First we will show that \( x(\epsilon) \) remains bounded as \( \epsilon \to 0 \). If \( x(\epsilon) \) is not bounded as \( \epsilon \to 0 \), then since \( x(\epsilon) \) is feasible for all \( \epsilon > 0 \) we conclude that there exists a directed cycle \( Y \) and a sequence \( \{\epsilon_n\} \to 0 \) such that \( u_{ij} = +\infty, x_{ij} \to +\infty \) for all \((i,j) \in Y^+\) and \( l_{ij} = -\infty, x_{ij} \to -\infty \) for all \((i,j) \in Y^-\). As discussed in Section 1, our assumption about the conjugate functions implies that

\[ \lim_{\xi \to +\infty} f_{ij}^{-}(\xi) = +\infty \quad \forall (i,j) \in Y^+ \quad \lim_{\xi \to -\infty} f_{ij}^{+}(\xi) = -\infty \quad \forall (i,j) \in Y^- . \]

This implies that for \( n \) sufficiently large we have:

\[ p_i(\epsilon_n) - p_j(\epsilon_n) > p_i(\epsilon_0) - p_j(\epsilon_0) \quad \forall (i,j) \in Y^+ \quad (4.2.15) \]

\[ p_i(\epsilon_n) - p_j(\epsilon_n) < p_i(\epsilon_0) - p_j(\epsilon_0) \quad \forall (i,j) \in Y^- . \]
However,

\[ \sum_{(i,j) \in Y^+} (p_i(\epsilon_n) - p_j(\epsilon_n)) - \sum_{(i,j) \in Y^-} (p_i(\epsilon_n) - p_j(\epsilon_n)) = \]

\[ = \sum_{(i,j) \in Y^+} (p_i(\epsilon_0) - p_j(\epsilon_0)) - \sum_{(i,j) \in Y^-} (p_i(\epsilon_0) - p_j(\epsilon_0)) = 0, \]

which contradicts (4.2.15). Therefore, \( x(\epsilon) \) is bounded as \( \epsilon \to 0 \).

Let \( \xi(\epsilon) \) be a flow that satisfies CS with the price vector \( p(\epsilon) \). We will show that \( \xi(\epsilon) \) is bounded. If \( u_{ij} < \infty \) then \( \xi_{ij}(\epsilon) \) is trivially bounded form above. If \( c_{ij} = +\infty \) then \( f_{ij}(\xi) \to +\infty \) as \( \xi \to +\infty \). Since \( x_{ij}(\epsilon) \) is bounded, we have \( p_i(\epsilon) - p_j(\epsilon) \) is bounded which in turn implies that \( \xi_{ij}(\epsilon) \) is bounded. A similar argument proves that \( \xi_{ij}(\epsilon) \) is bounded from below. Thus \( |\xi_{ij}(\epsilon) - x_{ij}(\epsilon)| \) is bounded for all \( (i,j) \in A \) as \( \epsilon \to 0 \). Combining this with lemma 2.3 the result follows. Q.E.D.

Proposition 4.2.7 cannot give us an estimate of how small \( \epsilon \) has to be in order to achieve a certain degree of optimality. In the case where all lower and upper bounds on the arc flows are finite, however, we can get an estimate of how small \( \epsilon \) has to be from the result of Proposition 4.2.6.

In the next section we derive a bound on the complexity of the algorithm. To do so, we will impose further assumptions on how the algorithm is operated.

### 4.3 Complexity Analysis

We continue the analysis of the \( \epsilon \)-relaxation algorithm by obtaining a bound on its running time. As we discussed in the introduction, it is not possible to express the size of the problem in terms of the problem data (e.g. flows and prices) since the cost functions are convex. A standard formulation that is amenable to complexity analysis techniques is to introduce an oracle that relates flows to costs. In particular, a standard oracle for the convex cost network flow problem would give us the following answers:
(1) When given a flow $x_{ij}$, the oracle returns the cost $f_{ij}(x_{ij})$, the left derivative $f_{ij}^{-}(x_{ij})$ and the right derivative $f_{ij}^{+}(x_{ij})$.

(2) When given a scalar $t_{ij}$, the oracle returns a flow $x_{ij}$ such that $f_{ij}^{-}(x_{ij}) \leq t_{ij} \leq f_{ij}^{+}(x_{ij})$.

Inquiries to such an oracle can be considered as simple operations. A complexity analysis of an algorithm for the convex cost network flow problem then becomes an estimate of the total number of simple operations performed by the algorithm.

To obtain the subsequent complexity bound, we need a rule for choosing the starting node at each iteration. In its pure form, where any node with positive surplus can be picked, our algorithm may take time which depends on the cost of the arcs. Such an example for the case where the costs are linear can be found in [BE88] (see also [BeT89, p. 384]). To avoid this behavior, we introduce an order in which the nodes are chosen in iterations. We will present the sweep implementation of the $\epsilon$-relaxation method which was introduced in [Ber86a] and was analyzed in more detail in [BE88], [BeT89], and [BC91] for the linear minimum cost flow problem. All the nodes are kept in a linked list $T$ which is traversed from the first to the last element. The order of the nodes in the list is consistent with the successor order implied by the admissible graph, that is, if a node $j$ is a successor of a node $i$, then $j$ must appear after $i$ in the list. If the initial admissible graph is empty, as is the case with the initialization of Eq. (6), the initial list is arbitrary. Otherwise, the initial list must be consistent with the successor order of the initial admissible graph. The list is updated in a way that maintains the consistency with the successor order. In particular, let $i$ be a source on which we perform an $\epsilon$-relaxation iteration. Let $N_i$ contain the nodes of $T$ that are lower than $i$ in $T$. If the price of $i$ changes, then node $i$ is removed from its position in $T$ and placed in the first list position. The next node to be picked for an iteration is the node $i' \in N_i$ with positive surplus which ranks highest in $T$, if such a node exists. Otherwise, the positive surplus node ranking highest in $T$ is picked.
A sweep cycle is a set of iterations whereby all nodes are chosen once from the list and an $\epsilon-$relaxation iteration is performed on nodes with positive surplus. The idea of the sweep implementation is that an $\epsilon-$relaxation iteration on a node $i$ that has predecessors with positive surplus may be wasteful since its surplus may be set to zero through a flow push and become positive again by a flow push at a predecessor node. This will not happen if the nodes are chosen according to the partial order induced by the admissible graph. The next lemma (see also the references cited earlier) proves that the list $T$ is compatible with the order induced by the admissible graph in the sense that every node will appear in the list after all its predecessors.

**Proposition 4.3.1** Every node appears in $T$ after all its predecessors.

**Proof:** We employ induction. As discussed in the previous section, the admissible graph is empty initially. Thus the lemma holds in the beginning of the algorithm. Next we note that a flow push does not create new predecessor relationships. Finally, after the price of a node $i$ rises, $i$ can have no predecessors and is moved to the head of the list before any possible descendants. Q.E.D.

We are now ready to derive our main complexity result about the algorithm. The dominant computational requirements are:

1. The computation required for price increase.
2. The computation required for saturating $\delta-$flow pushes.
3. The computation required for nonsaturating $\delta-$flow pushes.

Since we are using push lists $L_i$ that hold the arcs of the admissible graph, the computation time to examine any one arc in the algorithm is $O(1)$. Since there are $O(KN)$ price increases per node the requirements for (1) above are $O(KNA)$ operations. Furthermore, whenever a $\delta-$flow push is saturating, it takes a price increase of at least $\frac{\epsilon}{2}$ on one of its end nodes before the flow can be changed again. Thus the total requirement for (2) above is $O(KNA)$ operations. Finally, for (3) above we note that for each cycle there can be only one exhaustive $\delta-$flow push per node. Thus a time
Proposition 4.3.2 Assume that for some $K \geq 1$ the initial price vector $p^0$ for the $\epsilon-$relaxation method satisfies $K\epsilon-$CS together with some feasible flow $x^0$. Then, the number of cycles up to termination is $O(KN^2)$.

Proof: Let $N^+$ be the set of nodes with positive surplus that have no predecessor with positive surplus and let $N^0$ be the set of nodes with nonpositive surplus that have no predecessor with positive surplus. Then, as long as no price change takes place, all nodes in $N^0$ remain in $N^0$ and an iteration on a node $i \in N^+$ moves $i$ from $N^+$ to $N^0$. If no node changed price during a cycle then all nodes in $N^+$ will be moved to $N^0$ and the algorithm terminates. Therefore, there is a node price change in every cycle except possible the last one. Thus there are $O(KN^2)$ cycles. Q.E.D.

Thus (3) above takes $O(KN^3)$ operations. Adding the computational requirements for (1) and (2) and (3), and using the fact that $A \leq N^2$, we obtain an $O(KN^3)$ time bound for the algorithm.

Proposition 4.3.3 Assume that for some $K \geq 1$ the initial price vector $p^0$ for the $\epsilon-$relaxation method satisfies $K\epsilon-$CS together with some feasible flow $x^0$. Then, the method requires $O(KN^3)$ operations up to termination.

$\epsilon-$Scaling

The dependence of the running time bound on the parameter $K$ shows the sensitivity of the algorithm on the cost functions. A popular technique to get an improved complexity bound for network flow algorithms in linear programming is cost scaling (see [BlJ85], [EdK72], [Roc80]). An analysis of cost scaling applied on $\epsilon-$relaxation for the linear network flow problem is due to [BE87], (presented also in the subsequent publication [BE88]). In the convex cost case, however, cost scaling may be difficult to implement since the costs of arcs may be unbounded. A second scaling approach in connection with the $\epsilon-$relaxation method for linear problems, is $\epsilon-$scaling. This
method was originally introduced in [Ber79] as a means of improving the performance of the auction algorithm based on computational experimentation. It was further analyzed in [Gol87a], [GolT87b]. The reader is also referred to the textbooks [BeT89] and [Ber91].

The key idea of ε-scaling is to apply the algorithm several times, starting with a large value of ε and to successively reduce ε up to a final value that will give the desirable degree of accuracy to our solution. Furthermore, the price and flow information from one application of the algorithm is transferred to the next.

The basic form of the procedure is as follows: initially we select a scalar θ ∈ (0, 1) and we choose a value for ε⁰, and for k = 1,... we set εᵏ = θεᵏ⁻¹. The algorithm is applied for k = 1,... until k where k is the first integer for which εᵏ is below some desirable level ε'. Let (xᵏ, pᵏ) be the flow-price vector pair obtained at the kth application of the algorithm. Then xᵏ is feasible and satisfies εᵏ-CS with pᵏ. Furthermore, the admissible graph Gₖ after the kth application of the algorithm is acyclic. The starting price vector for the (k + 1)st application is pᵏ and the starting flow is xᵢⱼᵏ for the arcs (i, j) that satisfy εᵏ⁺¹-CS with pᵏ, and

\[ \sup \{ \xi | f_{ij}^+(\xi) \leq p_i^k - p_j^k - \frac{\varepsilon^{k+1}}{2}\} \]

otherwise. Observe that this choice of initial flows does not introduce any new arcs to Gₖ; as a result, the starting admissible graph for the (k + 1)st application of the algorithm is acyclic. The starting flow-price vector pair for the initial application of the algorithm satisfy ε⁰-CS and the initial admissible graph G₀ is acyclic.

We observe that at the beginning of the (k + 1)st application of the algorithm. (for k = 1,..., k), pᵏ satisfies εᵏ⁺¹ -CS with the feasible flow vector xᵏ. Based on the complexity analysis presented above, we conclude that the (k + 1)st application of the algorithm has running time O(θN³). However, θ is just a scalar and we conclude that the running time of the (k + 1)st application of the algorithm is O(N³). The algorithm will be repeated at most \( k = \lfloor \log_\theta (\frac{\varepsilon'}{\varepsilon'}) \rfloor \) where, as we mentioned earlier, ε' is the desirable final value for ε. Thus we have obtained the following:
Proposition 4.3.4 The running time of the $\epsilon$-relaxation algorithm with the sweep implementation and $\epsilon$-scaling as described above is $O(N^3 \ln(\frac{\epsilon}{\epsilon'})$).

In the introduction we referred to the work of [KMc93], where the tighten and cancel method for the convex cost problem is analyzed and the complexity bound of $O(NA \ln(N) \ln(\frac{\epsilon}{\epsilon'}))$ is derived. The complexity bound we obtained for the $\epsilon$-relaxation method is comparable to the one for the tighten and cancel method. In particular, we observe that for relatively dense problems where $A = \Theta(\frac{N^2}{\ln N})$ the $\epsilon$-relaxation algorithm has a more favorable complexity bound, while for sparse problems where $A = \Theta(N)$, the reverse is true.

4.4 The Reverse and Combined $\epsilon$-Relaxation Algorithms

The $\epsilon$-relaxation algorithm we presented in the previous section performed iterations only on nodes with positive surplus and we called it the forward algorithm. We can modify the forward algorithm so that iterations can be performed on nodes with negative surplus. We refer to this algorithm as the reverse algorithm. In the sequel, we give definitions analogous to the push list of a node and the $\delta$-flow push on an arc in the push list. In particular, given a pair $(x, p)$ satisfying $\epsilon$-CS, we define for each node $i \in N$ its pull list as the following set of arcs:

$$\{ (j, i) \mid \frac{\epsilon}{2} < p_j - p_i - f_{ji}^+(x_{ji}) \leq \epsilon \} \cup \{ (i, j) \mid -\epsilon \leq p_i - p_j - f_{ij}^-(x_{ij}) < -\frac{\epsilon}{2} \}.$$  

An arc $(j, i)$ (or $(i, j)$) of the pull list of the negative surplus node $i$ is said to be unblocked if there exists a $\delta > 0$ such that

$$p_j - p_i \geq f_{ji}^+(x_{ji} + \delta),$$  

(or $p_i - p_j \leq f_{ij}^-(x_{ij} - \delta)$, respectively). With a similar argument to the one in Section 4.2 we conclude that arcs in the pull list of a node are unblocked. For a pull list arc,
the supremum of $\delta$ for which the above relation holds is called the flow margin of the arc. Observe that our definitions of the pull list of a node do not affect the definition of an unblocked arc or of the flow margin of an arc. It can be seen with a similar analysis as in Section 4.2 that a pull list arc is necessarily unblocked. Furthermore, we observe that an arc $(i, j)$ (or $(j, i)$) is in the push list of node $i$ if and only if arc $(i, j)$ (or $(j, i)$ respectively) is in the pull list of node $j$. In addition, the direction of flow change on arc $(i, j)$ (or $(j, i)$) is the same regardless of whether we consider the arc as being in the push list of $i$ or as being in the pull list of $j$. Finally, as a result of our definitions, an arc $(i, j)$ (or $(j, i)$) cannot be simultaneously in the push and pull list of node $i$. Thus the set of nodes $N$ and the set $A^*$ containing all unblocked pull list arcs oriented in the direction of flow change define the admissible graph $G^* = (N, A^*)$.

We can perform two operations on a node of negative surplus:

- A price decrease on node $i$ decreases the price $p_i$ by the maximum amount that maintains $\epsilon-CS$.

- A $\delta$-flow pull along an arc $(j, i)$ (or along an arc $(i, j)$) consists of increasing the surplus $g_i$ of the node $i$ by an amount $\delta \leq g_i$ and increasing the flow of arc $(j, i)$ by $\delta$ (or decreasing the flow of $(i, j)$ by $\delta$, respectively). As a result the surplus $g_j$ of the opposite node $j$ decreases by $\delta$.

For a set of prices $\bar{p}_i$ we initialize the flows the same way as for the forward algorithm. i.e., we set

$$\bar{x}_{ij} = \sup\{\xi \mid f_{ij}^+(\xi) \leq \bar{p}_i - \bar{p}_j - \frac{\epsilon}{2}\}.$$

We observe that this initialization makes the initial admissible graph empty and thus acyclic. The reverse algorithm would operate as follows:

**Typical Iteration of the Reverse Algorithm**

**Step 1:** Select a node $i$ with negative surplus; if no such node exists, terminate.
Step 2: If the reverse push list of \( i \) is empty then go to Step 3. Otherwise, find an arc in the pull list of \( i \) and make a \( \delta \)-flow pull from the opposite node \( j \), where

\[
\delta = \min\{-g_i, \text{flow margin of arc}\}.
\]

If the surplus of \( i \) becomes zero, go to the next iteration; otherwise go to Step 2.

Step 3: Perform a price decrease at node \( i \). Proceed to a new iteration.

The reverse \( \epsilon \)-relaxation algorithm is the “mirror image” of the forward algorithm that we developed in the previous sections. Naturally, it has similar properties to the forward algorithm and its validity follows from a similar analysis. Thus the admissible graph remains acyclic throughout the algorithm and the algorithm terminates. The sweep implementation can also be modified appropriately so that it works for the reverse algorithm: All nodes are kept in a linked list \( T' \) which is traversed from the last to the first element. The initial list is arbitrary. A cycle is a set of iterations whereby all nodes are chosen once from the list and an \( \epsilon \)-relaxation is performed on nodes with negative surplus. Let \( i \) be a sink on which we perform a reverse \( \epsilon \)-relaxation iteration. Let \( N'_i \) contain the nodes of \( T' \) which are above \( i \) in \( T' \). If the price of \( i \) changes, then the node \( i \) is removed from its position in \( T' \) and placed in the last list position. The next node to be picked for an iteration is the node \( i' \in N'_i \) with negative surplus that ranks lowest in \( T' \), if such a node exists. Otherwise, the negative surplus node ranking lowest in \( T' \) is picked. It is easy to verify that the linked list structure \( T' \) is also related to the admissible graph. Actually, every node in \( T' \) appears after all its predecessors since if the price of a node decreases, that node can have no successors and is moved at the end of the list after any possible predecessors. Thus going through \( T' \) is like sweeping the nodes of negative surplus staring from those with no successors. Therefore, the complexity bound we obtained for the forward algorithm also holds for the reverse algorithm with the sweep implementation we just described.
It should be noted that while the forward algorithm creates the admissible graph starting from nodes with positive surplus, the reverse algorithm builds the admissible graph from nodes with negative surplus. Both algorithms effectively build the admissible graph so that it contains paths from sources to sinks thus allowing flow to decrease in the sources and increase in the sinks. The direction of flow changes on arcs of the admissible graph is the same for both algorithms. It is possible, therefore, to combine the forward and the reverse algorithm so that the resulting algorithm will operate on both positive and negative surplus nodes. Our intuition is that if we perform $\epsilon$-relaxation iterations on both sources and sinks, we will be able find the optimal solution faster for certain classes of problems, since the admissible graph is built both from the sources and the sinks. We refer to the resulting algorithm as the *combined algorithm*. We initialize the arc flows and node prices the same way we initialized them for the forward and the reverse algorithms so that the initial admissible graph is acyclic. The combined algorithm operates as follows:

*Typical Iteration of the Combined Algorithm*

Pick a node $i$ with nonzero surplus; if no such node exists then terminate. If $i$ has positive surplus then perform an iteration of the forward $\epsilon$-relaxation algorithm. If $i$ has negative surplus then perform an iteration of the reverse $\epsilon$-relaxation algorithm.

The combined algorithm maintains the acyclicity of the admissible graph since both its constituent algorithms (the forward and the reverse) do so. Thus, based on our earlier analysis for the forward algorithm, the number of flow pushes that the combined algorithm can perform before a price change can occur at a node are bounded. However, termination is not easily guaranteed for the algorithm as described so far. The reason is that a naive combination of the forward and reverse algorithms could make sources become sinks and vice-versa. Then it is possible that prices will oscillate and the algorithm will not terminate. An example illustrating this situation can be found in [BeT89, p. 373] where for linear costs $\epsilon$-relaxation fails to converge when
forward and reverse iterations are performed. One way to ensure that the algorithm terminates is to make the following assumption:

**Assumption 3** The number of times the surplus of a node changes sign is finite.

This idea is recurrent in many relaxation-like algorithms. Versions of this kind of assumption for two-sided algorithms can be found in [Ts86] for the relaxation algorithm and in [Pol92] for the auction shortest path algorithm. Termination is now guaranteed since after a finite number of iterations (say \( c \) iterations) we have two sets of nodes: A set consisting of nodes that may have non-negative surplus and whose prices are nondecreasing, and a set consisting of nodes that may have non-positive surplus and whose prices are nonincreasing. Since the admissible graph remains acyclic throughout the algorithm, we conclude that the number of possible \( \delta \)-flow pushes between successive price rises (not necessarily at the same node) is bounded. We only need to establish that the number of price rises that can be performed on each node is bounded. This can be proved with an identical analysis to the one in Proposition 4.2.3 except for the fact that the price of the sinks decreases. In particular, assume that after \( c \) iterations there is no node whose surplus will change sign in any subsequent iteration. Then, Proposition 4.2.3 for the combined algorithm can be stated as follows:

**Proposition 4.4.1** If the price vector after the \( c \) initial \( \epsilon \)-relaxation iterations is \( p^0 \) and it satisfies \( K \epsilon \)-CS with some feasible flow vector \( x^0 \) for some \( K \geq 1 \), then the \( \epsilon \)-relaxation method performs at most \( 2(K + 1)(N - 1) \) price increases per node.

**Proof:** Consider the pair \((x, p)\) at the beginning of an \( \epsilon \)-relaxation iteration. Since the surplus vector \( g = (g_1, \ldots, g_N) \) is not zero, and the flow vector \( x^0 \) is feasible, we conclude that for each source \( s \) with \( g_s > 0 \) there exists a sink \( t \) with \( g_t < 0 \) and a simple (i.e., having no cycles) path \( H \) from \( t \) to \( s \) such that:

\[
x_{ij} > x^0_{ij} \quad \forall (i, j) \in H^+
\]  
(4.4.16)
where $H^+$ is the set of forward arcs of $H$ and $H^-$ is the set of backward arcs of $H$.

(This can be seen from the Conformal Realization theorem ([Roc84] or [Ber91]) as follows. Consider the following problem: We replace the upper bounds on the flow by $u_{ij} - x_{ij}^0$, the lower bounds by $l_{ij} - x_{ij}^0$, the flow vector by $x^* = x - x^0$ and the $s_i$ by $0$. The transformation leaves the surplus of every node unchanged and does not change the prices generated by the algorithm. Since $g \neq 0$, by the Conformal Realization Theorem, we can find a path $H$ from a sink $t$ to a source $s$ which conforms to the flow $x$, i.e., the flow on the path is positive for all $(i, j) \in H^+$ and the flow is negative for all $(i, j) \in H^-$. Thus equations (4.4.16) and (4.4.17) follow.)

From Eqs. (4.2.3.1) and (4.4.17), and the convexity of the functions $f_{ij}$, for all $(i, j) \in A$ we conclude that

$$f_{ij}^-(x_{ij}) \geq f_{ij}^+(x_{ij}^0) \quad \forall (i, j) \in H^+$$

(4.4.18)

$$f_{ij}^+(x_{ij}) \leq f_{ij}^-(x_{ij}^0) \quad \forall (i, j) \in H^-$$

Since the pair $(p, x)$ satisfies $\epsilon$-CS we conclude that

$$p_i - p_j \in [f_{ij}^-(x_{ij}) - \epsilon, f_{ij}^+(x_{ij}) + \epsilon] \quad \forall (i, j) \in A$$

(4.4.19)

Similarly, since the pair $(p^0, x^0)$ satisfies $K\epsilon$-CS we have

$$p_i^0 - p_j^0 \in [f_{ij}^-(x_{ij}^0) - K\epsilon, f_{ij}^+(x_{ij}^0) + K\epsilon] \quad \forall (i, j) \in A$$

(4.4.20)

Combining equations (4.4.18), (4.4.19), (4.4.20), we derive the following relations:

$$p_i - p_j \geq p_i^0 - p_j^0 - (K + 1)\epsilon \quad \forall (i, j) \in H^+$$

(4.4.21)

$$p_i - p_j \leq p_i^0 - p_j^0 + (K + 1)\epsilon \quad \forall (i, j) \in H^-$$

(4.4.22)
Summing the above equations over all arcs of the path $H$ we get

$$p_t - p_s \geq p_t^0 - p_s^0 - (K + 1)|H|e, \quad (4.4.23)$$

where $(x, p)$ is the flow-price pair at the beginning of a combined ASSP iteration. $p^0$ is the initial price vector, and $|H|$ denotes the number of arcs of the path $H$. However, if a node is a sink we know that $p_t - p_t^0 \leq 0$, and we conclude that

$$p_s - p_s^0 \leq (K + 1)N\epsilon.$$

**Q.E.D.**

A similar argument proves that the number of possible price decreases that can be performed is finite. Thus the combined algorithm is shown to terminate.

For an actual implementation of the combined algorithm assumption 3 is easy to be satisfied by having a counter for each node that records the number of times the surplus of each node changes sign. If some counter exceeds a threshold then we may allow only forward iterations until the a new scaling phase begins or the algorithms terminates.

Another way to ensure termination is never to allow sources to become sinks. This can be done with a simple modification to the reverse flow push:

**Modified Reverse Flow Push:**

If an arc $(j, i) \in A$ is in the pull list of $i$, then:

(a) If $j$ is a source then increase the flow $x_{ji}$ by an amount

$$\delta = \min\{g_j, \text{flow margin of } (j, i)\}.$$

(b) If $j$ is a sink then increase the flow $x_{ji}$ by an amount

$$\delta = \min\{-g_i, \text{flow margin of } (j, i)\}.$$
If an arc \((i, j) \in \mathcal{A}\) is in the pull list of \(i\), then:

(a) If \(j\) is a source, decrease flow \(x_{ij}\) by an amount

\[
\delta = \min\{g_j, \text{flow margin of } (i, j)\}.
\]

(b) If \(j\) is a sink then decrease the flow \(x_{ij}\) by an amount

\[
\delta = \min\{-g_i, \text{flow margin of } (i, j)\}.
\]

With this flow pull we have that if the surplus of a node becomes nonnegative in the course of the algorithm then it remains nonnegative throughout the algorithm. Furthermore, if a node has negative surplus at some point then its price is less than or equal to its initial price and the node has never become a source so far. Observe, however, iterations on a sink \(i\) may change neither the flow nor the price of the node. This case may arise when the arc \((j, i)\) or \((i, j)\) satisfying \(\epsilon\)-CS with equality is incident to a node \(j\) with zero surplus. To ensure termination, a price and/or flow changing iteration must occur before a new iteration at node \(i\) is attempted. This is easy to do: perform an iteration at a node with positive surplus. Thus if we assume that we perform iterations on nodes with non-zero surplus in a cyclic fashion, then the modified combined algorithm will terminate.

Finally, the combined algorithm also accepts a sweep implementation. This can be done by maintaining two lists \(T\) and \(T'\), the former organized appropriately for the forward algorithm and the latter organized for the reverse algorithm. Therefore, nodes with positive surplus are picked from \(T\), whereas nodes with negative surplus are picked from \(T'\) in accordance to the respective sweep implementation. Thus we could alternately cycle through \(T\) and \(T'\) performing one cycle for forward iterations and one cycle of reverse iterations. A scheme whereby we mesh the two types of iterations can also be devised. Thus the complexity bound we proved for the one-sided algorithms is valid for the combined algorithm as well.
Extensive computational experimentation is needed to evaluate the performance of the one-sided and the two-sided algorithms and to compare the $\epsilon$-relaxation schemes with other successful algorithms (like the relaxation algorithm) for the convex cost network flow problem.

4.5 Computational Results

We developed two experimental codes implementing the forward and the combined $\epsilon$-relaxation algorithms, respectively, of this chapter and we tested them on a variety of convex cost problems.

The first code, named NE-RELAX-F, implements the forward $\epsilon$-relaxation method and of Section 4.2. The second, named NE-RELAX-FB, implements the combined algorithm of Section 4.4. Both codes were written in FORTRAN and were based on the corresponding implementations of the forward and combined $\epsilon$-relaxation algorithms for the linear cost problems. The implementations of the linear cost algorithms are the ones presented in Appendix A.7 of [Ber91] and have been shown to be quite efficient. Several changes and enhancements were introduced for the convex cost implementations: All computations involve real numbers rather than integers and no arc cost scaling is performed. Furthermore, the computations involving the push lists and price increases are different. In particular, the push lists for the nonlinear code usually involve more arcs than in the case of the linear cost implementation. Finally, the termination criterion is different: The linear cost algorithms for integer costs terminate when the value of $\epsilon$ is less than $\frac{1}{(n+1)}$, whereas the convex cost algorithms terminate when the primal and dual costs agree on a user specified number of digits. In all the our testing we used 15 digits of accuracy for our testing of the our implementations unless otherwise indicated. However, the sweep implementation, $\epsilon$-scaling and the general structure of the codes for linear and convex problems are identical. Initial testing on linear cost problems was performed using both the linear and the convex cost codes. Those tests showed that the convex cost implementations performed equally well to their linear counterparts which shows that convex cost algo-
rithms were coded efficiently. The different method for the creation of the push lists in combination with the speed of floating point unit computations often favored the convex cost implementation even though the termination criterion was more stringent for the convex cost codes than for their linear counterparts.

The forward and the combined $\epsilon-$relaxation algorithms were compared to the existing implementations of the relaxation methods for strictly convex and for mixed linear and strictly convex problems of [BHT87]. The code named NRELAX implements the relaxation method for strictly convex cost problems, whereas the code named MNRELAX implements the relaxation method for mixed linear and convex cost problems. Both codes are written in FORTRAN and have been shown ([BHT87]) to be quite efficient. All FORTRAN programs were compiled on a SUN SPARC-5 with 24 megabytes of RAM under the SOLARIS operating system. We also used the -O compiler optimization switch in order to take advantage of the floating point unit and the design characteristics of the SPARC-5 processor.

The test problems were created by two problem generators: (i) the public domain code NETGEN, by D. Klingman, A. Napier and J. Stutz [KNS74], which generates feasible assignment/transportation/transshipment problems with random structure and linear costs. We modified the created problems so that a user specified percent of the arcs had an additional quadratic cost coefficient in a user specified range; (ii) a generator that we wrote called CHAINGEN. This generator creates a long chain involving all the nodes. The user also specifies the additional number of arcs per node that should be generated. For example, if the user specifies 3 additional arcs per node $i$ then the arcs $(i, i + 2)$, $(i, i + 3)$, $(i, i + 4)$ are generated with random linear costs. Furthermore, the user can specify the percentage of nodes $i$ for which an additional arc $(i, i - 1)$ is generated with random linear cost. The resulting graphs have long diameters and our testing with linear cost algorithms has shown that these are particularly difficult problems for all methods. We modified the created problems so that a user specified percent of the arcs had an additional quadratic cost coefficient in a user specified range.

Our testing revolved around three main issues:
(a) the relative efficiency of the $\epsilon-$relaxation methods versus the relaxation methods;

(b) the insensitivity of the $\epsilon-$relaxation methods to ill-conditioned problems. For our tests, ill-conditioning appears when some of the arcs have large quadratic cost coefficients and some have very small quadratic cost coefficients;

(c) the effect of different graph architectures on the relative performance of the tested algorithms.

To better compare our results with those reported in [BHT87] we generated the NETGEN problems of the same size as in Table 1 of [BHT87]. These problems are described in Table 4.1 below. The computational results that we obtained are shown in Table 4.2. Our testing indicates that the $\epsilon-$relaxation algorithms are consistently faster than the relaxation algorithms on all the problems of Table 1. On average, the $\epsilon-$relaxation algorithms are about 3 times faster than the relaxation algorithms with a maximum of about 8 times faster running time.

The effect of ill-conditioning on the algorithms of this paper was demonstrated with the test problems described in Table 4.3. We generated problems which are increasingly ill-conditioned and the results of our testing are shown in Table 4.4. As expected the NRELAX code was affected dramatically by the amount of ill-conditioning and its performance was quite poor. The MNRELAX code was relatively insensitive to ill-conditioning which agrees with the results reported in [BHT87]. The $\epsilon-$relaxation implementations were particularly successful on our test problems. Not only was their performance almost unaffected by ill-conditioning, but their running times were consistently faster than the MNRELAX code by at least a factor of 5.

Finally, the effect of graph architecture on the relative efficiency of the algorithms of this paper was demonstrated by the CHAINGEN problems described in Table 4.5. The results of our testing are shown in Table 4.6. These problems involve long paths from the source to the sink and thus are quite difficult. The $\epsilon-$relaxation implementations outperform MNRELAX by at least an order of magnitude.

Our conclusion is that the $\epsilon-$relaxation algorithms of this chapter are very successful in solving the convex cost network flow problem. In particular, we believe that
the strength of the method lies in the fact that instead of searching for directions of dual cost improvement, it performs local price increases and flow pushes. This has two advantages that were observed during our testing: First, fewer computations are need for flow pushes and price rises. Secondly, the method is very stable numerically since flow pushes and price rises only involve one node. Finally, the \( \epsilon \)-relaxation method seems unaffected by ill-conditioning, thus becoming a method of choice for difficult problems for which existing methods have had little success.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Nodes</th>
<th>Arcs</th>
<th>Linear Cost</th>
<th>Quad Cost</th>
<th>Total Surplus</th>
<th>Capacity Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob1</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob2</td>
<td>200</td>
<td>1500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob3</td>
<td>200</td>
<td>2000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob4</td>
<td>200</td>
<td>2200</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob5</td>
<td>200</td>
<td>2900</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob6</td>
<td>300</td>
<td>3150</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob7</td>
<td>300</td>
<td>4500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob8</td>
<td>300</td>
<td>5155</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob9</td>
<td>300</td>
<td>6075</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob10</td>
<td>300</td>
<td>6300</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob11</td>
<td>400</td>
<td>1500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob12</td>
<td>400</td>
<td>2250</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob13</td>
<td>400</td>
<td>3000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob14</td>
<td>400</td>
<td>3750</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob15</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob16</td>
<td>400</td>
<td>1306</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob17</td>
<td>400</td>
<td>2443</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob18</td>
<td>400</td>
<td>1416</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob19</td>
<td>400</td>
<td>2836</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob20</td>
<td>400</td>
<td>1382</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob21</td>
<td>400</td>
<td>2676</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob22</td>
<td>1000</td>
<td>3000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[1000-2000]</td>
</tr>
<tr>
<td>prob23</td>
<td>1000</td>
<td>5000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[1000-2000]</td>
</tr>
<tr>
<td>prob24</td>
<td>1000</td>
<td>10000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>1000</td>
<td>[1000-2000]</td>
</tr>
</tbody>
</table>

Table 4.1: The NETGEN problems with all arcs having quadratic coefficients. The problems prob1-prob17 are identical to the problems 1-17 of Table 1 of [BHT87]. The problems named prob18, prob19, prob20, prob21 correspond to the problems 20, 23, 24, 25, respectively, of Table 1 of [BHT87].
<table>
<thead>
<tr>
<th>Problem</th>
<th>NRELAX</th>
<th>MNRELAX</th>
<th>NE-RELAX-F</th>
<th>NE-RELAX-FB</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob1</td>
<td>7.95</td>
<td>6.0</td>
<td>1.95</td>
<td>1.72</td>
</tr>
<tr>
<td>prob2</td>
<td>7.55</td>
<td>6.35</td>
<td>2.13</td>
<td>1.79</td>
</tr>
<tr>
<td>prob3</td>
<td>2.88</td>
<td>5.65</td>
<td>2.13</td>
<td>1.79</td>
</tr>
<tr>
<td>prob4</td>
<td>20.45</td>
<td>10.62</td>
<td>2.4</td>
<td>2.02</td>
</tr>
<tr>
<td>prob5</td>
<td>2.32</td>
<td>24.8</td>
<td>1.45</td>
<td>1.2</td>
</tr>
<tr>
<td>prob6</td>
<td>7.31</td>
<td>22.11</td>
<td>2.71</td>
<td>1.6</td>
</tr>
<tr>
<td>prob7</td>
<td>7.52</td>
<td>21.12</td>
<td>3.94</td>
<td>2.6</td>
</tr>
<tr>
<td>prob8</td>
<td>48.3</td>
<td>26.72</td>
<td>3.88</td>
<td>2.06</td>
</tr>
<tr>
<td>prob9</td>
<td>7.25</td>
<td>22.71</td>
<td>3.22</td>
<td>2.48</td>
</tr>
<tr>
<td>prob10</td>
<td>4.41</td>
<td>31.53</td>
<td>2.7</td>
<td>3.3</td>
</tr>
<tr>
<td>prob11</td>
<td>69.25</td>
<td>15.07</td>
<td>8.79</td>
<td>7.8</td>
</tr>
<tr>
<td>prob12</td>
<td>17.68</td>
<td>17.24</td>
<td>4.98</td>
<td>4.3</td>
</tr>
<tr>
<td>prob13</td>
<td>22.00</td>
<td>20.43</td>
<td>7.3</td>
<td>6.00</td>
</tr>
<tr>
<td>prob14</td>
<td>13.2</td>
<td>24.3</td>
<td>3.03</td>
<td>1.82</td>
</tr>
<tr>
<td>prob15</td>
<td>10.01</td>
<td>35.99</td>
<td>7.42</td>
<td>6.23</td>
</tr>
<tr>
<td>prob16</td>
<td>85.10</td>
<td>25.46</td>
<td>8.64</td>
<td>8.4</td>
</tr>
<tr>
<td>prob17</td>
<td>31.63</td>
<td>21.52</td>
<td>7.38</td>
<td>6.24</td>
</tr>
<tr>
<td>prob18</td>
<td>7.51</td>
<td>9.03</td>
<td>0.96</td>
<td>0.94</td>
</tr>
<tr>
<td>prob19</td>
<td>45.43</td>
<td>26.76</td>
<td>8.63</td>
<td>7.84</td>
</tr>
<tr>
<td>prob20</td>
<td>79.96</td>
<td>17.71</td>
<td>9.95</td>
<td>8.43</td>
</tr>
<tr>
<td>prob21</td>
<td>33.48</td>
<td>23.97</td>
<td>6.8</td>
<td>5.88</td>
</tr>
<tr>
<td>prob22</td>
<td>64.42</td>
<td>50.94</td>
<td>8.46</td>
<td>4.11</td>
</tr>
<tr>
<td>prob23</td>
<td>26.7</td>
<td>49.06</td>
<td>4.08</td>
<td>3.5</td>
</tr>
<tr>
<td>prob24</td>
<td>26</td>
<td>323.296</td>
<td>5.53</td>
<td>5.58</td>
</tr>
</tbody>
</table>

Table 4.2: Computational Results on a Sun Sparc 5 with 24MB memory. Running times are in seconds.
Table 4.3: The NETGEN problems with all arcs having quadratic coefficients. Half of the arcs have costs in the range [5,10] and the remaining half have the small quadratic coefficient indicated. The problems prob6 and prob12 are mixed convex problems where half of the arcs have linear cost and the remaining half had quadratic costs.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Nodes</th>
<th>Arcs</th>
<th>Linear Cost</th>
<th>Small Quad Cost</th>
<th>Total Surplus</th>
<th>Capacity Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob1</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>1</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob2</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>0.1</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob3</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>0.01</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob4</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>0.001</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob5</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>0.0001</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob6</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>0</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob7</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>1</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob8</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>0.1</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob9</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>0.01</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob10</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>0.001</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob11</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>0.0001</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
<tr>
<td>prob12</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>0</td>
<td>1000</td>
<td>[100-300]</td>
</tr>
</tbody>
</table>

Table 4.4: Computational Results on a Sun Sparc 5 with 24MB memory. Running times are in seconds. For the very ill-conditioned problems NRELAX was taking extremely long running time. For these cases we reduced the accuracy of the answer given by NRELAX in order to get a meaningful result. The numbers in parentheses indicate the number of significant digits of accuracy of the answer given by NRELAX. The running times of NE-RELAX-F, and NE-RELAX-FB for the mixed convex problems prob6 and prob12 are included to demonstrate the fact that these methods are not affected significantly by increasing ill-conditioning.

<table>
<thead>
<tr>
<th>Problem</th>
<th>NRELAX</th>
<th>MNRELAX</th>
<th>NE-RELAX-F</th>
<th>NE-RELAX-FB</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob1</td>
<td>3.6</td>
<td>3.6</td>
<td>0.5</td>
<td>0.52</td>
</tr>
<tr>
<td>prob2</td>
<td>20.95</td>
<td>4.3</td>
<td>0.61</td>
<td>0.97</td>
</tr>
<tr>
<td>prob3</td>
<td>56.1</td>
<td>3.6</td>
<td>0.67</td>
<td>1.14</td>
</tr>
<tr>
<td>prob4</td>
<td>(5)791.24</td>
<td>3.28</td>
<td>0.73</td>
<td>0.71</td>
</tr>
<tr>
<td>prob5</td>
<td>(5)1866.67</td>
<td>2.7</td>
<td>0.77</td>
<td>0.75</td>
</tr>
<tr>
<td>prob6</td>
<td>-</td>
<td>2.23</td>
<td>0.69</td>
<td>0.83</td>
</tr>
<tr>
<td>prob7</td>
<td>52.22</td>
<td>14.1</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>prob8</td>
<td>53.42</td>
<td>11.26</td>
<td>1.88</td>
<td>2.03</td>
</tr>
<tr>
<td>prob9</td>
<td>(5)80.5</td>
<td>13.76</td>
<td>2.3</td>
<td>2.68</td>
</tr>
<tr>
<td>prob10</td>
<td>(5)710.73</td>
<td>15.0</td>
<td>2.67</td>
<td>2.64</td>
</tr>
<tr>
<td>prob11</td>
<td>(4)5753.45</td>
<td>13.56</td>
<td>3.67</td>
<td>3.1</td>
</tr>
<tr>
<td>prob12</td>
<td>-</td>
<td>8.33</td>
<td>2.79</td>
<td>2.62</td>
</tr>
<tr>
<td>Problem Name</td>
<td>Nodes</td>
<td>Linear Cost</td>
<td>Total Surplus</td>
<td>Capacity Range</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-------------</td>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>prob1</td>
<td>50</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob2</td>
<td>100</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob3</td>
<td>150</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob4</td>
<td>200</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob5</td>
<td>250</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob6</td>
<td>300</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob7</td>
<td>350</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob8</td>
<td>400</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob9</td>
<td>450</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
<tr>
<td>prob10</td>
<td>500</td>
<td>[1-100]</td>
<td>1000</td>
<td>[100-1000]</td>
</tr>
</tbody>
</table>

Table 4.5: The CHAINGEN problems with all arcs having quadratic coefficients in the range [5,10]. Half of the nodes have an additional reverse arc.

Table 4.6: Computational Results on a Sun Sparc 5 with 24MB memory. Running times are in seconds. For these problems the NRELAX was taking extremely long running times even for 5 digits of accuracy. For this reason we are not reporting any running times for NRELAX on these problems. MNRELAX also was taking a very long time for the last problem and was stopped before termination. This is indicated by the > in front of the recorded time.
Chapter 5

The Generic Auction Algorithm

This chapter generalizes the $\epsilon$—relaxation algorithm we introduced in chapter 4. In particular, price increases can occur simultaneously at more than one node. Furthermore, we can perform cycles of many multiple price increases and at least one flow push and still be able to guarantee convergence of the algorithm. We call such an algorithm the \textit{generic auction algorithm}. It will be seen that the generic algorithm encompasses the $\epsilon$—relaxation algorithm as a special case. As in the case of linear costs, we expect that the analysis of the generic auction algorithm will also give rise to special algorithms suitable for problems with special structure, e.g. for the convex cost transportation problem.

5.1 The Algorithm

We will base our analysis on the definitions given in chapter 4. To keep the presentation complete we recapitulate a few of the basic definitions and then develop the generic auction algorithm for the convex min-cost flow problem. As in section 4.1 we say that a flow-price vector pair satisfies the $\epsilon$—CS conditions if for all $(i,j) \in A$ we have

$$p_i - p_j \in [f^{-}_{ij}(x_{ij}) - \epsilon, f^{+}_{ij}(x_{ij}) + \epsilon].$$
Given a pair \((x, p)\) satisfying \(\epsilon-\text{CS}\) we define, as before, the push list of each node \(i \in \mathcal{N}\) as the following set of arcs:

\[
\left\{(i,j) \mid \frac{\epsilon}{2} < p_i - p_j - f_{ij}^+(x_{ij}) \leq \epsilon \right\} \cup \left\{(j,i) \mid -\epsilon \leq p_j - p_i - f_{ji}^-(x_{ji}) < -\frac{\epsilon}{2} \right\}.
\]

An arc \((i,j)\) (or \((j,i)\)) of the push list of node \(i\) is said to be unblocked if there exists a \(\delta > 0\) such that

\[p_i - p_j \geq f_{ij}^+(x_{ij} + \delta),\]

(or \(p_j - p_i \leq f_{ji}^-(x_{ji} - \delta)\), respectively). An analysis similar to the one in Section 4.2 proves that any arc \((i,j)\) in the push list of a node is unblocked. Furthermore, recall (4.2) that for a push list arc, the supremum of \(\delta\) for which the above relation holds is the flow margin of the arc. Finally, The set of nodes \(\mathcal{N}\) and the set \(\mathcal{A}^*\) containing all unblocked push list arcs oriented in the direction of flow change define the admissible graph \(G^* = (\mathcal{N}, \mathcal{A}^*)\).

In earlier chapters we developed the two basic operations that can be performed, the \(\delta-\text{flow push}\) and the price rise. In the generic auction algorithm, we modify the way the price rise occurs. In particular, for the generic auction algorithm the two basic operations that can be performed are:

- A \(\delta-\text{flow push}\) along an arc \((i,j)\) (or along an arc \((j,i)\)) consists of decreasing the surplus \(g_i\) of node \(i\) by an amount \(\delta \leq g_i\) and increasing the flow of \((i,j)\) by \(\delta\) (or decreasing the flow of \((j,i)\) by \(\delta\), respectively). As a result the surplus \(g_j\) of the opposite node \(j\) increases by \(\delta\).

- A price rise is performed on a strict subset \(I\) of the set of nodes \(\mathcal{N}\). A price rise consists of raising the price of all nodes in \(I\) while leaving unchanged the flow vector and the prices of nodes not in \(I\). The price increment \(\gamma\) is the maximum amount by which the prices of the nodes in \(I\) can be raised without violating the \(\epsilon-\text{CS}\) conditions. In particular \(\gamma\) is the minimum element of the set \(S^+ \cup S^-\).
where

\[ S^+ = \{ p_j - p_i + f^+_i(x_{ij}) + \epsilon \mid (i, j) \in A \text{ such that } i \in I, j \notin I, x_{ij} < u_{ij} \} \]

\[ S^- = \{ p_j - f^-_i(x_{ji}) + \epsilon - p_i \mid (j, i) \in A \text{ such that } i \in I, j \notin I, x_{ji} > l_{ji} \}. \]

If the price increment $\gamma$ is strictly positive, we say that the price rise is substantive and if $\gamma = 0$ we say that the price rise is trivial. Note that if a substantive price rise occurs then the increment $\gamma$ is at least $\frac{\epsilon}{2}$. A trivial price rise does not change either the flow or the price vector; it is introduced to facilitate the presentation.

The generic algorithm can be described as a sequence of $\delta$-pushes and price rise operations. As we did with the $\epsilon$-relaxation algorithms of the previous sections, we assume that when a $\delta$-flow push is performed on an arc $(i, j)$ (or an arc $(j, i)$), the amount $\delta$ of flow pushed is determined as follows:

\[ \delta = \min\{ g_i, \text{ flow margin of arc} \}. \]

Furthermore, we initialize the flow-price pair so that the initial admissible graph is acyclic. The initialization we presented in section 4.2 can be applied for the generic auction algorithm also; given an initial price vector $\bar{p}$ we set the initial arc flow for every arc $(i, j) \in A$ to

\[ \bar{x}_{ij} = \sup\{ \xi \mid f^+_i(\xi) \leq \bar{p}_i - \bar{p}_j - \frac{\epsilon}{2} \}. \]

Following an analysis similar to the one we did in section (4.2) it can be seen that these two operations lead to flow-price pairs that satisfy $\epsilon$-CS. Furthermore, for a feasible problem either a $\delta$-flow push or a price rise is always possible if our current solution is not optimal. The following proposition formalizes and proves these properties.

**Proposition 5.1.1** Let $(x, p)$ be a flow-price pair satisfying $\epsilon$-CS.

(1) The flow-price pair obtained following a $\delta$-push or a price rise operation satis-
(2) Let I be a subset of nodes with strictly positive total surplus. If the sets $S^+$ and $S^-$ are empty, then the optimal distribution problem is infeasible.

**Proof:** (1) Since a flow push changes the flow on an unblocked arc it cannot result in violation of $\epsilon$-CS. If $p$ and $p'$ are the respective price vectors before and after a price rise operation on a set of nodes $I$, respectively, we have that for all arcs $(i,j)$ with $i \in I$ and $j \in I$, or with $i \notin I$ and $j \notin I$, the $\epsilon$-CS conditions hold for $(x,p')$ since it is satisfied for $(x,p)$ and $p_i - p_j = p'_i - p'_j$. For arcs $(i,j)$ with $i \in I$ and $j \notin I$, or with $j \in I$ and $i \notin I$, the $\epsilon$-CS condition is satisfied because of the choice of the price increment $\gamma$.

(2) Since the sets $S^+$ and $S^-$ are empty we must have

$$x_{ij} = u_{ij}, \quad \forall (i,j) \in A \text{ with } i \in I, \ j \notin I.$$  
$$x_{ij} = l_{ij}, \quad \forall (i,j) \in A \text{ with } j \in I, \ i \notin I.$$  

Then the total surplus of the nodes in $I$ is strictly positive which in turn implies that sum of the divergences of the nodes in $I$ exceeds the capacity of the cut $[I, \mathcal{N} \setminus I]$, and the optimal distribution problem is infeasible. Q.E.D.

The above proposition establishes that for a feasible problem, if some node $i$ has positive surplus, then there are two possibilities:

- The push list of $i$ is nonempty and a $\delta$-push is possible.

- The push list of $i$ is empty in which case a price increase is possible (Proposition 5.1.1).

The preceding observations motivate our generic algorithm which starts with a value for $\epsilon$ and a pair $(x,p)$ satisfying $\epsilon$-CS. The algorithm proceeds in iterations until $g_i \leq 0$ for all nodes $i \in \mathcal{N}$. The typical iteration of the generic algorithm is as follows:

**Typical Iteration**
Consider a set $I$ of sources. A finite number of $\delta$-pushes and price rises on the nodes of $I$ are performed and in any order. There should be at least one $\delta$-push but more than one price rise may be performed. For a $\delta$-push performed at a node $i \in I$, the flow increment should also satisfy $\delta < g_i$.

The analysis of Proposition 4.2.4 can be adapted for the generic algorithm to prove that the admissible graph remains acyclic throughout the algorithm. The following proposition establishes this fact:

**Proposition 5.1.2** The admissible graph $G^*$ remains acyclic throughout the generic algorithm.

**Proof:** Initially, the proposition holds trivially since the choice of starting flow and price vector for the algorithm ensure that the initial admissible graph is empty and thus acyclic. Next, we observe that $\delta$-flow pushes can only remove arcs from $A^*$, so that $G^*$ can acquire a cycle only following a price rise. Since following a price rise on a set $I$, the prices of all nodes in $I$ are increased by the same amount $\gamma$, no new arcs are introduced in $G^*$ between nodes in $I$. Thus a cycle must include nodes of $I$ as well as some nodes not in $I$. We claim that, following a price rise on $I$, there are no incoming arcs to any node $i \in I$ in the admissible graph. To see this, assume that after a price rise at node $i$ the arc $(j, i)$ belongs in $G^*$. Then arc $(j, i)$ belongs either in the push list of $j$ or in the push list of $i$. If $(j, i)$ belongs in the push list of $j$ then just before the price rise we had $p_j > p_i + f_{ji}^+(x_{ji}) + \frac{\epsilon}{2}$ and therefore $(j, i)$ cannot be unblocked. The case where $(j, i)$ belongs in the push list of $i$ is treated similarly, establishing the claim. Thus $(j, i)$ cannot be in $G^*$ and no cycles are possible. $\textbf{Q.E.D.}$

Having established the acyclicity of the admissible graph we can now prove that the generic algorithm terminates. The proof is similar to the one in [BC91].

**Proposition 5.1.3** Assume that the problem is feasible. If the increment of each $\delta$-push is given by

$$\delta = \min\{g_i, \text{ flow margin of arc}\},$$

77
then the generic algorithm terminates with a pair \((x, p)\) satisfying \(\epsilon-CS\).

**Proof:** We first make the following observations:

1. The algorithm preserves \(\epsilon-CS\); this is a consequence of proposition (5.1.1).
2. The prices of all nodes are monotonically nondecreasing during the algorithm.
3. Once a node has nonnegative surplus, its surplus stays nonnegative thereafter.
4. If at some time a node has negative surplus then its price has never been increased up to that time and must be equal to its initial price. This is a consequence of (c) above and the assumption that only nodes with nonnegative surplus can perform a price rise.

Suppose in order to reach a contradiction that the algorithm does not terminate. Then, since there is at least one \(\delta\)-flow push per iteration, an infinite number of \(\delta\)-flow pushes must be performed at some node \(m\) along an arc \((m, n)\) or some arc \((n, m)\). Let us assume it is arc \((m, n)\); a similar argument applies when the arc is \((n, m)\). First, we observe that the number of successive nonsaturating \(\delta\)-flow pushes that can be performed along \((m, n)\) is finite. Assume the contrary, that is, arc \((m, n)\) is in the push list of \(m\) and an infinite sequence of successive nonsaturating flow pushes are performed, along arc \((m, n)\) with no intervening saturating flow push. Then the surplus of node \(m\) must be exhausted infinitely often during this sequence. This can happen only if the surplus of some predecessor \(i^1\) of \(i^0\) is exhausted infinitely often during the sequence. Continuing in this manner we construct an infinite succession of predecessor nodes \\(\{i^k\}\). Thus some node in this sequence must be repeated, which is a contradiction since the admissible graph is acyclic. A similar argument holds for the case where \((m, n)\) is in the push list of \(n\). We conclude therefore, that \(g_m > 0\) or \(g_n > 0\) and infinite number of times and the algorithm performs an infinite number of saturating \(\delta\)-flow pushes along the arc \((m, n)\). This means that the prices \(p_m\) or \(p_n\) (or both) must increase by amounts of \(\frac{\delta}{2}\) an infinite number of times. Thus we have \(p_m \to \infty\) or \(p_n \to \infty\), while either \(g_m > 0\) or \(g_n > 0\) at the start of an infinite number of \(\delta\)-flow pushes.
Let $\mathcal{N}^\infty$ be the set of nodes whose prices increase to $\infty$; this set includes at least one of the nodes $m$ and $n$. To preserve $\epsilon$-CS, we must have after a sufficient number of iterations, that for arcs with finite flow bounds

\begin{align*}
x_{ij} &= u_{ij} \quad \text{for all } (i, j) \in \mathcal{A} \text{ with } i \in \mathcal{N}^\infty, j \notin \mathcal{N}^\infty \\
x_{ji} &= l_{ij} \quad \text{for all } (j, i) \in \mathcal{A} \text{ with } i \in \mathcal{N}^\infty, j \notin \mathcal{N}^\infty
\end{align*}

while for arcs with infinite flow bounds

\begin{align*}
x_{ij} &\to \infty \quad \text{for all } (i, j) \in \mathcal{A} \text{ with } i \in \mathcal{N}^\infty, j \notin \mathcal{N}^\infty \\
x_{ji} &\to -\infty \quad \text{for all } (j, i) \in \mathcal{A} \text{ with } i \in \mathcal{N}^\infty, j \notin \mathcal{N}^\infty.
\end{align*}

After some iteration, by (4) above, every node in $\mathcal{N}^\infty$ must have nonnegative surplus, so the sum of surpluses of the nodes in $\mathcal{N}^\infty$ must be positive at the start of the $\delta$-flow pushes where either $g_m > 0$ or $g_n > 0$. It follows using the argument of the proof of proposition (5.1.1(b)) that

\[0 < \sum_{i \in \mathcal{N}^\infty} s_i - \sum_{(i,j) \in \mathcal{A} | i \in \mathcal{N}^\infty, j \notin \mathcal{N}^\infty} x_{ij} + \sum_{(j,i) \in \mathcal{A} | i \in \mathcal{N}^\infty, j \notin \mathcal{N}^\infty} x_{ji},\]

where the values of $x_{ij}$ and $x_{ji}$ are the ones we defined above. For any feasible vector the above relation means that the sum of the divergences of nodes in $\mathcal{N}^\infty$ exceeds the capacity of the cut $[\mathcal{N}^\infty, \mathcal{N} \setminus \mathcal{N}^\infty]$, which is impossible. It follows that there exists no feasible flow vector, contradicting the hypothesis. Thus the algorithm terminates. Since upon termination we have $g_i \leq 0$ for all $i$ and the problem is assumed feasible it follows that $g_i = 0$ for all $i$. \textbf{Q.E.D.}

The requirement of at least one $\delta$-flow push per iteration has now become evident: It limits the number of trivial price rises that can be executed. This is best illustrated in the example found in [Ber91] which we present in figure (5.1.1) for completeness.
Figure 5.1.1: An example illustrating the need for at least one $\delta$-flow push in order to guarantee convergence. In particular, if we alternate price increases at node 1 and node 2 starting with node 1, the algorithm never terminates.

An argument similar to the one in Proposition 4.2.3 can be applied here to prove that the total number of substantive price rises on sets containing a certain source node is at most $2(K + 1)N$. We have the following proposition:

**Proposition 5.1.4 (Number of Price Increases).** Assume that for some $K \geq 1$, the initial price vector $p^0$ for the $\epsilon$-relaxation method satisfies $KeCS$ together with some feasible flow vector $x^0$. Then, in the course of the algorithm the price $p_i$ of any node $i$ with $g_i > 0$ satisfies

$$p_s - p^0_s \leq (K + 1)|H|\epsilon \leq (K + 1)(N - 1)\epsilon.$$  

**Proof:** Consider the pair $(x, p)$ at the beginning of an $\epsilon$-relaxation iteration. Since the surplus vector $g = (g_1, \ldots, g_N)$ is not zero, and the flow vector $x^0$ is feasible, we conclude that for each source $s$ with $g_s > 0$ there exists a sink $t$ with $g_t < 0$ and a simple (i.e., having no cycles) path $H$ from $t$ to $s$ such that:

\begin{align*}
  x_{ij} &> x^0_{ij} \quad \forall (i, j) \in H^+ \quad (5.1.1) \\
  x_{ij} &< x^0_{ij} \quad \forall (i, j) \in H^- \quad (5.1.2)
\end{align*}
where $H^+$ is the set of forward arcs of $H$ and $H^-$ is the set of backward arcs of $H$.

(This can be seen from the Conformal Realization theorem ([Roc84] or [Ber91]) as follows. Consider the following problem: We replace the upper bounds on the flow by $u_{ij} - x_{ij}^0$, the lower bounds by $l_{ij} - x_{ij}^0$, the flow vector by $x^* = x - x^0$ and the $s_i$ by $0$.

The transformation leaves the surplus of every node unchanged and does not change the prices generated by the algorithm. Since $g \neq 0$, by the Conformal Realization Theorem, we can find a path $H$ from a sink $t$ to a source $s$ which conforms to the flow $x$, i.e., the flow on the path is positive for all $(i, j) \in H^+$ and the flow is negative for all $(i, j) \in H^-$. Thus equations (5.1.1) and (5.1.2) follow.)

From Eqs. (5.1.1) and (5.1.2), and the convexity of the functions $f_{ij}$, for all $(i, j) \in A$ we conclude that

$$f_{ij}(x_{ij}) \geq f_{ij}^+(x_{ij}^0) \quad \forall (i, j) \in H^+$$

$$f_{ij}^+(x_{ij}) \leq f_{ij}^-(x_{ij}^0) \quad \forall (i, j) \in H^-$$

Since the pair $(p, x)$ satisfies $\varepsilon$-CS we conclude that

$$p_i - p_j \in [f_{ij}^-(x_{ij}^0) - \varepsilon, f_{ij}^+(x_{ij}) + \varepsilon] \quad \forall (i, j) \in A$$

Similarly, since the pair $(p^0, x^0)$ satisfies $K\varepsilon$-CS we have

$$p_i^0 - p_j^0 \in [f_{ij}^-(x_{ij}^0) - K\varepsilon, f_{ij}^+(x_{ij}^0) + K\varepsilon] \quad \forall (i, j) \in A$$

Combining equations (5.1.3), (5.1.4), (5.1.5), we derive the following relations:

$$p_i - p_j \geq p_i^0 - p_j^0 - (K + 1)\varepsilon \quad \forall (i, j) \in H^+$$

$$p_i - p_j \leq p_i^0 - p_j^0 + (K + 1)\varepsilon \quad \forall (i, j) \in H^-$$
Summing the above equations over all arcs of the path $H$ we get

$$p_t - p_s \geq p_t^0 - p_s^0 - (K + 1)|H|\epsilon,$$  \hspace{1cm} (5.1.8)

where $|H|$ denotes the number of arcs of the path $H$. We observed earlier that if a node is a sink at some time, then its price has remained unchanged since the beginning of the algorithm. Since the path is simple we have that $|H| \leq N - 1$. Therefore, equation (4.2.10) yields

$$p_s - p_s^0 \leq (K + 1)|H|\epsilon \leq (K + 1)(N - 1)\epsilon.$$  \hspace{1cm} (5.1.9)

**Q.E.D.**

A corollary of the above proposition is the following: Since each substantive price increase is at least $\frac{\epsilon}{2}$ (due to the initialization of the price vector and the way price rises occur) we conclude from equation (5.1.9) that the total number of price rises that can be performed for a source node $s$ is at most $2(K + 1)(N - 1)$. Thus, if we choose the sets $I$ so that all of them contain at least one node $i$ with $g_i > 0$ then the algorithm can perform at most $O(KN^2)$ substantive price rises. Then the algorithm operates in a fashion similar to the $\epsilon-$relaxation algorithm we proved earlier.

Furthermore, the algorithm admits a sweep implementation similar to the one for the $\epsilon-$relaxation algorithm. In particular, we maintain a linked list as before. If a price rise involves nodes of a set $I$, then these nodes are placed on the top of the linked list in the order they appeared in the linked list $T$ before the price rise. The position of nodes not in $I$ is not changed. Let $i$ be the node in $I$ ranking lowest in $T$ and $N_i$ be the set of nodes ranking lower $i$ in $T$. Then the next set $I'$ on which we will perform the next generic iteration can be created by picking nodes with positive surplus from $N_i$ in the order they appear in $T$ (highest ranking in $T$, the next higher ranking and so on). If $N_i$ does not contain any nodes with positive surplus then we start from the top of $T$.

The node list $T$ will always be compatible with the partial order induced by the admissible graph in the sense that each node will appear in the list after all its
predecessors. This can be proved with the same argument used for Proposition 4.3.1. Thus a similar analysis as the one done for \( \varepsilon \)-relaxation proves that the complexity bound derived in section (4.3) holds for the generic algorithm where each sets \( I \) contains at least one node with strictly positive surplus.

The generic auction algorithm contains the \( \varepsilon \)-relaxation algorithm as a special case. In particular, if we choose as our set \( I \) to be the singleton \( i \), where \( g_i > 0 \) the generic auction iterations are identical to the \( \varepsilon \)-relaxation iterations.
Chapter 6

The Auction/Sequential Shortest Paths Algorithm

The auction/sequential shortest path (ASSP) algorithm was proposed in [Ber92] in the context of the classical minimum cost flow problem where the arc costs are linear. In this section we will extend it to the case of convex costs.

The $\epsilon$-relaxation algorithm and the generic algorithm analyzed in the previous chapters performed flow pushes and price increases on positive surplus nodes using local information about their incident arcs. The ASSP algorithm operates in a different way; it does not make any flow push until an unblocked path of minimum cost connecting a source and a sink has been found. Once such an unblocked path is found, a flow is pushed along the path from the source to the sink. Thus after such a flow push the surplus of a source decreases and the deficit of a sink decreases, whereas the surpluses of the rest of the nodes in the graph remain unchanged. In contrast the $\epsilon$-relaxation algorithm allowed a node to decrease its own surplus by increasing the surplus of a neighboring node.

6.1 The Algorithm

The algorithm maintains a price vector and a flow vector satisfying the $\epsilon$-Complementary slackness conditions introduced in Section 4. To facilitate the presentation
we review some of our definitions. In particular, the flow vector and the price vector satisfy $\epsilon$--CS if and only if

$$f_{ij}(x_{ij}) - \epsilon \leq p_i - p_j \leq f_{ij}^+(x_{ij}) + \epsilon, \quad \forall (i, j) \in A.$$ 

Given a pair $(x, p)$ satisfying $\epsilon$--CS we define the push list of each node $i \in \mathcal{N}$ as the following set of arcs:

$$\left\{(i, j) \mid \frac{\epsilon}{2} < p_i - p_j - f_{ij}^+(x_{ij}) \leq \epsilon\right\} \cup \left\{(j, i) \mid -\epsilon \leq p_j - p_i - f_{ji}^-(x_{ji}) < -\frac{\epsilon}{2}\right\}.$$ 

An arc $(i, j)$ (or $(j, i)$) is said to be unblocked if there exists a $\delta > 0$ such that

$$p_i - p_j \geq f_{ij}^+(x_{ij} + \delta),$$

(or $p_j - p_i \leq f_{ji}^-(x_{ji} - \delta)$, respectively). For a push list arc, the supremum of $\delta$ for which the above relation holds is called the flow margin of the arc. As we proved in Section 4.2, arcs in the push list of a node are unblocked. We consider the set of arcs $A^*$ which contains all unblocked push list arcs, oriented in the direction of flow change and we refer to the graph $G^* = (\mathcal{N}, A^*)$ as the admissible graph. The notion of the admissible graph will prove useful in the analysis that follows. Finally, we define a price rise on a node $i$ as the operation that increases the price of $i$ by the maximum amount that does not violate $\epsilon$--CS.

We introduce now some concepts that are essential for the ASSP algorithm, specifically, the notion of a path and the operations performed on it. A path $P$ is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ and a corresponding sequence of $k - 1$ arcs such that the $i$th arc on the sequence is either $(n_i, n_{i+1})$ (in which case it is called a forward arc) or $(n_{i+1}, n_i)$ (in which case it is called a reverse arc). We denote by $s(P)$ and $t(P)$ the starting node and the terminal node respectively of path $P$. We also define the sets $P^+$, $P^-$ containing the forward and reverse arcs of $P$, respectively. A path that consists of a single node (and no arcs) is called a degenerate path. A path is called simple if it has no repeated nodes. It is called unblocked if all of its arcs are un-
locked. An unblocked path starting at a source and ending at a sink is called an augmenting path. An augmentation (compare with the definition of δ-flow push we gave in section (4.2)) along such a path starting at a source \( s(P) \) and ending at a sink \( t(P) \), consists of increasing the flow of all the arcs in \( P^+ \), decreasing the flow of all arcs in \( P^- \), decreasing the surplus of the source \( g_s(P) \), and increasing the surplus of the sink \( g_t(P) \) by the same amount

\[
\delta = \min \left\{ g_s(P), -g_t(P), \min \text{ of flow margins of the arcs of } P \right\}. \tag{6.1.1}
\]

We define two operations on a given path \( P = (n_1, n_2, \ldots, n_k) \): A contraction of \( P \) deletes the terminal node of \( P \) and the corresponding terminal arc. An extension of \( P \) by an arc \((n_k, n_{k+1})\) or an arc \((n_{k+1}, n_k)\), replaces \( P \) by the path \( (n_1, n_2, \ldots, n_k, n_{k+1}) \) and adds to \( P \) the corresponding arc.

The ASSP algorithm, described formally below, maintains a flow-price pair satisfying \( \epsilon \)-CS and also a simple path \( P \), starting at some positive surplus node. At each iteration, the path \( P \) is either extended or contracted. In case of a contraction the price of the terminal node of \( P \) is strictly increased. In the case of an extension, no price change occurs, but if the new terminal node has negative surplus then an augmentation along \( P \) is performed. Following an augmentation, \( P \) is replaced by the degenerate path that consists of a single node with positive surplus and the process is repeated. The algorithm terminates when all nodes have nonnegative surplus. Then either all nodes have zero surplus and the flow vector \( x \) is feasible, or else some node has negative surplus showing that the problem is infeasible. Throughout the algorithm the flow-price pair \( (x, p) \) and the path \( P \) satisfy the following:

1. \( \epsilon \)-CS is maintained and the admissible graph corresponding to \( (x, p) \) is acyclic.

2. \( P \) belongs to the admissible graph, i.e., it is unblocked and simple.

The algorithm is initialized in the same way as the \( \epsilon \)-relaxation algorithm, rendering the initial admissible graph acyclic. In particular, given an initial price vector...
\( \bar{p} \) we set the initial arc flow of every arc \((i, j) \in A\) as follows:

\[
\bar{x}_{ij} = \sup \{ \xi \mid f_{ij}(\xi) \leq \bar{p}_i - \bar{p}_j - \varepsilon \}.
\]

As we proved in section (4.2), for these choices \( \varepsilon \)-CS is satisfied and the initial admissible graph is acyclic since its arc set is empty. A typical iteration of the algorithm is as follows:

Typical Iteration of the Action/Sequential Shortest Path Iteration

**Step 1:** Pick a node with positive surplus and let the path \( P \) consist of only this node. If no such node exists then stop.

**Step 2:** Let \( i \) be the terminal node of the path \( P \). If the push list of \( i \) is empty, then go to step 3; else go to step 4.

**Step 3 (Contract Path):** Perform a price rise at \( i \). If \( i \notin s(P) \), contract \( P \). Go to step 2.

**Step 4 (Extend Path):** Pick an arc \((i, j)\) (or \((j, i)\)) from the push list of \( i \) and extend \( P \). If the surplus of \( j \) is negative go to step 5; otherwise go to step 2.

**Step 5 (Augmentation):** Perform an augmentation along the path \( P \) by the amount \( \delta \) defined in (6.2.2). Go to step 1.

We proceed now to establish some basic properties of the algorithm and prove its finite termination.

**Proposition 6.1.1** Suppose that at the start of an iteration:

- \((x, p)\) satisfies \( \varepsilon \)-CS and the corresponding admissible graph is acyclic.
- \( P \) belongs to the admissible graph.

Then the same is true at the start of the next iteration.
Proof: Suppose that the iteration involves a contraction. Then, by definition, the price rise preserves the $\epsilon$-CS conditions. Since only the price of $i$ changed and no arc flow changed, the admissible graph remains unchanged except for the incident arcs to $i$. In particular, all incident arcs of $i$ in the admissible graph at the beginning of the iteration are deleted and the arcs in the push list of $i$ at the end of the iteration are added. Since all these arcs are outgoing from $i$ in the admissible graph, a cycle cannot be formed. Finally, after the contraction, $P$ does not contain the terminal node $i$, so it belongs to the admissible graph before the iteration. Thus $P$ consists of arcs that still belong to the admissible graph after the iteration.

Suppose now that the iteration involves an extension. Since the extension arc $(i, j)$ or $(j, i)$ is unblocked and belongs to the push list of $i$, it belongs to the admissible graph and thus $P$ belongs to the admissible graph after the extension. Since no flow or price changes with an extension, the $\epsilon$-CS conditions and the admissible graph do not change after an extension. If there is a subsequent augmentation because of Step 5, the $\epsilon$-CS conditions are not affected, while the admissible graph will not gain any new arcs, so it remains acyclic. Q.E.D.

The above proof also demonstrates the importance of relaxing the CS conditions by $\epsilon$. If we had taken $\epsilon = 0$ then the preceding proof would break down, and the admissible graph might not remain acyclic after an augmentation. For example, if following an augmentation, the flow of some arc $(i, j)$ with linear cost and flow margin $\delta_{ij}$ lies strictly between the flow $x_{ij}$ it had before the augmentation and $x_{ij} + \delta_{ij}$, then both the arc $(i, j)$ and the arc $(j, i)$ would belong to the admissible graph closing a cycle.

We will now proceed to prove the validity of the algorithm. First, we ensure that a path from a source to some sink can be found. The sequence of iterations between successive augmentations (or the sequence of iterations up to the first augmentation) will be called an augmentation cycle. Let us fix an augmentation cycle and let $\hat{p}$ be the price vector at the start of this cycle. Let us now define an arc set $A_R$ by introducing for each arc $(i, j) \in A$, two arcs in $A_R$: an arc $(i, j)$ with length $f_{ij}^+(x_{ij}) + \hat{p}_j - \hat{p}_i + \epsilon$ and an arc $(j, i)$ with length $\hat{p}_i - f_{ij}^-(x_{ij}) - \hat{p}_j + \epsilon$. The resulting graph $G_R = (\mathcal{N}, A_R)$
will be referred to as the reduced graph. Note that because the pair \((x, p)\) satisfy \(\epsilon\)-CS the arc lengths of the reduced graph are nonnegative. Furthermore, the reduced graph contains no zero length cycles since such a cycle would belong to the admissible graph which we proved to be acyclic. During an augmentation cycle the reduced graph remains unchanged, since no arc flow changes except for the augmentation at the end.

It can now be seen that the augmentation cycle is just the auction shortest path algorithm of [Ber91] and it constructs a shortest path in the reduced graph \(G_R\) starting at a source \(s(P)\) and ending at a sink \(t(P)\). By the theory of the auction shortest path algorithm, a shortest path in the reduced graph from a source to some sink will be found if one exists. Such a path will exist if the convex cost flow problem is feasible.

We now observe that the number of price rises that can be performed by our algorithm is bounded. Proposition (4.2.3) applies to our algorithm without any change. In particular, if the initial price vector \(p^0\) for the ASSP algorithm satisfies \(K\epsilon\)-CS with some feasible flow vector \(x^0\) for some \(K \geq 1\), then the algorithm performs \(O(KN)\) price increases per node of positive surplus.

Finally, we have to bound the total number of augmentations the algorithm performs between two successive price rises (not necessarily at the same node). First we observe that there can be at most \(N\) successive extensions before either a price rise occurs or an augmentation is possible. We say that an augmentation is saturating if the flow increment is equal to the flow margin of at least one arc of the path. The augmentation is called exhaustive if, after it is performed, the surplus of the starting node \(s(P)\) becomes zero (compare with similar definitions we gave in section (4.2)). An augmentation cannot introduce new arcs in the admissible graph \(G^*\) since a saturating augmentation along a path \(P\) removes from \(G^*\) all the arcs of \(P\) whose flow margin is equal to the flow increment. Thus there can be at most \(O(A)\) saturating augmentations. Furthermore, if a node has zero surplus at some point in the algorithm, then its surplus remains zero for all subsequent iterations of the algorithm. Thus there can be at most \(O(N)\) exhaustive flow pushes. Therefore at most \(O(NA)\) augmentations can be performed between successive prices rises. Thus the algorithm terminates with a feasible flow vector, and a flow-price pair satisfying
the $\epsilon$--CS conditions.

The algorithm as described has a complexity similar to the $\epsilon$--relaxation without the sweep implementation. To get a better complexity bound for the algorithm we could modify the way it operates. For example, we consider a hybrid algorithm which performs $\epsilon$--relaxation iterations according to the sweep implementation along with some additional ASSP iterations. This hybrid algorithm points out the issues that may lead to an efficient implementation of the ASSP algorithm. In particular, all the nodes are kept in a linked list $T$ which is traversed from the first to the last element. The initial list is arbitrary. During the course of the algorithm, the list is updated as follows: Whenever a price rise occurs at a node, the node is removed from its current position at the list and is placed at the first position of the list. Furthermore, we perform a fixed number $C$ of ASSP iterations at the end of which we perform an $\epsilon$--relaxation iteration at $s(P)$ and pick a new source from $T$. Let $N_{s(P)}$ contain the nodes of $T$ that are lower than $s(P)$ in $T$. The next node to be picked for an iteration is the node $i' \in N_{s(P)}$ with positive surplus which ranks highest in $T$, if such a node exists. Otherwise, we declare that a cycle of the algorithm has been completed and the positive surplus node ranking highest in $T$ is picked.

This implementation of ASSP is in essence the sweep implementation of the $\epsilon$--relaxation algorithm with some additional (at most $C$ at a time) iterations of the ASSP algorithm. In particular, we note that a cycle for the hybrid algorithm, is the set of iterations whereby all nodes of positive surplus in $T$ performed either an $\epsilon$--relaxation iteration or at least one a price rise. Whenever a price increase occurs on a node $i$, then $i$ can have no predecessors and is placed at the top of the list. Thus, the nodes appear in the list before all their predecessors. Furthermore, if no price rise has occurred during a cycle, then all nodes with positive surplus have performed an $\epsilon$--relaxation iteration and Proposition 4.3.2 applies. We conclude that from the analysis of Section 4.3 that the hybrid algorithm has a similar complexity bound with the $\epsilon$--relaxation algorithm with the sweep implementation.

Although such an implementation looks cumbersome -- after all why not just use the $\epsilon$--relaxation algorithm -- the main idea is of importance in practice and will be
used in actual implementations of the algorithm. In particular, if the ASSP algorithm has made numerous iterations with some starting node $s(P)$ without finding a path to a sink, we may deduce that the path to some sink is "long" and many more iterations may be needed for the algorithm to find it. We can make an augmentation along our current path and pick a new node on which to iterate. Thus we may benefit in two ways. First, if the terminal node of the path is not a source, then, after the augmentation, we have a source potentially closer to a sink than $s(P)$. Secondly, we allow the algorithm to pick a new source on which to iterate, enabling sources that are close to sinks to find unblocked paths first. It has been shown [Pol94] that such an implementation is successful in the case of linear costs, especially when the paths from sources to sinks are unusually long.

At this point we will briefly compare the $\epsilon$–relaxation algorithm with the ASSP algorithm. As we noted in the opening of this section, the ASSP algorithm performs augmentations only when a path connecting a source and a sink is found. Thus augmentations do not affect the surpluses of intermediate nodes of the path; as a result when a node has zero surplus at some iteration, its surplus is not changed in subsequent iterations. In contrast, a $\delta$–flow push of the $\epsilon$–relaxation algorithm is performed "locally" and there is no guarantee whether the flow is moving in the right direction, i.e., towards a sink; this may cause flow to be pushed back and forth between neighboring nodes before any progress is made. The ASSP makes a more effective flow augmentation but at a price: there may be a lot of extensions and contractions before a path from a source to a sink can be found and an augmentation can be made. For problems where there exist paths with many intermediate nodes from sources to sinks (e.g., grid graphs with large diameters) the ASSP algorithm may prove more efficient than the $\epsilon$–relaxation algorithm. Since the flow has to follow long paths anyway, it is better to make a flow push when such a path is known than to waste time by pushing flow in the "wrong" direction.

In the next section, we will formulate the reverse ASSP algorithm, which maintains a path ending at a sink and performs iterations at the starting node of the path. We also develop a combined algorithm which operates on nodes of non-zero surplus.
6.2 The Reverse and Combined ASSP Algorithm

The ASSP algorithm we presented in the previous section performed iterations only on nodes with positive surplus; we call it the forward algorithm. We can modify the forward algorithm so that iterations can be performed on nodes with negative surplus; the resulting algorithm is called the reverse algorithm. In the sequel we give a number of definitions similar to the ones we gave for the forward algorithm. Given a pair \((x, p)\) satisfying \(\epsilon-\text{CS}\) we define for each node \(i \in \mathcal{N}\) its pull list as the following set of arcs:

\[
\left\{(j, i) \mid \frac{\epsilon}{2} < p_j - p_i - f^+_{ji}(x_{ji}) \leq \epsilon\right\} \cup \left\{(i, j) \mid -\epsilon \leq p_i - p_j - f^-_{ji}(x_{ji}) < -\frac{\epsilon}{2}\right\}.
\]

An arc \((j, i)\) (or \((i, j)\)) is said to be unblocked if there exists a \(\delta > 0\) such that

\[p_j - p_i \geq f^+_{ji}(x_{ji} + \delta),\]

(or \(p_i - p_j \leq f^-_{ij}(x_{ij} - \delta)\), respectively). For a pull list arc, the supremum of \(\delta\) for which the above relation holds is called the flow margin of the arc. With a similar argument to the one in Section 4.2 we conclude that arcs in the pull list of a node are unblocked. Observe that our definition of the pull list of a node does not affect the definition of an unblocked arc or of the flow margin of an arc. In particular, we observe that an arc \((i, j)\) (or \((j, i)\)) is in the push list of node \(i\) if and only if arc \((i, j)\) (or \((j, i)\) respectively) is in the pull list of node \(j\). Furthermore, the direction of flow change on arc \((i, j)\) (or \((j, i)\)) is the same regardless of whether we consider the arc as being in the push list of \(i\) or as being in the pull list of \(j\). Finally, as a result of our definitions, an arc \((i, j)\) (or \((j, i)\)) cannot be simultaneously in the push and pull list of node \(i\). Thus the set of nodes \(\mathcal{N}\) and the set \(\mathcal{A}^*\) containing all unblocked pull list arcs oriented in the direction of flow change define the admissible graph \(G^* = (\mathcal{N}, \mathcal{A}^*)\). We also define a price decrease on a node \(i\): It is the operation that decreases the price of \(i\) by the maximum amount that does not violate \(\epsilon-\text{CS}\).

The definition of the path is identical to the one we gave for the forward algorithm.
We repeat it here for completeness. A path $P$ is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ and a corresponding sequence of $k - 1$ arcs such that the $i$th arc on the sequence is either $(n_i, n_{i+1})$ (in which case it is called \textit{forward} arc) or $(n_{i+1}, n_i)$ is which case it is called \textit{reverse} arc). We denote by $s(P)$ and $t(P)$ the starting node and the terminal node respectively of path $P$. Thus the definitions of a degenerate, unblocked and simple path respectively and of the sets $P^+$ and $P^-$ are identical to the ones in section (6.1). An unblocked path starting at a source and ending at a sink is called an \textit{augmenting path}. An \textit{augmentation} along such a path starting at a source $s(P)$ and ending at a sink $t(P)$, consists of increasing the flow of all the arcs in $P^+$, decreasing the flow of all arcs in $P^-$, decreasing the surplus of the source $(g_s(P))$, and increasing the surplus of the sink $g_t(P)$ by the same amount

$$\delta = \min \left\{ g_s(P), -g_t(P), \min \text{ flow margins of the arcs of } P \right\}. \quad (6.2.2)$$

So far there have been no differences with the forward algorithm in the various definitions related to a path. We now define the two operations, contraction and extension, that are performed on the path $P = (n_1, n_2, \ldots, n_k)$ in a slightly different manner. The difference with the forward algorithm is that the operations occur at the starting node rather than the terminal node of the path. Thus a \textit{contraction} of $P$ deletes the starting node of $P$ and the corresponding starting arc. An \textit{extension} of $P$ by an arc $(n_1, n_0)$ or an arc $(n_0, n_1)$, replaces $P$ by the path $(n_0, n_1, n_2, \ldots, n_k)$ and adds to $P$ the corresponding arc.

The reverse ASSP algorithm, described formally below, maintains a flow-price pair satisfying $\epsilon$-CS and also a simple path $P$, ending at some negative surplus node. At each iteration, the path $P$ is either extended or contracted. In case of a contraction, the price of the starting node of $P$ is strictly decreased. In the case of an extension, no price change occurs, but if the new terminal node has positive surplus then an augmentation along $P$ is performed. Following an augmentation, $P$ is replaced by the degenerate path that consists of a single node with negative surplus and the process is repeated. The algorithm terminates when all nodes have nonnegative surplus. Then
either all nodes have zero surplus and the flow vector \( x \) is feasible, or else some node has positive surplus showing that the problem is infeasible. Throughout the algorithm the flow-price pair \((x, p)\) and the path \( P \) satisfy the following:

(1) CS is maintained and the admissible graph corresponding to \((x, p)\) is acyclic.

(2) \( P \) belongs to the admissible graph, i.e., it is unblocked and simple.

The algorithm is initialized in the same way as the \(\epsilon\)-relaxation algorithm; the initial admissible graph is therefore acyclic. In particular, given an initial price vector \( \bar{p} \) we set the initial arc flow for every arc \((i, j) \in A\) to

\[
\bar{x}_{ij} = \sup\{\xi | f^+_{ij}(\xi) \leq \bar{p}_i - \bar{p}_j - \frac{\epsilon}{2}\}.
\]

As we proved in section (4.2), for these choices \(\epsilon\)-CS is satisfied and the initial admissible graph is acyclic since its arc set is empty. We proceed now to describe a typical iteration of the reverse algorithm.

*Typical Iteration of the Reverse ASSP Iteration*

**Step 1:** Pick a node with negative surplus and let the path \( P \) consist of only this node. If no such node exist then stop.

**Step 2:** Let \( i \) be the starting node of the path \( P \). If the pull list of \( i \) is empty, then go to step 3; otherwise, go to step 4.

**Step 3 (Contract Path):** Perform a price decrease at \( i \). If \( i \neq t(P) \), contract \( P \). Go to step 2.

**Step 4 (Extend Path):** Pick an unblocked arc \((i, j)\) (or \((j, i)\)) from the pull list of \( i \) and extend \( P \). If the surplus of \( j \) is positive go to step 5; otherwise got to step 2.

**Step 5 (Augmentation):** Perform an augmentation along the path \( P \) by the amount \( \delta \) defined in (6.2.2). Go to step 1.
As we see, the algorithm performs extensions, contractions and augmentations on a path rooted at a sink. It is in every respect the “mirror image” of the forward algorithm. Therefore, the same analysis applies.

It is possible to combine the forward and the reverse algorithm, resulting in an algorithm which operates on both positive and negative surplus nodes. Our intuition is that if we perform $\epsilon$-relaxation iterations on both positive and negative surplus nodes, we will be able find the optimal solution faster for certain classes of problems, since the search for augmenting paths is done both from sources and from sinks. For the case of the traditional shortest path problem, the two-sided algorithm was dramatically faster, requiring many fewer less iterations to find a shortest path. This is because the forward algorithm can take advantage of the prices set by the reverse and vice-versa. We expect similar behavior for the ASSP algorithm since, as we showed earlier (section (6.1)), the algorithm finds a shortest augmenting path in the reduced graph. We refer to the resulting algorithm as the *combined algorithm*. We initialize the arc flows and node prices the same way we initialized them for the forward and the reverse algorithms, so that the initial admissible graph is acyclic. The combined algorithm operates as follows:

*Typical Iteration of the Combined Algorithm*

Pick a node $i$ with nonzero surplus; if no such node exists then terminate.

If $i$ has positive surplus then perform several forward ASSP iterations. If $i$ has negative surplus then perform several reverse ASSP iterations.

The combined algorithm maintains the acyclicity of the admissible graph since both its constituent algorithms (the forward and the reverse) do so. Based on our earlier analysis of the forward algorithm, the number of flow pushes that the combined algorithm can perform before a price change (increase or decrease) can occur at a node are bounded. However, termination is not easily guaranteed for the algorithm described thus far. The reason is that a naive combination of the forward and reverse algorithms may fail to find any path from an origin to a destination. In particular, the prices of some nodes may be increased by some forward ASSP iteration and then
decreased to the level before the forward iteration by a subsequent reverse ASSP iteration. Thus prices may oscillate and the algorithm may not terminate. The reader is referred to [Pol92] where an example of such oscillations is presented for the case of the traditional shortest path problem. To guarantee termination, we have to ensure that a new node is not picked unless either an augmentation takes place or a price increase at the starting node of a forward path or a price decrease at the ending node of a reverse path is executed. Since the admissible graph remains acyclic throughout the algorithm, we conclude that the number of possible $\delta$-flow pushes between successive price rises (not necessarily at the same node) is bounded. We only need to establish that the number of price rises that can be performed on each node is bounded. This can be proved with an analysis identical to the one in proposition (4.2.3) except for the fact that the prices of the sinks decrease. In particular, assume that the initial price vector $p^0$ for the combined ASSP algorithm satisfies $K\epsilon-$CS with some feasible flow vector $x^0$ for some $K \geq 1$. Then with an analysis similar to the one of (4.2.3) we conclude that (compare eq. (4.2.10))

$$p_t - p_s \geq p_t^0 - p_s^0 - (K+1)N\epsilon,$$  \hspace{1cm} (6.2.3)

where $(x,p)$ is the flow-price pair at the beginning of a combined ASSP iteration. However, for every sink node $t$ we know that $p_t - p_t^0 \leq 0$, and we conclude that

$$p_s - p_s^0 \leq (K+1)N\epsilon.$$

Since a price rise operation increases the price of a node by at least $\frac{\epsilon}{2}$ we conclude that the total number of price increases is bounded. A similar argument proves that the number of possible price decreases that can be performed is finite. Thus the combined algorithm terminates. Finally, we note that a one origin-many destinations combined scheme similar to the one developed for the standard shortest path problem ([Ber91], [Pol92]) may be used to find many augmenting paths from a source to several sinks.
6.3 Computational Results

We developed the experimental code SSP-NE which implements the ASSP algorithm and tested it against the implementations of the $\epsilon$-relaxation algorithms of the previous chapters and the relaxations methods of [BHT87]. Our implementation was written in FORTRAN and was based on the program written for the linear cost algorithm.

Several changes and enhancements were introduced for the convex cost implementation: All computations involve real numbers rather than integers and no arc cost scaling is performed. Furthermore, the computations involving the push lists and price increases are different. Finally, the termination criterion is different: The linear cost algorithms for integer costs terminate when the value of $\epsilon$ is less than $\frac{1}{(n+1)}$, whereas the convex cost algorithms terminate when the primal and dual costs agree on a user specified number of digits. Initial testing on linear cost problems was performed using both the linear and the convex cost codes. Those tests showed that the convex cost implementation performed equally well to its linear counterpart, indicating that the convex cost algorithm was coded efficiently.

We tested SSP-NE against the $\epsilon$-relaxation implementations (NE-RELAX-F and NE-RELAX-FB) and the relaxation implementations (NRELAX and MNRELAX of [BHT87]) on a variety of problems. All FORTRAN programs were compiled on a SUN SPARC-5 with 24 megabytes of RAM under the SOLARIS operating system. We also used the -O compiler optimization switch in order to take advantage of the floating point unit and the design characteristics of the SPARC-5 processor.

The test problems were created by two problem generators: (i) the public domain code NETGEN, by D. Klingman, A. Napier and J. Stutz [KNS74], which generates feasible assignment/transportation/transshipment problems with random structure and linear costs. We modified the created problems so that a user-specified percentage of the arcs had an additional quadratic cost coefficient within a user-specified range; (ii) a generator that we wrote called CHAINGEN. This generator creates a long chain involving all the nodes. The user also specifies the additional number of arcs.
per node that should be generated. For example, if the user specifies 3 additional arcs per node \(i\) then the arcs \((i, i + 2), (i, i + 3), (i, i + 4)\) are generated with random linear costs. Furthermore, the user can specify the percentage of nodes \(i\) for which an additional arc \((i, i - 1)\) is generated with random linear cost. The resulting graphs have long diameters and our testing with linear cost algorithms has shown that these are particularly difficult problems for all methods. We modified the created problems so that a user specified percent of the arcs had an additional quadratic cost coefficient in a user-specified range.

Our testing revolved around three main issues:

(a) the relative efficiency of the ASSP algorithm versus the \(\epsilon\)–relaxation methods and the relaxation methods;

(b) the insensitivity of the ASSP algorithm to ill-conditioned problems. For our tests, ill-conditioning appears when some of the arcs have large quadratic cost coefficients and some have very small quadratic cost coefficients;

(c) the effect of different graph architectures on the relative performance of the tested algorithms.

To better compare our results with those reported in [BHT87] we generated the same NETGEN problems as in Table 6.1 of [BHT87]. These problems are described in Table 1 below. The computational results that we obtained are shown in Table 6.2. Our testing indicates that the ASSP algorithm performs a bit better than the \(\epsilon\)–relaxation algorithm and is consistently faster than the relaxation algorithms of [BHT87] on all the problems of Table 1. On average, the ASSP algorithm are about 3 times faster than the relaxation algorithms with a maximum of about 8 times faster running time.

The effect of ill-conditioning on the algorithms of this paper was demonstrated with the test problems described in Table 6.3. We generated problems which are increasingly ill-conditioned and the results of our testing are shown in Table 6.4. As expected, the NRELAX code was affected dramatically by the amount of ill-conditioning and its performance was quite poor. The MNRELAX code was relatively insensitive to ill-conditioning which agrees with the results reported in [BHT87].
\(\epsilon\)-relaxation implementations were particularly successful on our test problems as reported earlier in the thesis. The ASSP algorithm and the \(\epsilon\)-relaxation method performed equally well and for some of the test problems (Table 2) the ASSP method outperformed the \(\epsilon\)-relaxation method. However, on the ill-conditioned problems and the CHAINGEN problems the \(\epsilon\)-relaxation was slightly better than the ASSP. In any case however, not only was the performance of ASSP almost unaffected by ill-conditioning, but its running time was consistently faster than the MNRELAX code by at least a factor of 3.

Finally, the effect of graph architecture on the relative efficiency of the algorithms of this paper was demonstrated by the CHAINGEN problems described in Table 6.5. Our experience with the linear cost algorithms was that the ASSP algorithm is well suited for graphs involving long paths. The results of our testing are shown in Table 6.6. These problems involve long paths from the source to the sink and thus are quite difficult. The ASSP algorithm performed slightly worse than the \(\epsilon\)-relaxation implementations and outperformed the MNRELAX by at least an order of magnitude. Our conclusion is that the methods of this chapter are very successful in solving the convex cost network flow problem and in several cases faster than the \(\epsilon\)-relaxation methods of earlier chapters.
<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Nodes</th>
<th>Arcs</th>
<th>Linear Cost</th>
<th>Quad Cost</th>
<th>Total Surplus</th>
<th>Capacity Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob1</td>
<td>200</td>
<td>1300</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob2</td>
<td>200</td>
<td>1500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob3</td>
<td>200</td>
<td>2000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob4</td>
<td>200</td>
<td>2200</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob5</td>
<td>200</td>
<td>2900</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob6</td>
<td>300</td>
<td>3150</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob7</td>
<td>300</td>
<td>4500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob8</td>
<td>300</td>
<td>5155</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob9</td>
<td>300</td>
<td>6075</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob10</td>
<td>300</td>
<td>6300</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob11</td>
<td>400</td>
<td>1500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob12</td>
<td>400</td>
<td>2250</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob13</td>
<td>400</td>
<td>3000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob14</td>
<td>400</td>
<td>3750</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob15</td>
<td>400</td>
<td>4500</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob16</td>
<td>400</td>
<td>1306</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob17</td>
<td>400</td>
<td>2443</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob18</td>
<td>400</td>
<td>1416</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob19</td>
<td>400</td>
<td>2836</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob20</td>
<td>400</td>
<td>1382</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob21</td>
<td>400</td>
<td>2676</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[100-500]</td>
</tr>
<tr>
<td>prob22</td>
<td>1000</td>
<td>3000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[1000-2000]</td>
</tr>
<tr>
<td>prob23</td>
<td>1000</td>
<td>5000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[1000-2000]</td>
</tr>
<tr>
<td>prob24</td>
<td>1000</td>
<td>10000</td>
<td>[1-100]</td>
<td>[5,10]</td>
<td>10000</td>
<td>[1000-2000]</td>
</tr>
</tbody>
</table>

Table 6.1: The NETGEN problems with all arcs having quadratic coefficients. The problems prob1-prob17 are identical to the problems 1-17 of Table 1 of [BHT87]. The problems named prob18, prob19, prob20, prob21 correspond to the problems 20, 23, 24, 25, respectively, of Table 1 of [BHT87].
<table>
<thead>
<tr>
<th>Problem</th>
<th>NRELAX</th>
<th>MNRELAX</th>
<th>NE-RELAX-F</th>
<th>NE-RELAX-FB</th>
<th>SSP-NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob1</td>
<td>7.95</td>
<td>6.0</td>
<td>1.95</td>
<td>1.72</td>
<td>1.09</td>
</tr>
<tr>
<td>prob2</td>
<td>7.55</td>
<td>6.35</td>
<td>2.13</td>
<td>1.79</td>
<td>1.27</td>
</tr>
<tr>
<td>prob3</td>
<td>2.88</td>
<td>5.65</td>
<td>2.13</td>
<td>1.79</td>
<td>0.707</td>
</tr>
<tr>
<td>prob4</td>
<td>20.45</td>
<td>10.62</td>
<td>2.4</td>
<td>2.02</td>
<td>1.42</td>
</tr>
<tr>
<td>prob5</td>
<td>2.32</td>
<td>24.8</td>
<td>1.45</td>
<td>1.2</td>
<td>1.13</td>
</tr>
<tr>
<td>prob6</td>
<td>7.31</td>
<td>22.11</td>
<td>2.71</td>
<td>1.6</td>
<td>1.43</td>
</tr>
<tr>
<td>prob7</td>
<td>7.52</td>
<td>21.12</td>
<td>3.94</td>
<td>2.6</td>
<td>1.69</td>
</tr>
<tr>
<td>prob8</td>
<td>48.3</td>
<td>26.72</td>
<td>3.88</td>
<td>2.06</td>
<td>2.7</td>
</tr>
<tr>
<td>prob9</td>
<td>7.25</td>
<td>22.71</td>
<td>3.22</td>
<td>2.48</td>
<td>2.54</td>
</tr>
<tr>
<td>prob10</td>
<td>4.41</td>
<td>31.53</td>
<td>2.7</td>
<td>3.3</td>
<td>3.02</td>
</tr>
<tr>
<td>prob11</td>
<td>69.25</td>
<td>15.07</td>
<td>8.79</td>
<td>7.8</td>
<td>4.88</td>
</tr>
<tr>
<td>prob12</td>
<td>17.68</td>
<td>17.24</td>
<td>4.98</td>
<td>4.3</td>
<td>2.91</td>
</tr>
<tr>
<td>prob13</td>
<td>22.00</td>
<td>20.43</td>
<td>7.3</td>
<td>6.00</td>
<td>4.14</td>
</tr>
<tr>
<td>prob14</td>
<td>13.2</td>
<td>24.3</td>
<td>3.03</td>
<td>1.82</td>
<td>2.93</td>
</tr>
<tr>
<td>prob15</td>
<td>10.01</td>
<td>35.99</td>
<td>7.42</td>
<td>6.23</td>
<td>4.11</td>
</tr>
<tr>
<td>prob16</td>
<td>85.10</td>
<td>25.46</td>
<td>8.64</td>
<td>8.4</td>
<td>4.87</td>
</tr>
<tr>
<td>prob17</td>
<td>31.63</td>
<td>21.52</td>
<td>7.38</td>
<td>6.24</td>
<td>4.14</td>
</tr>
<tr>
<td>prob18</td>
<td>7.51</td>
<td>9.03</td>
<td>0.96</td>
<td>0.94</td>
<td>0.91</td>
</tr>
<tr>
<td>prob19</td>
<td>45.43</td>
<td>26.76</td>
<td>8.63</td>
<td>7.84</td>
<td>5.07</td>
</tr>
<tr>
<td>prob20</td>
<td>79.96</td>
<td>17.71</td>
<td>9.95</td>
<td>8.43</td>
<td>7.5</td>
</tr>
<tr>
<td>prob21</td>
<td>33.48</td>
<td>23.97</td>
<td>6.8</td>
<td>5.88</td>
<td>4.11</td>
</tr>
<tr>
<td>prob22</td>
<td>64.42</td>
<td>50.94</td>
<td>8.46</td>
<td>4.11</td>
<td>2.44</td>
</tr>
<tr>
<td>prob23</td>
<td>26.7</td>
<td>49.06</td>
<td>4.08</td>
<td>3.5</td>
<td>4.3</td>
</tr>
<tr>
<td>prob24</td>
<td>26</td>
<td>323.296</td>
<td>5.53</td>
<td>5.58</td>
<td>5.23</td>
</tr>
</tbody>
</table>

Table 6.2: Computational Results on a Sun Sparc 5 with 24MB memory. Running times are in seconds.
Table 6.3: The NETGEN problems with all arcs having quadratic coefficients. Half of the arcs have costs in the range [5,10] and the remaining half have the small quadratic coefficient indicated. The problems prob6 and prob12 are mixed convex problems where half of the arcs have linear cost and the remaining half have quadratic costs.

Table 6.4: Computational Results on a Sun Sparc 5 with 24MB memory. For the very ill-conditioned problems NRELAX was taking extremely long running time. For these cases we reduced the accuracy of the answer given by NRELAX in order to get a meaningful result. The numbers in parentheses indicate the number of significant digits of accuracy of the answer given by NRELAX. The running times of NE-RELAX-F, NE-RELAX-FB, and SSP-NE for the mixed convex problems prob6 and prob12 are included to demonstrate the fact that these methods are not affected significantly by increasing ill-conditioning.
Table 6.5: The CHAINGEN problems with all arcs having quadratic coefficients in the range [5,10]. Half of the nodes have an additional reverse arc.

Table 6.6: Computational Results on a Sun Sparc 5 with 24MB memory. For these problems the NRELAX was taking extremely long running times even for 5 digits of accuracy. For this reason we are not reporting any running times for NRELAX on these problems. MNRELAX also was taking a very long time for the last problem and was stopped before termination. This is indicated by the > sign.
Chapter 7

Conclusion

This thesis has proposed $\epsilon-$relaxation and auction algorithms for the convex cost network flow problem with favorable complexity bounds. It was shown that the approach to solving the problem by introducing the $\epsilon-$complementary slackness conditions and decoupling the price from the flow changes is a powerful one leading to a variety of algorithms with impressive practical performance. This work extends the algorithmic framework developed for the linear cost problems in the pioneering work of [Ber79] and the subsequent publications we have cited in our presentation. This corpus of work on the linear $\epsilon-$relaxation and auction algorithms along with our contributions on the convex cost problem has led to an improved understanding of the algorithmic framework and insight about its connection with existing approaches.

In particular, we have seen that by introducing the $\epsilon-$CS conditions, we have been able to quantize the amount of price rises. This was combined with the idea of maintaining an acyclic admissible graph in order to prove not only the finite termination of the algorithm but a performance guarantee as well, which compares favorably with existing state-of-the-art algorithms. Furthermore, the computational results we obtained for these algorithms on different types of graphs are consistent with the complexity analysis and indicate that for the convex cost problem the methods of this thesis have improved the state-of-the-art by a substantial margin.

Further research is needed to understand the reasons that make $\epsilon-$relaxation more successful than methods that find directions of descent like the relaxation methods. In
particular, for the linear cost case the relaxation methods are often much better than the $\epsilon$-relaxation counterparts whereas for the convex cost case the opposite is true. There seems to be a certain switching point in performance based on the number of linear cost arcs versus the number of strictly convex cost arcs in a graph. A related subject is to test computationally the algorithms proposed on problems with special structure such as the transportation problem. After streamlining the computations we may obtain a scheme that avoids unnecessary operations and solves the problem efficiently. Finally, further testing may be needed with other existing methods like the alternating directions method of [EcF94] or the methods that explicitly linearize the arc costs ([Mey79], [KaM84], and [Roc84]).

Another research issue has to do with the apparent relationship between the $\epsilon$-relaxation algorithms and the proximal minimization algorithm. In particular, for the linear cost problem Tseng in some unpublished work ([Ts89]) has shown how the $\epsilon$-relaxation iterations can be viewed as a dual coordinate ascent method for a problem where the cost function $F(x)$ has been augmented by a quadratic penalty function so that the resulting cost is of the form $F(x) + \alpha||x||^2$. His analysis shows that the two algorithms are related and that a complexity bound – not as favorable as for the pure $\epsilon$-relaxation – can be obtained for the dual coordinate ascent algorithm. At the current state of our research, we believe that the analysis can be extended to encompass the convex cost problem. This is an interesting problem from a purely theoretical standpoint since it places under a common framework the $\epsilon$-relaxation algorithms and standard methods of convex cost optimization. Furthermore, we believe that such an analysis can become the basis for a connection of the $\epsilon$-relaxation algorithm with the proximal minimization algorithm (see [BeT89]). The proximal minimization algorithm has been quite efficient in practice and well-suited for parallel computation ([EcF94]). If a relationship with $\epsilon$-relaxation can be proved, then a global type of convergence rate can be obtained for the proximal point algorithm applied to the linear cost problem. The global convergence rate analysis in Maugaskarian et.al. ([MaS87]) as well as Eckstein ([Ec89]) seem to point to a methodology that would unify the two algorithms.
Bibliography


[GlM75] Glowinski, R., Marocco, A., “Sur l’approximation par elements finis d’ordre un et la resolution par penalisation-dualite d’une classe de problemes de Dirichlet non lineaires,” Revue Francaise d’ Automatique In-


[Ts89] Tseng, P., Unpublished work.

