National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

ISBN 0-315-83809-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Using Interior Point Methods to Solve the Multicommodity Network Flow Problem

by

Christopher B. Browne

Submitted to the Faculty of Administration
on April 8, 1993, in partial fulfillment of the
requirements for the degree of
Master of Systems Science

Abstract

This thesis explores applications of Interior Point methods as popularized by Karmarkar [36] for solving Multicommodity Network Flow problems (MCNF). In these problems, several commodities must be shipped between various nodes of a network. The goal is to satisfy the shipping requirements at a minimum cost, while respecting processing capacities on the joint flow of commodities.

The thesis presents a unified view of current methods for multicommodity networks, from the early formulations to current work that has involved nonlinear and interior point methods. It compares and contrasts some Interior Point methods, and discusses how they can be applied to the MCNF problem.

It builds upon the first Interior Point partitioning method proposed to solve MCNF, and implements a related partitioning method. This scheme allows network subproblems to be solved using efficient network simplex algorithms: then the results are coordinated using a linear program that is solved with an Interior Point affine scaling algorithm. It presents computational results, comparing the results from the partitioning method with those found by solving an LP formulation of the problem using the affine scaling method.

Thesis Supervisor: Jean-Michel Thizy
Title: Associate Professor, Faculty of Administration

Contents

Acknowledgments

Thank you to those who assisted directly in the preparation of this thesis. Between professors and students, I needed help many times, and got it. Many thanks to Professor Thizy, who has put in a lot of hours.

I'd like to thank my parents for putting up with me while I worked on this paper. There have been some late nights, and times when I have been so much into the math that my communications skills have disappeared. My parents have been very patient.

My brothers also need thanking. They often had no idea what I was up to, and were reasonably forgiving when I had to chase them away when I needed quietness.

The completion of the task has shown strongly the truth of the following:

> And further, by these, my son, be admonished: of making many books *there is* no end; and much study is a weariness of the flesh.

Ecclesiastes 12:12

Chapter 1

Introduction

1.1 Problem Statement

In the Multicommodity Network Flow problem (MCNF), several commodities must be shipped between various nodes of a network. Units of a commodity are processed along each arc at a variable cost, and processing capacities may constrain the joint flow of commodities. The goal is to satisfy the shipping requirements at a minimum cost.

Commodities may share various characteristics that influence the process in varying ways, such as: mass, density, length, width, height, area, volume, viscosity, critical mass[1]. Processing limits for each characteristic result in a set of joint constraints.

Multicommodity Networks have been used to model a variety of applications. MCNF was first formulated to describe the routing of messages in communications networks [22,34], and subsequently by Hu [32] and White [68].

Scheduling problems involving transportation systems have been another area for applications of MCNF. Early work included planning of urban traffic systems [15,55]. Railway car [11,61] and airline scheduling [65] are ongoing areas for research. Recently, several authors have used USAF Military Airlift Command problems for a Patient Distribution System that determines how well patients can be evacuated from Europe [38,60,62].

Inventory and transportation models introduce natural network structures combined with production constraints, and generate MCNF problems. Applications have included warehouse location [24], manufacturing planning [30], and operations scheduling for oil refineries [43].

In the realm of social planning, MCNF has been used to assigning students to schools to achieve desired

[1] For nuclear fissionables like Plutonium and Uranium 235

ethnic compositions [19].

1.2 Multicommodity Network Flow Formulation

To define MCNF formally, consider a network processing K commodities where:

- $\mathcal{A}^k$ defines the set of arcs for commodity k, for $k = 1, \ldots, K$,
- $x_{i,j}^k$ is the flow of commodity k from node i to node j,
- $c_{i,j}^k$ is the cost per unit of flow of commodity k from node i to node j,
- g_j^k is the requirement of commodity k at node j,
- $\bar{h}_{i,j}^k$ is the flow usage of mutual resources by commodity k from node i to j,
- $h_{i,j}$ is the mutual resource capacity from node i to j,
- $u_{i,j}^k$ is the bound on the flow usage of commodity k from node i to j.

The MCNF problem may be defined as:

Linear Program Formulation 1

Minimize

$$\sum_{k,i,j} c_{i,j}^k x_{i,j}^k$$

subject to

$$\sum_i x_{i,j}^k - \sum_o x_{j,o}^k = g_j^k \text{ for all } (i,j) \in \mathcal{A}^k,\ (j,o) \in \mathcal{A}^k,\ k = 1, \ldots K \tag{1.1}$$

$$\sum_k \bar{h}_{i,j}^k x_{i,j}^k \ \leq h_{i,j} \ \text{ for all } (i,j) \in \bigcup_{k=1}^{K} \mathcal{A}^k, \tag{1.2}$$

$$u_{i,j}^k \ \geq x_{i,j}^k \ \geq 0 \text{ for all } (i,j,k) \in \mathcal{A}^k,\ k = 1, \ldots K. \tag{1.3}$$

Equation 1.1 describes flow conservation constraints. Flows of each commodity into a node must equal the flows out of the node, except at sources and sinks, where there will either be net inflows or net outflows.

Equation 1.2 describes a general set of joint capacities.

It is assumed for ease of notation that $\mathcal{A}^k = \mathcal{A}^{k'}$ for any two commodities $k \neq k'$, although computational speed may be gained by recognizing that they can vary widely.

For each commodity, Equation 1.3 limits the amount of flow that can traverse each individual arc. To simplify the presentation, the upper bound u is omitted from most formulations presented in the thesis. When necessary, the individual capacity bounds are represented explicitly. The software developed can accomodate such individual bounds.

The equations 1.1 and 1.2 defining the MCNF problem can be represented in block format as:

Minimize

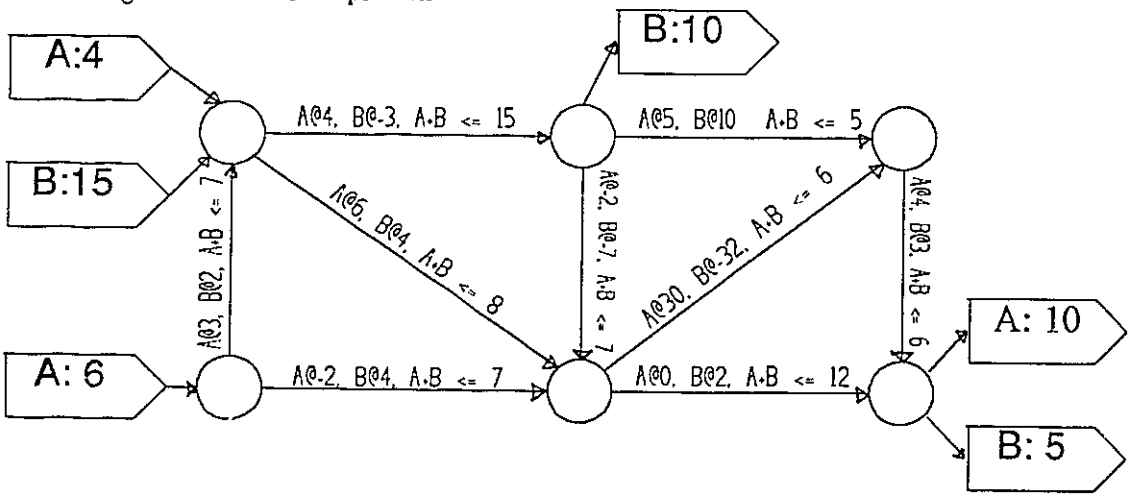
$$c_1x_1 + c_2x_2 + c_3x_3 + \ldots + c_kx_K$$

subject to

$$\begin{array}{rcl}
G_1x_1 & = & g_1, \\
G_2x_2 & = & g_2, \\
G_3x_3 & = & g_3, \\
\ddots & = & \vdots \\
G_Kx_K & = & g_K, \\
H_1x_1 + H_2x_2 + H_3x_3 + \ldots + H_Kx_K & \leq & h, \\
x_1,\ x_2,\ x_3,\ \ldots\ x_K & \geq & 0,
\end{array}$$

where $G_1, G_2, \ldots, G_K$ represent network node incidence matrices, $x_1, x_2, \ldots, x_K$ represent the flows of each commodity, and $H_1, H_2, \ldots, H_K$ describe the joint constraints relating to each commodity.

Here is a diagram of an MCNF problem:



This picture displays costs and constraints on flows of commodities A and B as follows; the arc marked with `A@4, B@3, A+B <= 15*` denotes that:

- Each unit flow of A has a cost of 4,
- Each unit flow of B has a cost of 3,

- The joint capacity constraint is $A + B \leq 15$, i.e. commodities A and B must share a total capacity of 15 units for the arc.

1.3 Motivation: MCNF versus Pure Network Flow Problems

Whereas special purpose algorithms can solve pure network flow problems hundreds of times faster than general linear programming methods, improvements found for MCNF are less spectacular — on the order of 3 times that of the general decomposition method for linear programming [3]. Therefore, there is opportunity for performance improvements by investigating new methods for large linear programs. Interior Point algorithms solve linear programming problems in polynomial time, and in practice accelerate the solution of very large linear programs. Unlike the simplex method, which travels from one extreme vertex to the next, Interior Point algorithms travel through the interior of the feasible region of possible solutions.

Another approach is to use the special structure of MCNF. This approach, long exploited with the simplex method, has been re-explored in the context of Interior Point methods and parallel algorithms. This thesis explores the potential of Interior Point methods to solve MCNF more efficiently. Beyond MCNF, this thesis addresses a relatively unexploited method: combining Interior Point methods and decomposition methods.

1.4 Achievements in the Thesis

This thesis presents an original synthesis of the decomposition and partitioning approaches for MCNF, by examining them in a nonlinear programming framework. Original relationships are derived between methods, as delineated below:

- New non-simplicial methods of solving MCNF were thoroughly compared,
- Various Interior Point methods described in new implementations [27, 49, 50, 60] are compared in Section 5.3.
- In Section 3.2, the work of Kapoor and Vaidya [35] is corrected (Equation 3.13) and is related to the classical partitioning and decomposition methods for MCNF.
- The classical gradient and Newton projections are related to simplex partitioning in Section 4.1.
- A shifted barrier method and augmented Lagrangian methods are shown to yield similar relaxations in Section 5.4, as envisioned in the conclusion of C. J. Staniec's dissertation [62].

- Two methods of solving MCNF by Interior point methods: shifted barrier and central pricing methods of Goffin et al. [27] are shown to follow the same trajectory in Section 5.3.3.

- Rosen's partioning method is shown in Section 4.3 to follow the steps of the dual partition method of Grigoriadis and White [29].

- Nonsimplicial partitioning was chosen for implementation.

  The implementation of a partitioning method which combines interior and exterior trajectories, an original approach not found in the literature, is described in Section 6.1.4. Interior Point methods are mostly used for very large scale problems, using sophisticated data structures and numerical methods, which are often proprietary. The affine scaling algorithm that was used did not have very good accuracy, and was fairly slow. Therefore it was neither possible nor beneficial to test it on extensive data. However, initial estimates of efficiency are performed.

  Based on experimentation there was no evidence of superiority of partitioning using affine scaling over using affine scaling to solve the larger linear program formulation.

1.5 Plan of Thesis

Chapter 1 defines and positions the problem. The classical literature on MCNF is reviewed in Chapter 2 with a brief introduction to Interior Point methods and their applications to MCNF. Chapter 3 lays the foundations for the application of Interior Point Methods to MCNF and corrects a partitioning method proposed by Kapoor and Vaidya [35]. Partitioning techniques are analyzed in depth in Chapter 4, as a prerequisite to an algorithm that was implemented. An alternative strategy has been widely used: Decomposition methods, surveyed in Chapter 5, span such well-known methods as simplicial, Lagrangian, Dantzig-Wolfe, and Augmented Lagrangian decompositions. Recent implementations of these decompositions using Interior Point methods suggest techniques to improve the efficiency of the partitioning method. Chapter 6 uses the methods devised in the previous two chapters to design an affine scaling and partitioning algorithm. Computational experiments compare algorithmic strategies and assess their performance, based on the size of MCNF problems. Chapter 7 proposes directions for further development.

Chapter 2

Literature Review

2.1 Classical Solution Methods for MCNF

This section contains a chronological review of the solution methods for MCNF. After a review of classical methods, Interior Point methods are introduced for a better understanding of more novel approaches.

Early Linear Program Formulation of MCNF

Kalaba & Juncosa [34] describe one of the earliest multicommodity network formulations used to solve routing problems for telecommunication networks, involving stations, links and users. They describe three sets of linear constraints, relating to:

1. Supply/Demand at each station
2. Capacities for each arc
3. Capacities for each switching station

They propose maximizing a functional that involves a combination of cost and performance, and point out that a similar formulation could be used to model flows of traffic in either urban or national transportation systems. They informally envisage how such a formulation could assist in the design of telecommunications systems by attaching costs to the presence/absence of stations and arcs.

Shortest Chain Algorithm for MCNF

One of the earliest analyses of multicommodity networks was done by Ford and Fulkerson [22]. They describe the difficulty of solving MCNF: the mutual capacity constraints prevent the use of algorithms

dedicated to single commodity network optimization, and the large size of MCNF problems makes direct use of the simplex method impractical. To obviate the large requirements of the simplex method, they propose an algorithm that searches for shortest chains within the networks. They note that even this method would not be sufficient to solve large problems.

Partitioning

Partitioning methods for MCNF attempt to capitalize on the success of the simplex methods for network optimization. Many steps of the simplex algorithm to solve MCNF can be performed efficiently within the network substructure that MCNF problems possess. In particular, a network basis is completed at each iteration to form part of the basis of the MCNF problem. The remainder of the network structure is then manipulated by nonspecialized simplex steps.

Grigoriadis and White [29] describe a dual partitioning method to solve the MCNF problem in which they first compute optimal solutions to the network subproblems, and then determine pivot adjustments based on a set of *current* constraints, representing those mutual constraints that are "most binding," i.e., constraints whose slack variable is nonbasic, or "nearly nonbasic."

Decomposition Methods

Unlike partitioning methods, decomposition methods do not complete bases, but solve separate subproblems, the solutions of which are coordinated by a general linear program.

Tomlin [63] applied the principle of Dantzig-Wolfe decomposition to MCNF and presents its equivalence to the shortest chain construction of Ford and Fulkerson.

Ali and Kennington [4, 39] describe primal decomposition of multicommodity flow problems using a price directive decomposition.

Kennington and Shalaby [40] implement a resource directive decomposition to reallocate scarce resources using a subgradient optimization approach.

Ali et al. [5] compare the performance of these methods to that of a partitioning method. They found that the resource directive decomposition method determined solutions approximately twice as quickly as either partitioning or price directive decomposition. Unfortunately, the resource directive decomposition did not guarantee convergence to an optimum, whereas the other methods did guarantee convergence.

Penalty Methods

Classical decomposition algorithms are reputed to have slow convergence rates near optimality. Nonlinear penalty methods have been applied to overcome this difficulty [62]. The Reduced Simplicial

Decomposition has been successfully applied to this nonlinear optimization.

Combinatorial Aspects of MCNF

A review of the early MCNF literature would not be complete without reference to the first textbook devoting considerable space to MCNF. Hu [32] describes several variations of the Multicommodity Network Flow problem (MCNF). He shows how MCNF is distinct from single commodity networks. In such networks, arc flows in opposing directions cancel one another. In MCNF, the flows of differing commodities do not cancel one another. The book emphasizes the structural properties of the problem rather than computational methods. It is a useful reference as it presents relationships between MCNF and pure network flow problems; among the following topics, the latter two still form challenging areas of research.

- **Feasibility** — Is there a solution that allows all flows to get from the sources to the sinks?
- **Maximum flow** — What is the maximum possible flow through the network? This requires a value function so that flows of differing commodities may be appropriately traded off against each other. The determination of such a value function may be a problem.
- **Integer solutions** — Can an integer solution be found? Integrality can often be easily guaranteed with simpler networks, but it is more difficult to force integrality for MCNF problems. There may be a unique non-integral solution.
- **Synthesis of a Communication Network** — One interpretation of a multicommodity network is as the model of a communications network in which nodes represent stations, and each commodity represents a message that must be communicated from one station to another. The major issue in this case is whether or not the required information flows are feasible.

2.2 Development of Interior Point Methods

The focus of the literature review of this thesis temporarily shifts to a new method of solving linear programs, the Interior Point method, because many subsequent designs of MCNF algorithms are based on it.

Interior Point methods may be distinguished from the simplex method as follows:

The sequence of solutions provided by the simplex method travels along the edges of the feasible region, from vertex to vertex. Interior Point methods produce a sequence of points, each of which is interior to the feasible region.

Motivation for Interior Point Methods

The computation time of simplex algorithms to solve linear programs can increase exponentially with problem sizes. There have been many attempts to produce improved Linear Programming algorithms that can guarantee worst-case polynomial time performance. The first Linear Programming method that was shown to run in polynomial time was the Ellipsoid method of Khachian [41]. It established the existence of polynomial algorithms for linear programming, a long standing theoretical question, but has not yielded any practical algorithms because the matrices that it generates get increasingly dense. Karmarkar [36] presents a polynomial-time algorithm for linear programming, that has been more practical to use. Thus, Interior Point methods have revived interest in many applications of linear programming. In the past few years, many new algorithms to solve pure network flow problems or MCNF either refer to Interior Point methods or directly apply them. A presentation of the basic elements of the methods is necessary for a full understanding of the progress of the research described in this chapter.

Projective Method of Linear Programming

Karmarkar's method [36] consists of applying repeatedly a projective transformation to the feasible set of the problem:

$$\{\min_x cx : \; Ax = 0, \; e^T x = 1, \; x \geq 0\}$$

In the transformed space, the non-negativity constraints are replaced by a sphere, and a step improves the objective at a rate sufficient to guarantee convergence of a polynomial number of such transformations to an optimum solution of the problem. These steps are next reviewed in detail.

Karmarkar's algorithm creates a sequence of points $x^0, x^1, \ldots, x^k$ satisfying the constraints of the linear program. It computes x^{k+1} using x^k as follows:

Let $B = \begin{bmatrix} AD_{x^k} \\ \hline e^T \end{bmatrix}$, where $D_{x^k} = \begin{bmatrix} x_1^k & 0 & \ldots & 0 \\ 0 & x_2^k & \ldots & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \ldots & x_n^k \end{bmatrix}$. This notation for the diagonal matrices has been adopted throughout the thesis. e is used to refer to the column vector $(1, 1, \ldots 1, 1)^T$ throughout the thesis, and is assumed to have conformable dimensions with the vector/matrix equations in which it is used.

Compute the direction of improvement

$$c_p = \left[I - B^T(BB^T)^{-1}B\right] D_{x^k} c^T.$$

Apply a projective transformation yielding the new point x'

$$x' = \frac{D_{x^k}^{-1} x}{e^T D_{x^k}^{-1} x},$$

then the iterate

$$x'' = x' - \alpha r \frac{c_p}{|c_p|}.$$

where: r is the radius of the inscribed sphere. Usually $r = \frac{1}{\sqrt{n(n-1)}}$. $\alpha \in (0,1)$ is the proportion of the distance towards the edge that is traversed. Karmarkar's proof of polynomial convergence assumes $\alpha = 1/4$.

Apply the inverse projective transformation to x''

$$x^{k+1} = \frac{D_{x^k} x''}{e^T D_{x^k} x''}.$$

Several commercial LP solvers are based on Karmarkar's algorithm or its variants. Most of the ongoing work on Interior Point methods is based on a simplification of Karmarkar's original method that avoids projective transformations. These affine scaling methods do not guarantee polynomial convergence, but perform quite well in practice.

Affine Scaling Method

Dikin [21], and later Barnes [6] and Vanderbei et al. [64] describe affine scaling versions of Karmarkar's algorithm. A projected gradient search is used to find the optimum:

- Compute the direction of improvement, c_p:

  $$c_p = \left[I - B^T (BB^T)^{-1} B\right] D_{x^k} c^T,$$

  where $B = AD_{x^k}$.

- Determine the maximum step size that maintains feasibility:

  $$\rho = \frac{1}{\max_i (P_{x^k} c)_i}$$

  where $P_x = I - D_x A^T (AD_x^2 A^T)^{-1} AD_x$ is the projection of x onto the null space of AD_{x^k}.

- Compute the new point x^{k+1}

  $$x^{k+1} = x^k - \alpha \rho c_p$$

  $\alpha \in (0,1)$ ensures that all coordinates of x^{k+1} remain greater than zero.

The implementation used in this thesis is based on that of Vanderbei et al [64]. Affine scaling methods were later applied to primal-dual formulations of linear programs with great success [42, 46, 51, 52], as described in Appendix A.

2.3 Advanced Methods for Large Scale MCNF

The successful application of Interior Point methods to very large problems (i.e. airlift of commodities [60]), has renewed interest in MCNF. For example, Lustig and Li [45] use a predictor-corrector interior point method to solve a combination of the primal and dual formulations. The ensuing research is merely surveyed next with no further explanation because it is analyzed in later chapters.

Partitioning for Interior Methods

Kapoor and Vaidya [35] present ways to specialize the Interior Point method for MCNF, accelerating its convergence. They restrict their attention to the feasibility problem, i.e., finding a flow satisfying the constraints of MCNF. Their method, which is discussed in more detail in Chapter 3.2, has not yet been implemented.

Decomposition for MCNF

Meyer and Schultz [60] apply a method of optimizing block angular problems, using variant Interior Point methods, to find approximate solutions of multicommodity network flow problems.
Jones et al.[33] apply Interior Point methods to a node-chain formulation of the multicommodity network flow problem.

Parallelization

One of the additional benefits of decomposition methods is that they allow problems to be parallelized [60,73,75]. The smaller single-commodity networks can be solved much more quickly than the larger networks, so that the solutions are found faster even if they are solved serially. They may be solved in parallel on separate processors, further decreasing the time required.
Interior Point methods used to solve master problems that arise from decomposition methods for MCNF may be performed partly in parallel for each commodity. This reduces memory consumption so that larger problems can be solved on smaller computers.

Chapter 3

Large Scale LP Implementation of MCNF

In this chapter, MCNF is treated as a large linear program. Interior Point methods are applied to it, and its size is reduced by adapting the reduction techniques of Kapoor and Vaidya. This is a prelude to the partitioning techniques analyzed in Section 4.3 and implemented in Section 6.1.4.

Early analysis of MCNF [22] involved formulating the problem as a linear program, and then attempting to solve it using a general simplex method. Unfortunately, these linear programs grow to enormous size, even for apparently small problems. For instance, a problem involving 20 nodes, 50 arcs, and 20 commodities would require 1000 variables and 400 network constraints, ignoring the joint constraints. This makes the use of Interior Point methods, which are more efficient than the simplex method for very large problems, appealing.

3.1 Interior Point Method for Linear Programming

Interior Point methods are first analyzed on general linear programs:

Linear Program Formulation 2

$$\{\min cx : \ Ax = b, \ x \geq 0\},$$

and then specialized to MCNF. Whereas the main steps of the algorithm have been described in Section 2.2, the following sections focus on initialization and reduction schemes.

3.1.1 Canonical Form

Traditionally, Interior Point methods have considered an alternative formulation, the canonical form of Karmarkar [36]

Linear Program Formulation 3

$$\{\min_x cx : \ Ax = 0, \ e^T x = 1, \ x \geq 0\},$$

in which

- $x^0 = \frac{e}{n}$ is a feasible interior point in this formulation,
- the optimum objective value is $cx^* = 0$.

3.1.2 Initialization

Several transformations can be used to transform an LP from Formulation 2 to Formulation 3. The best known is the original transformation of Karmarkar [36]. An alternative affine transformation is reviewed in Appendix B. Consider a given instance of Linear Program Formulation 2:

Linear Program Formulation 4

Minimize

$$c'x$$

subject to

$$A'x = b',$$

$$x \geq 0.$$

With a classical scheme using one artificial variable, Karmarkar produces an interior point [17]: choose an arbitrary initial point, $x^\bullet > 0$ (which may not satisfy $Ax^\bullet = b$), and let $a = b' - A'x^\bullet$; define the new system:

Linear Program Formulation 5

Minimize

$$c'x + M\dot{x}$$

subject to

$$A'x + a\dot{x} = b',$$

$$x \geq 0, \dot{x} \geq 0.$$

Formulation 5 may be relabelled $\{\min cx : Ax = b,\ x \geq 0\}$, for which an interior point $(x^\bullet, 1)$ has been defined. Alternatively, in keeping with the simplex method, $\dot{x}$ may be replaced by a vector of artificial variables, which avoids the dense column a. Initialization schemes specialized to MCNF are introduced in Section 3.2, and compared in Section 6.1.3.

3.2 Interior Point Method for MCNF

In this subsection, Interior Point methods are tailored to MCNF. A reduction method decreases the number of calculations required by the algorithm. Consider now MCNF in restricted block form:

Linear Program Formulation 6

Minimize

$$cx$$

subject to

$$Gx = g,$$

$$Hx + Iy = h,$$

$$x, y \geq 0,$$

where I is an identity matrix of size n, and n is the number of arcs. Assume here that $Gx = g$ contains artificial variables (which may be cast as part of the network as flows on artificial arcs); their large objective coefficients will drive them to zero, eliminating infeasibility. Let $w = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$; Karmarkar's projective transformation [36, pages 386-387] $T_{w_0}(w) = \frac{D_{w_0}^{-1} w}{e^T D_{w_0}^{-1} w + 1}$ transforms the initial starting point w_0 to the centre of the simplex, yielding:

Linear Program Formulation 7

Minimize

$$cD_w w'$$

subject to

$$GD_x x' - gz' = 0, \tag{3.1}$$

$$HD_x x' + D_y y' - hz' = 0, \tag{3.2}$$

$$e^T w' = 1, \tag{3.3}$$

$$w' \geq 0, \tag{3.4}$$

where $w' = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$.

The projective method seeks a feasible direction w^d that reduces the cost of the objective function by solving an optimization problem in a sphere that replaces Equation 3.4. For ease of notation, the transformed space is translated so that its origin is the centre of a sphere. Let $w'' = \begin{pmatrix} x'' \\ y'' \\ z'' \end{pmatrix} = w' - \frac{e}{n+2}$. Since $\frac{e}{n+2}$ satisfies Equations (3.1)-(3.3), the translation produces a zero right-hand side, and the direction of descent is the solution of:

Linear Program Formulation 8

Minimize

$$cD_w w''$$

subject to

$$GD_x x'' - gz'' = 0, \tag{3.5}$$

$$HD_x x'' + D_y y'' - hz'' = 0, \tag{3.6}$$

$$e^T w'' = 0, \tag{3.7}$$

$$w''^T w'' \leq 1. \tag{3.8}$$

3.2.1 Reduction Technique

To diminish the size of the problem, the capacity constraints (3.6) can be eliminated from Linear Program Formulation 8. Reexpressed as $y'' = D_y^{-1} h z'' - D_y^{-1} H D_x x''$, Equations 3.6 can be used to substitute for the slack variables y''. The matrix, $R = \begin{bmatrix} I & 0 \\ -D_y^{-1} H D_x & D_y^{-1} h \\ 0 & 1 \end{bmatrix}$ defines the reduction $T : \mathbf{x} = \begin{pmatrix} x'' \\ z'' \end{pmatrix} \rightarrow w'' = R\mathbf{x}$.

R transforms $w''^T w'' \leq 1$ to Equation 3.12, where:

$$Q = \begin{bmatrix} I + D_x H^T D_y^{-2} H D_x & -D_x H^T D_y^{-2} h \\ -h^T D_y^{-2} H D_x & 1 + h^T D_y^{-2} h \end{bmatrix} = R^T R. \tag{3.9}$$

With y'' eliminated, the system can be described as:

Linear Program Formulation 9

Minimize

$$\gamma \mathbf{x} \tag{3.10}$$

subject to

$$\bar{A}\mathbf{x} = 0, \tag{3.11}$$

$$\mathbf{x}^T Q \mathbf{x} \leq 1, \tag{3.12}$$

where

$$\bar{A} = \begin{bmatrix} GD_x & 0 \\ e_x^T - e_y^T D_y^{-1} H D_x & 1 + e_y^T D_y^{-1} h \end{bmatrix},$$

and

$$\gamma = cDR.$$

For this reduced problem, Kapoor & Vaidya's calculation of the optimal direction, c_p, is incorrect in [35]. Using the Lagrangian multiplier described by Karmarkar [36], an optimal solution – representing a step direction – is given as:

$$c_p = \left[Q^{-1} - Q^{-1}\bar{A}^T\left(\bar{A}Q^{-1}B^T\right)^{-1}\bar{A}Q^{-1}\right]\gamma^T \tag{3.13}$$

In summary, by eliminating the slack variables of the mutual capacity constraints, a more compact calculation of the projective step may be obtained. The implementation described in Section 6.1.4 is based on a similar reduction using the flow conservation constraints $Gx = g$.

3.2.2 Kapoor and Vaidya's formulation as a Phase I method

Kapoor and Vaidya's original algorithm [35] actually uses a somewhat different formulation. They seek only a feasible solution of MCNF by stipulating a cost function featuring only artificial variables, which vanish at the optimum. For each commodity, their network has one supply node and one demand node. This special network formulation can replace the network flow formulation (1.1)–(1.3) by adding for each commodity:

- a source node and a sink node, commonly called "super source" and "super sink," and
- arcs directed from demand nodes in the original network to the sink node, and arcs directed from the source node to the supply nodes of the original network, which replace supplies and demands by capacities on the arcs.

Analyzing a network with unique supply and demand nodes for each commodity, Kapoor and Vaidya add backflow arcs directed from the sink to the source, and transform the formulation into a circulation problem. The total demand for the commodity is treated as a capacity constraint, the slack of which is an artificial variable. Using this approach, (1.1)–(1.3) can be transformed to:

Linear Program Formulation 10

Maximize

$$e^T x_b$$

subject to

$$\left.\begin{array}{l} \begin{bmatrix} & -I_{s_k} & 0 & 0 \\ G & 0 & 0 & 0 \\ & 0 & I_{d_k} & 0 \\ 0 & e_{s_k}^T & 0 & -1 \\ 0 & 0 & -e_{d_k}^T & 1 \end{bmatrix} \begin{pmatrix} x_k \\ x_{s_k} \\ x_{d_k} \\ x_{b_k} \end{pmatrix} = 0, \\ I_{s_k} x_{s_k} \le h_{s_k}, \\ I_{d_k} x_{d_k} \le h_{d_k}, \end{array}\right\} \quad \text{for } k = 1, \ldots, K,$$

$$Hx \le h,$$

$$x \ge 0,$$

where s indexes the supply nodes, d indexes the demand nodes, and b indexes the backflows. The formulation of a circulation problem is closer to Karmarkar's canonical formulation because the right hand side g is 0. However, K nodes and arcs are added, with corresponding capacity constraints, increasing the size of the problem solved. Because each of these nodes is only connected to the original nodes bearing supply or demand for the particular commodity, the increase is moderate. In Linear Program Formulation 10, the arcs $\mathcal{A}^k$ are represented by $\begin{bmatrix} & -I_{s_k} & 0 \\ G & 0 & 0 \\ & 0 & I_{d_k} \end{bmatrix}$, therefore the assumption that $\mathcal{A}^k = \mathcal{A}^{k'}$ must be waived for the new arcs s_k and d_k. It can also be waived for the rest of the network, which may allow the elimination of arcs and nodes that are not used for a given commodity k. To distinguish the n_2 additional arcs indexed by s, d or b, Kapoor and Vaidya reorder the columns. The resulting identity incidence matrix I_{n_2} contains in I_K all of the backflows created: $I_{n_2} = \begin{bmatrix} I_K & 0 \\ 0 & I_{n_2-K} \end{bmatrix}$. Then the expanded mutual capacity constraint matrix is

$$H' = \underbrace{\begin{bmatrix} I_{n_2} & 0 \\ 0 & H \end{bmatrix}}_{Kn+n_2} \left.\begin{array}{l} \\ \\ \end{array}\right\} n + n_2,$$

in which the original mutual capacity coefficient matrix is

$$H = \underbrace{\left[\begin{array}{cccc|cccc|c|cccc} 1 & 1 & \ldots & 1 & 0 & 0 & \ldots & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 0 & 1 & 1 & \ldots & 1 & \ldots & 0 & 0 & \ldots & 0 \\ & & & & & & \ddots & & & & & & \\ 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & \ldots & 1 & 1 & \ldots & 1 \end{array}\right]}_{K \times n} \Bigg\} n.$$

A simple labeling method introduces an initial point corresponding to a positive circulation, scaled to satisfy the joint constraints. In case a super source and a super sink had to be added, the formulation of Kapoor and Vaidya can be reduced by omitting the backflow and the capacity constraint of each commodity.

Maximize

$$\sum_{k=1}^{K} e_{s_k}^T x_{s_k}$$

subject to

$$\left.\begin{array}{l} \left[\begin{array}{ccc} & -I_{s_k} & 0 \\ G & 0 & 0 \\ & 0 & I_{d_k} \\ 0 & e_{s_k}^T & -e_{d_k}^T \end{array}\right] \left(\begin{array}{c} x_k \\ x_{s_k} \\ x_{d_k} \end{array}\right) = 0, \\ I_{s_k} x_{s_k} \leq h_{s_k}, \\ I_{d_k} x_{d_k} \leq h_{d_k}, \end{array}\right\} \text{ for } k = 1, \ldots, K,$$

$$Hx \leq h,$$

$$x \geq 0.$$

Chapter 4

Partitioning

This chapter presents the partitioning method used in the implementation and its relationship with Interior Point methods.

4.1 Introduction to the Partitioning Method for MCNF

Kapoor and Vaidya's method has been presented as a specialization of a general projective algorithm, where the numerical steps to calculate the iterate can be streamlined. Viewing the method as an application of a logarithmic barrier optimization provides additional insight. Given the following LP formulation:

Minimize

$$cx$$

subject to

$$Ax = b,$$

$$x \geq 0,$$

an optimal solution may be found as the limit solution of a nonlinear program in which a barrier function forces the variables to remain positive:

$$\lim_{\mu \to 0} \min cx - \mu \sum_i \ln x_i$$

subject to

$$Ax = b.$$

Gill, Murray, Saunders, Tomlin and Wright [25] show that Karmarkar's step can be replicated by a projected Newton step on the barrier function for an appropriate choice of the parameters μ. This alternative view and the associated formulation eases the understanding of Kapoor and Vaidya's approach: it preserves the original variables and constraints, and the nonlinear program can be solved using classical partitioning or decomposition methods for MCNF, such as are presented in Chapters 5 and 6.

Application of Logarithmic Barrier Functions to MCNF

The logarithmic barrier technique may also be applied when the constraint matrix consists of several blocks, as is the case for MCNF problems. Consider the MCNF problem formulation:

Linear Program Formulation 11

Minimize

$$cx$$

subject to

$$Gx = g,$$

$$Hx + y = h,$$

$$x, y \geq 0.$$

A barrier function replaces the non-negativity constraints by modifying the objective function:

Minimize

$$cx - \mu \left[\sum_i \ln x_i + \sum_j \ln y_j \right]$$

subject to

$$Gx = g,$$

$$Hx + y = h.$$

$y \geq 0$ does not need to be enforced explicitly, therefore it may be permanently replaced by $(h - Hx)$, leaving:

Minimize

$$cx - \mu \left[\sum_i \ln x_i + \sum_i \ln (h_i - H_i x) \right]$$

subject to

$$Gx = g.$$

This problem can be solved as a nonlinear program with network flow conservation constraints. In the absence of joint capacities, this method has recently been implemented [56, 57] as an Interior Point algorithm to solve pure network flow problems. Solving MCNF using an Interior Point method could be an extension of this algorithm to a more general logarithmic barrier function.

Interior Point methods have been successful for very large pure network flow problems. If the size is not extremely large, a more classical solution approach, which prefigures the partitioning algorithm proposed in this thesis, is to maintain the nonnegativity constraint $x \geq 0$, to relax the problem as a nonlinear network flow problem and to solve it by variants of the very efficient simplex algorithm for pure networks.

Minimize

$$cx - \mu \sum_i \ln (h_i - H_i x)$$

subject to

$$Gx = g,$$

$$x \geq 0.$$

The nonlinear component of the objective function $F_\mu(x) = cx - \mu \sum_i \ln (h_i - H_i x)$ is represented in Figure 4-1. Solution of pure network flow problems with linear or nonlinear objective functions is described in the next section.

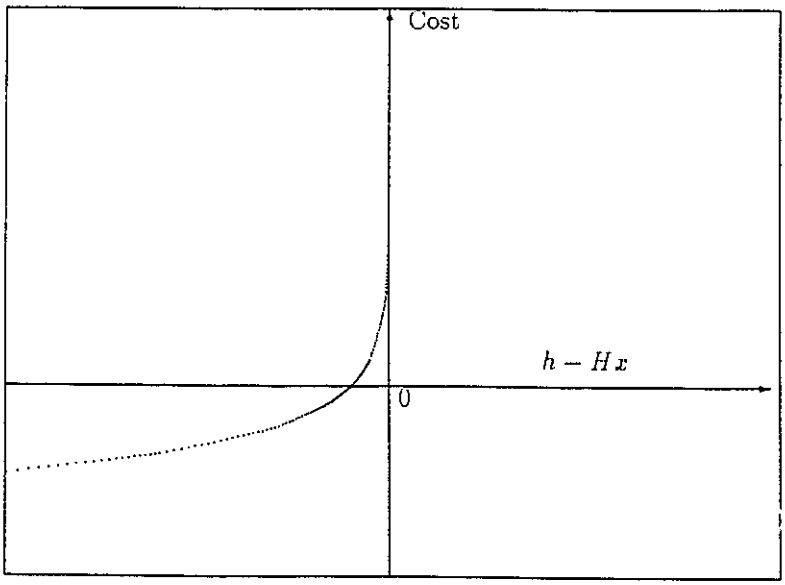


Figure 4-1: Logarithmic barrier function

4.2 Solution of Pure Network Flow Problems

The review focuses on pure Network Flow problems with linear objective functions, and their use to solve the subproblems of the nonlinear problem.

4.2.1 Methods for Networks with Linear Objective Functions

Simplex-Based Methods

The classical methods of solving network flow problems involve specializations or variants of the simplex method. These combinatorial methods are usually quite efficient because they can search through alternative solutions very quickly. On the other hand, they do not guarantee any convergence rate. The simplex method usually performs well in spite of the theoretical disadvantages.
Recent research has focused on the design of polynomial and strongly polynomial algorithms, such as the dual simplex algorithm due to Orlin [54] that performs $O(n^3 \log n)$ pivots.
Helgason & Kennington [38] present an extensive treatment of classical algorithms, and Ahuja et al. [3] reviews recent advances in minimum cost network flow algorithms.

Relaxation Algorithms

Among many relaxation algorithms, some use classical methods such as the simplex framework, or subgradient optimization, for a survey, see Bertsekas [12]. Some of the most successful are by Bertsekas et al. [13], part of a family of relaxation algorithms that either:

- augment flow from an excess node to a deficit node along a path whose arcs all have reduced costs of zero, or
- adjust the potentials of some subset of the nodes.

Using Interior Methods

Resende and Veiga [56, 57] propose the use of a dual affine scaling algorithm for linear programming to solve minimum cost network flow problems. Interior methods tend to be less affected by degeneracy than the simplex method, which can easily stall under such conditions. Primal network simplex algorithms have the advantage of being implemented using integer arithmetic, as compared to the floating point required for interior point methods. However, the number of iterations required for interior methods tend to grow slowly with problem size, in contrast to the simplex method. Thus, one would expect Interior Point methods to become competitive for very large problem sizes.

Resende and Veiga's [56,57] initial results indicate that this interior point method is competitive, and their DLNET software found solutions more quickly than the network simplex code NETFLO [38] for the larger problems tested. The point at which they found that the methods perform equally well is when the number of arcs is on the order of 50,000.

An additional advantage of using interior point algorithms as compared to the simplex method is that it is often possible to parallelize the interior algorithm, which is quite difficult with the simplex method.

Double Scaling

The algorithms with the best polynomial orders have required sophisticated data structures, which has caused high computational overheads, making them impractical for use [3, page 176].

The most recent work has been on a "double scaling" algorithm due to Ahuja et al. [2]. It runs in order $O(nm \ln \ln U \ln nC)$ time, where n is the number of nodes, m is the number of arcs, U is an upper bound for the arc bounds, and C is the maximum cost over the arcs.

Ahuja et al. [3] report that "it appears that the relaxation algorithm of Bertsekas and Tseng [13] and the primal simplex algorithm due to Grigoriadis [28] are the two fastest algorithms for solving the minimum cost flow problem in practice."

4.2.2 Methods for Networks with Nonlinear Objective Functions

Many applications of network problems have nonlinear costs. Others arise from various barrier or penalty schemes to enforce non-negativity or side constraints such as MCNF joint capacity constraints. Consider a network problem with the nonlinear objective function $F(x)$:

Minimize

$$F(x)$$

subject to

$$Gx = g,$$

$$x \geq 0,$$

Nonlinear partitioning methods rely on the the block structure $G = \left[\begin{array}{c|c} G_B & G_N \end{array} \right]$, where B indexes a regular submatrix of G. Thus, $G_B x_B + G_N x_N = g$, or $x_B = G_B^{-1}(g - G_N x_N)$. Denote $F(x_B, x_N)$ as $F'(x_N)$. A reduced problem features only the variables x_N:

Minimize

$$F'(x_N)$$

subject to

$$G_B^{-1}(g - G_N x_N) \geq 0,$$

$$x_N \geq 0,$$

which is related to the logarithmic barrier formulation of the reduced problem found in Section 4.3. Various methods reduce the cost while preserving feasibility of the network subproblems, such as Wolfe's reduced gradient $\nabla F'(x_N) - \left(G_B^{-1} G_N\right)^T \nabla F_{x_B}(x_B, x_N)$ [69, 70], Zangwill's convex simplex method [71, 72], or Bertsekas and Gafni's Newton step [14].

For example, Beck, Lasdon and Engquist [10] describe a computer implementation of an algorithm that solves problems with a nonlinear objective function and linear network constraints, based on the method of Murtaugh and Saunders [53]. The algorithm partitions the decision variables into three sets: basic x_B, superbasic x_S and fixed nonbasic x_N. The basic and non-basic variables are computed as they would be in a simplex-based algorithm, using the linear constraints. The "superbasic" variables are positive nonbasic variables called forth by nonlinearities.

Beck, Lasdon and Engquist found that this method was less efficient than the network simplex method for a linear objective function. They compared the performance of a two phase version in which the initial linear phase is solved using the simplex method to that of a single phase "Big M" version, in which the problem is nonlinear from the beginning. The two phase version dominated the "Big M" method both in terms of the number of iterations required and in CPU time.

Many other nonlinear methods have been applied to network flow problems, including Frank & Wolfe's decomposition [23] and Reduced Simplicial Decomposition [31], both of which are reviewed in Section 5.3. For a survey, see Dembo et al. [74].

4.3 Partitioning

Even before nonlinear methods were introduced, early Linear Programming algorithms exploited the partial network structure of MCNF [22]. In particular, partitioning uses the block structure of the problem. Consider again the restricted block formulation of MCNF:

Minimize

$$cx$$

subject to

$$Ax = b,$$

$$x \geq 0,$$

where A has the block structure:

$$A = \left[\begin{array}{c|c} G & 0 \\ \hline H & I \end{array}\right]$$

and G is the node-arc incidence matrix describing a network problem. In Section 3.2.1, partitioning was applied intuitively by using the block I to reduce the number of explicit constraints and variables. Similarly, in a simplex algorithm (or the nonlinear optimization of Section 4.2.2), partitioning uses a partial basis, G_B, that spans the columns of $[\, G \mid 0 \,]$. Let G_N denote the remainder of the columns. $A = \left[\begin{array}{c|c} G_B & G_N \\ \hline H_B & H_N \end{array}\right]$, $b = \begin{pmatrix} g \\ h \end{pmatrix}$. Simplex calculations are speeded by the ease with which solutions, x, to the systems $G_B x = \alpha$ or $x G_B = \beta$ can be found using specialized network algorithms (where α or β represents any quantity arising in the partitioning). An implementation of this method by Barr, Farhangian and Kennington [7] for network optimization with additional general constraints resulted in code approximately twice as fast as MINOS and XMP, two popular LP solvers. Kennington et al. describe a primal partitioning method directly adapted to MCNF [5].

4.3.1 Dual Partitioning Methods

In a dual simplex method, $x_B \geq 0$ is temporarily relaxed. If a variable x_B is negative, it can leave the basis, i.e. be placed in the set of variables for which the nonnegativity property is enforced. In the dual partitioning method [29], the negative basic slack variables considered first for possible enforcement are those which are violated by the most recent network solution. Iterations of dual partitioning are described in Tableaux G, and H, in Subsection 4.3.3, as it is compared to Rosen's partitioning method.

4.3.2 Rosen's Partitioning Algorithm

Rosen's partitioning algorithm [59] uses the basis to reduce the problem explicitly, as in Section 4.2.2. The constraint matrix may be replaced by $\bar{A}x = \bar{b}$, where

$$\bar{A} = \left[\begin{array}{c|c} I & G_B^{-1} G_N \\ \hline 0 & H_N - H_B G_B^{-1} G_N \end{array}\right]$$

and

$$\bar{b} = \begin{pmatrix} G_B^{-1} g \\ h - H_B G_B^{-1} g \end{pmatrix}.$$

Ignoring the non-negativity constraints $x_B \geq 0$, the reduced problem is:

Linear Program Formulation 12

$$\sum_i c_i' x_{iN}' = \min \sum_i c_i' x_{iN} : \sum_i H_i' x_{iN} = h_0', \; x_{iN} \geq 0$$

where

$$c_i' = c_{iN} - c_{iB} G_{iB}^{-1} G_{iN}$$

$$H_i' = H_{iN} - H_{iB} G_{iB}^{-1} G_{iN}$$

$$h' = h - H_B x_B.$$

The algorithm must ensure that the flow adjustments $-G_{iB}^{-1} G_{iN} x_{iN}'$ suggested by the reduced problem are applied back to the networks. After adjustment, the basic variables of the subproblem satisfying $x_{iB}' = x_{iB} - G_{iB}^{-1} G_{iN} x_{iN}' < 0$ are exchanged for some positive variables of the reduced problem. If one pivot reestablishes feasibility, Rosen asserts that this method follows the dual simplex algorithm. Lasdon [43] notes that the method is a dual ascent method. The two results are combined below to show that the method follows the same sequence of steps as dual partitioning.

4.3.3 Implementation of Rosen's partitioning as a Dual Simplex Algorithm

First, Rosen's partitioning method is reviewed in tableau format. Consider a block schematic tableau of Rosen's partitioning method after optimization of the reduced problem, where the last row (indexed by s) symbolizes the subproblems, the remainder (indexed by r) symbolizes the reduced problem, and A, b, c are simple mnemonics indicating current quantities (e.g. reduced cost):

Tableau A

0	0	c_N	
0	I	A_{rN}	b_r
I	A_{sR}	A_{sN}	b_s

Suppose that the solution is not optimal, i.e. the basic subproblem variables $x_s = b_s - A_{sR} b_r \not\geq 0$. Some basic variables of the reduced problem (to be indexed by R_1) will replace some negative basic variables (to receive the index 3) of the subproblem to make the subproblem feasible again. Before expressing the exchange, the preceding tableau is expanded as:

Tableau B

0	0	0	0	c_N	
0	0	I	0	A_{1N}	b_1
0	0	0	I	A_{2N}	b_2
I	0	A_{3R_1}	A_{3R_2}	A_{3N}	b_3
0	I	A_{4R_1}	A_{4R_2}	A_{4N}	b_4

The exchange, performed by premultiplying the bottom two rows by $\begin{pmatrix} A_{3R_1}^{-1} & 0 \\ -A_{4R_1}A_{3R_1}^{-1} & I \end{pmatrix}$, yields the following tableau:

0	0	0	0	c_N	
0	0	I	0	A_{1N}	b_1
0	0	0	I	A_{2N}	b_2
$A_{3R_1}^{-1}$	0	I	$A_{3R_1}^{-1}A_{3R_2}$	$A_{3R_1}^{-1}A_{3N}$	$A_{3R_1}^{-1}b_3$
$-A_{4R_1}A_{3R_1}^{-1}$	I	0	$A_{4R_2} - A_{4R_1}A_{3R_1}^{-1}A_{3R_2}$	$A_{4N} - A_{4R_1}A_{3R_1}^{-1}A_{3N}$	$b_4 - A_{4R_1}A_{3R_1}^{-1}b_3$

Note that Rosen's method does not specify how the subproblem feasibility had to be restored. A primal algorithm is suggested, but a dual simplex algorithm can be used, eventually amounting to the same premultiplication. Next, a new reduced problem is formed using Linear Program Formulation 12:

0	0	0	0	c_N	
$-A_{3R_1}^{-1}$	0	0	$-A_{3R_1}^{-1}A_{3R_2}$	$A_{1N} - A_{3R_1}^{-1}A_{3N}$	$b_1 - A_{3R_1}^{-1}b_3$
0	0	0	I	A_{2N}	b_2
$A_{3R_1}^{-1}$	0	I	$A_{3R_1}^{-1}A_{3R_2}$	$A_{3R_1}^{-1}A_{3N}$	$A_{3R_1}^{-1}b_3$
$-A_{4R_1}A_{3R_1}^{-1}$	I	0	$A_{4R_2} - A_{4R_1}A_{3R_1}^{-1}A_{3R_2}$	$A_{4N} - A_{4R_1}A_{3R_1}^{-1}A_{3N}$	$b_4 - A_{4R_1}A_{3R_1}^{-1}b_3$

A basis of the reduced problem is then restored by a premultiplication by $\begin{pmatrix} -A_{3R_1} & -A_{3R_2} \\ 0 & I \end{pmatrix}$

Tableau C

0	0	0	0	c_N	
I	0	0	0	$A_{3N} - A_{3R_1}A_{1N} - A_{3R_2}A_{2N}$	$b_3 - A_{3R_1}b_1 - A_{3R_2}b_2$
0	0	0	I	A_{2N}	b_2
$A_{3R_1}^{-1}$	0	I	$A_{3R_1}^{-1}A_{3R_2}$	$A_{3R_1}^{-1}A_{3N}$	$A_{3R_1}^{-1}b_3$
$-A_{4R_1}A_{3R_1}^{-1}$	I	0	$A_{4R_2} - A_{4R_1}A_{3R_1}^{-1}A_{3R_2}$	$A_{4N} - A_{4R_1}A_{3R_1}^{-1}A_{3N}$	$b_4 - A_{4R_1}A_{3R_1}^{-1}b_3$

Since $b_3 - A_{3R_1}b_1 - A_{3R_2}b_2 \not\geq 0$, feasibility of the reduced problem must be restored. Permuting column blocks 1 and 3, Tableau C can be summarized as:

Tableau D

0	0	$\bar{c}_N$	
0	I	$\bar{A}_{rN}$	$\bar{b}_r$
I	$\bar{A}_{sR}$	$\bar{A}_{sN}$	$\bar{b}_s$

Suppose that a series of pivots will exchange some nonbasic variables (to be indexed by N_1) for some basic variables (to be indexed by $\bar{R}_1$). First expand the preceding tableau to yield Tableau E:

Tableau E

0	0	0	$\bar{c}_{N_1}$	$\bar{c}_{N_2}$	
0	I	0	$\bar{A}_{1N_1}$	$\bar{A}_{1N_2}$	$\bar{b}_1$
0	0	I	$\bar{A}_{2N_1}$	$\bar{A}_{2N_2}$	$\bar{b}_2$
I	$\bar{A}_{sR_1}$	$\bar{A}_{sR_2}$	$\bar{A}_{sN_1}$	$\bar{A}_{sN_2}$	$\bar{b}_s$

Pivoting yields the tableau:

Tableau F

0	$-\bar{c}_{N_1}\bar{A}_{1N_1}^{-1}$	0	0	$\bar{c}_{N_2} - \bar{c}_{N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	
0	$\bar{A}_{1N_1}^{-1}$	0	I	$\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	$\bar{A}_{1N_1}^{-1}\bar{b}_1$
0	$-\bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}$	I	0	$\bar{A}_{2N_2} - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	$\bar{b}_2 - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{b}_1$
I	$\bar{A}_{sR_1}$	$\bar{A}_{sR_2}$	$\bar{A}_{sN_1}$	$\bar{A}_{sN_2}$	$\bar{b}_s$

Next, Rosen's partitioning is compared with the dual simplex method. First, suppose a dual simplex algorithm was used to process the initial tableau, Tableau A, transforming it into:

Tableau G

0	0	c_N	
0	I	A_{rN}	b_r
I	0	$A_{sN} - A_{sR}A_{rN}$	$b_s - A_{sR}b_r$

which, once expanded consistently with Tableau B, yields:

Tableau H

0	0	0	0	c_N	
0	0	I	0	A_{1N}	b_1
0	0	0	I	A_{2N}	b_2
I	0	0	0	$A_{3N} - A_{3R_1}A_{1N} - A_{3R_2}A_{2N}$	$b_3 - A_{3R_1}b_1 - A_{3R_2}b_2$
0	I	0	0	$A_{4N} - A_{4R_1}A_{1N} - A_{4R_2}A_{2N}$	$b_4 - A_{4R_1}b_1 - A_{4R_2}b_2$

Since $b_3 - A_{3R_1}b_1 - A_{3R_2}b_2 \not\geq 0$, dual pivots to restore feasibility apply to row 3 of the preceding tableau. To simplify the algebraic description of further pivots, notice that rows 2 and 3 of Tableau H are the same as rows 2 and 1 of Tableau C. The similarity is enhanced by permuting rows 1 and 3 of Tableau H, obtaining a tableau which by comparison with Tableau D can be written as:

Tableau I

0	0	$\bar{c}_N$	
0	I	$\bar{A}_{rN}$	$\bar{b}_r$
I	0	$\bar{A}_{sN} - \bar{A}_{sR}\bar{A}_{rN}$	$\bar{b}_s - \bar{A}_{sR}\bar{b}_r$

Tableau I, expanded consistently with Tableau D, yields the following:

Tableau J

0	0	0	$\bar{c}_{N_1}$	$\bar{c}_{N_2}$	
0	I	0	$\bar{A}_{1N_1}$	$\bar{A}_{1N_2}$	$\bar{b}_1$
0	0	I	$\bar{A}_{2N_1}$	$\bar{A}_{2N_2}$	$\bar{b}_2$
I	0	0	$\bar{A}_{sN_1} - \bar{A}_{sR_1}\bar{A}_{1N_1} - \bar{A}_{sR_2}\bar{A}_{2N_1}$	$\bar{A}_{sN_2} - \bar{A}_{sR_1}\bar{A}_{1N_2} - \bar{A}_{sR_2}\bar{A}_{2N_2}$	$\bar{b}_s - \bar{A}_{sR_1}\bar{b}_1 - \bar{A}_{sR_2}\bar{b}_2$

After this change of notation, the series of dual pivots, identical to those performed on Tableau E, yields the tableau:

Tableau K

0	$-\bar{c}_{N_1}\bar{A}_{1N_1}^{-1}$	0	0	$\bar{c}_{N_2} - \bar{c}_{N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	
0	$\bar{A}_{1N_1}^{-1}$	0	I	$\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	$\bar{A}_{1N_1}^{-1}\bar{b}_1$
0	$-\bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}$	I	0	$\bar{A}_{2N_2} - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	$\bar{b}_2 - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{b}_1$
0	$\mathcal{A}_1$	0	0	$\mathcal{A}_2$	β

where

$$\mathcal{A}_1 = -\left(\bar{A}_{sN_1} - \bar{A}_{sR_1}\bar{A}_{1N_1} - \bar{A}_{sR_2}\bar{A}_{2N_1}\right)\bar{A}_{1N_1}^{-1},$$

$$\mathcal{A}_2 = \bar{A}_{sN_2} - \bar{A}_{sR_1}\bar{A}_{1N_2} - \bar{A}_{sR_2}\bar{A}_{2N_2} - \left(\bar{A}_{sN_1} - \bar{A}_{sR_1}\bar{A}_{1N_1} - \bar{A}_{sR_2}\bar{A}_{2N_1}\right)\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2},$$

$$\beta = \bar{b}_{sN_2} - \bar{A}_{sR_1}\bar{b}_1 - \bar{A}_{sR_2}\bar{b}_2 - \left(\bar{A}_{sN_1} - \bar{A}_{sR_1}\bar{A}_{1N_1} - \bar{A}_{sR_2}\bar{A}_{2N_1}\right)\bar{A}_{1N_1}^{-1}\bar{b}_1.$$

Simple algebraic calculations show that Tableau K is identical to Tableau L below, which would be obtained by bringing Rosen's partitioning's final Tableau F to standard form:

Tableau L

0	$-\bar{c}_{N_1}\bar{A}_{1N_1}^{-1}$	0	0	$\bar{c}_{N_2} - \bar{c}_{N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	
0	$\bar{A}_{1N_1}^{-1}$	0	I	$\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	$\bar{A}_{1N_1}^{-1}\bar{b}_1$
0	$-\bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}$	I	0	$\bar{A}_{2N_2} - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}$	$\bar{b}_2 - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{b}_1$
I	Λ_1	0	0	Λ_2	β_2

where

$$\Lambda_1 = \bar{A}_{sR_1} - \bar{A}_{sN_1}\bar{A}_{1N_1}^{-1} + \bar{A}_{sR_2}\bar{A}_{2N_1}\bar{A}_{1N_1}^{-1},$$

$$\Lambda_2 = \bar{A}_{sN_2} - \bar{A}_{sN_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2} - \bar{A}_{sR_2}\left(\bar{A}_{2N_2} - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{A}_{1N_2}\right),$$

$$\beta_2 = \bar{b}_s - \bar{A}_{sN_1}\bar{A}_{1N_1}^{-1}\bar{b}_1 - \bar{A}_{sR_2}\left(\bar{b}_2 - \bar{A}_{2N_1}\bar{A}_{1N_1}^{-1}\bar{b}_1\right).$$

The identity of the last two Tableaux proves the following property:

Property 1

If in the subproblems of Rosen's partitioning method, a dual algorithm restores feasibility, every step of the method corresponds to a step of dual partitioning.

Even if the feasibility of the subproblems is restored by a primal problem, the following holds:

Property 2

Every step of the reoptimization of the reduced problem of Rosen's partitioning can be identified with one of the steps of a dual partitioning.

Thus, the results of Grigoriadis and White [29] should prefigure our implementation.

Chapter 5

Decomposition Methods

Decomposition methods have been widely used to solve MCNF. Although the implementation of the thesis is based on partitioning, it also applies some ideas proposed in decomposition methods. This chapter also unifies ideas used in state of the art MCNF methods, e.g. sliding logarithmic barrier methods and augmented Lagrangians, with the partitioning method.

5.1 Inner and Outer Approximations and Simplicial Decomposition

In Section 4.1, the objective function with its barrier penalty function was treated as a general convex function, $F_\mu(x) = cx - \mu \sum_j \ln(h_j - H_j x)$, and solution methods were described in Section 4.2. In this chapter, alternative nonlinear programming methods are reviewed, such as Frank and Wolfe's algorithm [23] or simplicial decomposition [31,66]. To solve $\{\min_x F_\mu(x) : Gx = g,\ x \geq 0\}$ by simplicial decomposition requires solving two approximations simultaneously: an inner approximation, and a tangential (or outer) approximation. The inner approximation of F_μ, given by selecting some feasible points x^τ, $\tau = 1, \ldots, t$, is $\min F(x) | x \in conv\{(x^\tau), \tau = 1, \ldots, t\}$, often presented explicitly as a Master problem:

Problem Formulation 1

$$\min_{\rho \geq 0, x^{t+1}} F_\mu(\hat{x}^t)$$

subject to

$$\sum_{\tau=1}^{t} \rho_k^\tau = 1, \quad k = 1, \ldots, K,$$

$$\sum_{\tau=1}^{t} \rho_k^\tau x_k^\tau = \hat{x}_k^t, \quad k = 1, \ldots K.$$

The tangential approximation of F_μ at $\hat{x}^t$ is $F_\mu(\hat{x}^t) + \nabla F_\mu(\hat{x}^t)(x - \hat{x}^t)$, which yields the following relaxation of the problem:

Linear Program Formulation 13 *Relaxed (Sub)problem*

$$F_\mu(\hat{x}^t) + \nabla F_\mu(\hat{x}^t)(x^{t+1} - \hat{x}^t) = \min_x F_\mu(\hat{x}^t) + \nabla F_\mu(\hat{x}^t)(x - \hat{x}^t)$$

subject to

$$Gx = g,$$

$$x \geq 0,$$

where $F_\mu(\hat{x}^t)$ and $\nabla F_\mu(\hat{x}^t)\hat{x}^t$ are constant terms. In the case of MCNF, the subproblems described in Formulation 13 can be solved as K independent minimal cost network flow problems. Each relaxation makes use of points generated by the other one. When both approximations yield a common estimate, the optimal value has been found.

Various implementations of this general scheme consider a relaxation of the master problem by restricting the set of feasible points, x^τ, $\tau = 1, \ldots, t$. In Reduced Simplicial Decomposition [31] (RSD), the number of points x^τ that are retained to resolve the master problem is limited by a parameter r. As long as the limit r is not reached, the method is identical to the original simplicial decomposition of Hohenbalken [66]. It has been found that the convergence rate improves as r increases, with the tradeoff that the master problem becomes larger and correspondingly more time consuming to solve.

When the size of the vertex set is constrained to 1, the method coincides with the classical decomposition algorithm of Frank and Wolfe [23].

The network subproblems of MCNF, which represent the outer approximation, are relatively easy to solve accurately. On the other hand, the inner approximation, which forms the master problem, can itself be approximated by a further inner approximation without hampering the use of the method. Thus, the master problem is generally approximated in the implementations reviewed in this chapter.

5.2 Using Decomposition with a Sliding Logarithmic Barrier Function

Meyer and Schultz [60] apply RSD to logarithmic barrier functions arising from block angular problems such as multicommodity network flow problems. The principle is implemented in Section 6.1.3.

Linear Program Formulation 14

Minimize

$$cx + Mv$$

subject to

$$Gx = g$$

$$Hx + Iy - Iv = h$$

$$x, y, v \geq 0,$$

where v is a vector of artificial variables that vanish at the optimum, due to their large objective coefficient, M.

As in Section 4.1, replacing the nonnegativity requirements $y \geq 0$ by a logarithmic barrier function yields the compact formulation:

Minimize

$$cx + Mv - \mu \sum_i \ln (v_i + h_i - H_i x)$$

subject to

$$Gx = g$$

$$x, v \geq 0.$$

A pure network flow algorithm produces a feasible solution. Instead of explicitly introducing v as an artificial variable, Meyer and Schultz modify these variables parametrically, using a modified barrier function: $-\mu \sum_{i=1}^{I} \ln (h_i - H_i x)$, where $\hbar = h + v$ and at iteration t, $\mu_t = 10/2^t$. The objective function is depicted graphically in Figure 5-1. If the problem has a solution, the value of $\hbar^t$ is decreased, as:

$$\hbar_i^{t+1} = \begin{cases} h_i & \text{if } H_i x^t < h_i \\ H_i x^t + \Delta & \text{if } H_i x^t \geq h_i \end{cases},$$

where $\Delta > 0$ is a constant. (Meyer and Schultz propose $\Delta = 1$.) Various updates of μ have been proposed. A popular one used in the primal dual method of McShane et al. [46] is $\mu = \frac{b\pi^T - c^T x}{n^2}$ (where π represents the dual values), a scaled duality gap which converges to zero as the solution becomes optimal.

When $\hbar$ reaches h, a feasible solution of the MCNF problem has been found.

For each value of μ, the authors minimize F_μ approximately using a method similar to RSD, with a Master problem:

Problem Formulation 2 *Minimize*

$$cx - \mu \sum_i \ln (h_i - H_i x_k^t)$$

subject to

$$\sum_{\tau=1}^{t} \rho_k^\tau = 1, \quad k = 1, \ldots, K,$$

$$\sum_{\tau=1}^{t} \rho_k^\tau x_k^\tau = \hat{x}_k^t, \quad k = 1, \ldots K.$$

This master problem will be compared with that of Section 5.3.3.

5.3 Penalty Methods

The objective function F_μ may arise from a barrier function, or alternatively from a penalty method, a classical solution method reducing constraint violations. Penalty functions Φ are distinct from barrier functions, which prevent violations, yet they receive similar algebraic treatment [26].

$$\min_{x \geq 0} F_\mu(x) = cx + \Phi_\mu(h - Hx)$$

subject to

$$Gx = g.$$

Quadratic functions have been widely used to impose penalties. A quadratic penalty $\Phi_\mu(x) = \mu|(h - Hx)^+|^2$ as shown in Figure 5-2 has been applied to large scale multicommodity transshipment problems [62]. Note that when a constraint is feasible, the corresponding penalty is zero. The quadratic penalty is analyzed in the more general framework of augmented Lagrangian methods in Section 5.4.

5.3.1 Lagrangian Relaxation

Lagrangian relaxation is a special case of a linear penalty method: $\Phi_\mu(x) = \mu(h - Hx)$. Unlike other penalty functions, Φ_μ not only penalizes constraint violations, but also rewards feasibility. The optimum is found at a saddle point:

$$\max_{\lambda \geq 0} \min_{x \geq 0} cx - \lambda(h - Hx)$$

subject to

$$Gx \leq g.$$

(The Lagrangian multiplier will be denoted λ instead of μ, because both λ and μ are used with the Augmented Lagrangian of Section 5.4.) As in all penalty methods, the constraints $Hx \leq h$ of the original problem are permitted to be violated during interim stages of solution. Through increases in the Lagrange multipliers λ_i, the cost of violations is increased as necessary for all of the constraints to eventually be satisfied. Section 5.3.2 describes Dantzig-Wolfe decomposition, a classical implementation of Lagrangian relaxation.

Combining Phases Via "Big M" Method

Lagrangian Relaxation is used implicitly in the popular "Big M" method of finding an initial feasible point. (see for instance Sections 3.1.2 and 5.2.) In its simplest form, M is the unique value of the Lagrangian multipliers. A widely used variant is to decrease M as infeasibility decreases. More sophisticated versions of the "Big M" method resembles typical implementations of Lagrangian relaxation, particularly when feasibility is obtained in several phases. For instance, with MCNF, one phase could search for a set of flows that satisfy mutual capacity constraints[1], but that may not satisfy the network flow conservation constraints. A second phase would search for a solution that also satisfies the conservation constraints, without considering the cost of the solution. A last phase in which the true costs are used would follow.

By allowing simultaneous rather than phased relaxation, the optimization problem may travel in a less constrained way towards the optimum, thus finding a solution more quickly. On the other hand, keeping the phases separate has some advantages. Firstly, each phase may preserve the nature of a subproblem, such as a pure network flow problem. Secondly, if no feasible solution is reached, a final unconstrained solution may not provide much information about the cause of the infeasibility (see also Section 7).

5.3.2 Implementation by Dantzig-Wolfe Decomposition

This classical technique [20, 38] has been widely implemented for Lagrangian relaxation. The following paragraph shows informally that Dantzig-Wolfe Decomposition can be viewed as a specialization of RSD to the dual function $F(x) = \max_{\lambda \geq 0} cx + \lambda(Hx - h)$. For a formal treatment, see Rockafellar [58]. Since a minimum of the function is sought, only finite values of the maximum need to be considered, i.e. the minimization can be restricted to $X = \{Hx - h \leq 0\}$. Over X, $F(x) = cx$, and the master problem becomes:

Linear Program Formulation 15

$$\min_{\rho \geq 0, \hat{x}^t} c\hat{x}^t$$

[1] See Section 7

subject to

$$\sum_{\tau=1}^{t} \rho_k^\tau H x_k^\tau \leq h$$

$$\sum_{\tau=1}^{t} \rho_k^\tau = 1, \quad k = 1, \ldots, K,$$

$$\sum_{\tau=1}^{t} \rho_k^\tau x_k^\tau = \hat{x}_k^t, \quad k = 1, \ldots K.$$

Let λ^t determine a subdifferential of F at $\hat{x}^t$: $\nabla F(\hat{x}^t) = c + \lambda^t H$; then $F(\hat{x}^t) + \nabla F(\hat{x}^t)(x - \hat{x}^t) = c\hat{x}^t + \lambda^t(H\hat{x}^t - h) + (c + \lambda^t H)(x - \hat{x}^t) = cx + \lambda^t(Hx - h)$. As an application of the RSD algorithm, one gets:

Linear Program Formulation 16 *Subproblem*

$$\min_x cx + \lambda^t(Hx - h)$$

subject to

$$Gx = g,$$

$$x \geq 0.$$

The master problem has much fewer constraints than the original problem. A new set of prices, λ^t are associated with each arc and are given to the subproblems via the reduced costs $c' = c + \lambda^t H$. The solution of the subproblems produces a new point for the master problem. Thus, in the MCNF literature, this method is called Price Directive Decomposition [38].

5.3.3 Dantzig-Wolfe Decomposition by Interior Point Methods

Goffin et al. [27] describe an algorithm for solving nondifferentiable optimization problems which they apply to nonlinear MCNF. In the restrictive context of linear MCNF, this method can be viewed as solving the master problem by an interior point method, namely:

Linear Program Formulation 17

$$\min_{\rho \geq 0, \hat{x}^t} c\hat{x}^t - \mu \sum_i \ln y$$

subject to

$$\sum_{\tau=1}^{t} \rho_k^\tau H x_k^\tau + y = h$$

$$\sum_{\tau=1}^{t} \rho_k^\tau = 1, \quad k = 1, \ldots, K,$$

$$\sum_{\tau=1}^{t} \rho_k^\tau x_k^\tau = \hat{x}_k^t, \quad k = 1, \ldots K.$$

By elimination of the slack variables, the problem becomes:

Problem Formulation 3

$$\lim_{\mu \to 0} \min_{\rho \geq 0, \hat{x}^t} c\hat{x}^t - \mu \sum_i \ln (h_i - \rho_k^\tau H_i x_k^\tau)$$

subject to

$$\sum_{\tau=1}^{t} \rho_k^\tau = 1, \quad k = 1, \ldots, K,$$

$$\sum_{\tau=1}^{t} \rho_k^\tau x_k^\tau = \hat{x}_k^t, \quad k = 1, \ldots K.$$

Note the resemblence between Problem Formulation 3 and Problem Formulation 2, used for the sliding barrier method [60]. As in the simplex algorithm, dual variables λ^{t+1} can be calculated. They are solutions of an Interior Point method formulation of the dual master problem (Problem Formulation 3):

Linear Program Formulation 18

$$\max_x \sigma$$

subject to

$$\sigma \leq cx^\tau + \lambda(Hx^\tau - h) \quad \tau = 1, \ldots, t,$$

which can be obtained by applying a logarithmic barrier penalty to the nonnegative slack variables:

Linear Program Formulation 19

$$\lim_{\mu \to 0} \max_{\sigma, \lambda, \varsigma} \sigma + \mu \sum_{\tau=1}^{t} \log \varsigma^\tau$$

subject to

$$\sigma = cx^\tau + \lambda^\tau (Hx^\tau - h) - \varsigma_{x^\tau}, \quad \tau = 1, \ldots, t.$$

Rather than finding the optimum of the master problem as $\mu \to 0$, Goffin et al. find a feasible solution of a master problem in which $c\hat{x}^t \leq \bar{z}$ replaces the objective function, where $\bar{z}$ is an upper bound on the value of the master problem, which is equivalent to using a value $\mu = \bar{z} - z$. An approximate dual solution λ_μ^t is then used to solve a new subproblem. The subproblems will be identical to those of the sliding barrier method [60] if the barrier parameter of the latter is:

$$\mu = \lambda_{i\mu}^t \left(h_i - H_i \hat{x}^t\right).$$

Since this condition is satisfied by the parameter μ and the optimal value λ^t of Problem Formulation 19, the subproblems of both methods will be identical if they use the same barrier parameter and solve the master problems exactly.

The preceding result shows:

Property 3

If the same barrier parameter μ is chosen to solve the master problem of the sliding barrier method as is used in the logarithmic barrier approximation of the master problem of Dantzig-Wolfe decomposition, then the methods will follow the same trajectory.

In fact, Goffin et al. use the primal formulation of the master problem (Problem Formulation 3) as they find it easier to add a column corresponding to a new point x^{t+1} than a constraint, using a method described in [49, 50]. Note that $cx + \lambda^t(Hx - h)$ is a tangential approximation of F, and not merely F_μ. Thus, every point generated by previous values participates in the approximate solution of the master problem; similarly for the shifted barrier method, the optimum found for a parameter $\mu' > \mu$ could be used to solve the master problem with parameter μ.

5.4 Augmented Lagrangian

Augmented Lagrangian methods combine nonlinear penalty functions and Lagrangian relaxation (for a detailed presentation, see Luenberger [44, page 406]). Augmented Lagrangian methods were originally designed to relax equality constraints; one way of handling the inequality constraints $\mathcal{H}x \leq h$ is to add a slack $z^T z$. Denote $\bar{x} = \begin{pmatrix} x \\ z \end{pmatrix}$ and $\mathcal{H}(\bar{x}) = Hx + z^T z - h$. MCNF is reformulated as:

Problem Formulation 4

Minimize

$$c\bar{x}$$

subject to

$$\mathcal{H}(\bar{x}) = 0,$$

$$Gx = g,$$

$$\bar{x} \geq 0.$$

The augmented Lagrangian is the function

$$F_{\mu,\lambda}(\bar{x}) = c\bar{x} + \lambda\mathcal{H}(\bar{x}) + \frac{\mu}{2}|\mathcal{H}^+(\bar{x})|^2, \text{ with } \mu > 0,\ \lambda \geq 0. \tag{5.1}$$

This function clearly has the components of both the Lagrangian relaxation and a quadratic penalty function (pure quadratic penalties for MCNF is a special case of Equation 5.1 for $z = 0$ and x including slack variables).

- The Lagrangian component of the objective function may not be convex, and the optimal λ^* may not yield a minimum x^* of the objective function $F_{0,\lambda}(x) = cx + \lambda\mathcal{H}(x)$. The penalty function makes the problem convex, for a sufficiently large value of the penalty parameter μ, and makes x^* a minimum point rather than just a stationary point.

- If only the penalty function is used, the problems tend to be ill-conditioned, since μ often must be very large. As the Lagrange multiplier λ approaches its optimal value, λ^*, the Lagrangian component of the objective function vanishes; the problem has become a pure penalty function problem, and yet excessively large values of μ have been avoided.

The explicit dependence of F on the additional variable z can be removed by noting that $z^T z$ is continuous at the boundary of the mutual capacity constraint, and can be expressed analytically as $z_i^2 = \max[0, -H_i x - \frac{\lambda_i}{c}]$. Thus, $F_{\mu,\lambda}(\bar{x})$ is simplified as follows:

$$\mathcal{F}_{\mu,\lambda}(x) = \min_x \left\{ cx + \sum_i \frac{1}{2\mu} \left[\max\{0, \lambda_i + \mu H_i x\}^2 - \lambda_i^2 \right] \right\}$$

Figure 5-3 shows a family of such augmented Lagrangian functions. Note the resemblance to the shifted barrier method of Meyer and Schultz [60] displayed in Figure 5-1. This indicates a link between shifted barrier and augmented Lagrangian functions.

Dual Methods

Since the augmented Lagrangian can be treated as a convex function, it is natural to consider the method from the dual viewpoint. Ill-conditioning is mitigated as follows: if a penalty term is incorporated into a problem, the primal problem becomes increasingly ill-conditioned as $\mu \to \infty$. The dual problem simultaneously becomes correspondingly more well-conditioned, so that the iterations that determine new values for λ become more accurate, which allows the penalty factor μ to be reduced in size.

5.4.1 Implementation of the Augmented Lagrangian Approach

To solve the MCTP (Multi-Commodity Transportation Problem), Staniec [62] uses an augmented Lagrangian penalty function which is optimized by Reduced Simplicial Decomposition. Details of the method are given next for comparison with the partitioning method implemented in Chapter 6.

Parameter update

The problem is solved as follows, given some initial multiplier λ_t, and a penalty multiplier p_t:

1. Find a point x^t that satisfies $Gx^t = g$ and lowers the value of $c(x) + \lambda^t\mathcal{H}(x) + \frac{1}{2}p_t|\mathcal{H}^+(x)|^2$.

2. Find λ^{t+1} using a first order update $\lambda_i^{t+1} = \max\{0, \lambda_i^t + p_t(H_i\hat{x}^t - h_i)\}$.

3. Determine p_{t+1}, typically in one of the following ways:

   (a) p remains constant [44],

   (b) p increases towards a finite value p_{lim} [44],

   (c) p grows towards infinity ($p_{t+1} = rp_t$, $r > 1$) [44,62],

   (d) p is increased when convergence (based on modifying λ) seems to stall [26].

Staniec [62, page 94] found that the best results were achieved for his set of MCTP problems with a small initial value of p_0, and a slow increase, $p_t = 1.2^t p_0$.
Gill, Murray & Wright [26] suggest that a complete minimization of the objective function with respect to x is not worthwhile. Since the accuracy of the multiplier λ is limited, particularly in the early stages of solution, and this restrains the rate of convergence of $\hat{x}^t$ towards x^*, it may be desirable to update λ more often.

Staniec's Implementation

Augmented Lagrangians can be solved by Reduced Simplicial Decomposition. Staniec apparently does not treat the subproblem as a product of single commodity flows x_k, therefore the index k is omitted. This implementation is slightly simplified to:

Algorithm 1 *1. Initialize*

Let $t = 0$.

Determine an initial point $\hat{x}^0$, e.g. $c\hat{x}^0 = \min_x cx \mid Gx = g, x \geq 0$.

Set $F_{p,\lambda}(\hat{x}^0) = \infty$.

2. Solve the Subproblems

$$z(x^{t+1}) = \nabla F_{p,\lambda}(\hat{x}^t)x^{t+1} = \min_x \nabla F_{p,\lambda}(\hat{x}^t)x \mid Gx = g, x \geq 0$$

If $z(x^{t+1}) = \nabla F_{p,\lambda}(\hat{x}^t)\hat{x}^t$, then the optimum for the relaxation with p has been found. In this case, if the current optimum for the relaxation $z(x^{t+1})$ is not an optimal solution of the original problem, increase p so as to make violations more expensive; otherwise, STOP.

3. Update the Master problem

$$F_{p,\lambda}(\hat{x}^{t+1}) = \min F_{p,\lambda}(x) | x \in conv(X^t)$$

If $|X^t| = r$, then discard from X^t the vertex corresponding to $t_0 = \arg\min_{\tau=1,\ldots,r} \rho^\tau$ (See Section 5.1 for a definition of ρ^τ).

Discard from X^t the points x^τ for which $\rho^\tau = 0$. Increment t, and go to Step 2.

Every r iterations, perform a resource allocation, $P(\hat{x}^{t+1})$ (see LP Formulation 22), using $\hat{x}^{t+1}$ as an upper bound on each arc, corresponding to a fragment of the mutual capacity apportioned to the flow of each commodity currently traversing it. If the current solution violates a mutual capacity constraint, the violation is distributed evenly over the commodities [62, page 46]. This forces the solution to periodically become feasible.

5.5 Using a Decomposition Method for Resource Allocation

Another decomposition has been applied to a variant of the formulation of MCNF [40]. In this Section, no attempt is made to analyze conspicuous parallels with Price directive decomposition (described in Section 5.3.2) or other decomposition methods.

Linear Program Formulation 20

$$\min_{x,\bar{x}} \sum_{k=1}^{K} c_k x_k$$

subject to

$$Gx = g,$$

$$H\bar{x} = h,$$

$$\bar{x} \geq x \geq 0.$$

5.5.1 Master and Subproblems

For a feasible vector $\bar{x}^t$, the K subproblems may be defined as:
Linear Program Formulation 21

$$P_k(\bar{x}_k^t): \min_{x_k} c_k x_k$$

subject to

$$Gx_k = g_k,$$

$$0 \leq x_k \leq \bar{x}_k^t.$$

The dual formulation of the subproblems is:

Linear Program Formulation 22

$$P_k(\bar{x}_k^t): \; g_k\gamma_k - \bar{x}_k^t\xi_k = \max_{\gamma_k,\xi_k} g_k\gamma_k - \bar{x}_k^t\xi_k$$

subject to

$$\gamma_k G - \xi_k \leq c_k,$$

$$\xi_k \geq 0.$$

The dual prices $\gamma_k^{t+1}, \xi_k^{t+1}$ are communicated to the master problem, given here in its dual form:

Linear Program Formulation 23

$$\min_{\bar{x},\gamma} \sum \sigma_k$$

subject to

$$H\bar{x} = h,$$

$$\sigma \geq g\gamma^\tau - \bar{x}\xi^\tau, \quad \tau = 1, \ldots t+1, \; \bar{x} \geq 0.$$

The master problem produces an allocation $\bar{x}^{t+1}$ of available resources satisfying the joint capacity constraints $H\bar{x} = h$ to be used by the subproblems.

However, $\bar{x}$ is a very large vector, having K elements for each arc in the network, therefore the master problem is very large, even if the methods reviewed for other master problems are applied. Yet, in practice, many arcs not involved in the joint capacity constraints can be eliminated in a pre-processing stage. The individual flows in such arcs would be bounded simply by the upper bounds.

5.5.2 A Decomposition Algorithm

An algorithm that uses the resource directive decomposition is as follows:

Algorithm 2

1. *Initialize the problem with a feasible set of individual network flow bounds* $\bar{x}$.

   $\bar{x}$ may be initialized using known upper bounds on individual flows, or using heuristics to satisfy $H\bar{x} = h$.

   Alternatively, resource allocation could be combined with relaxation by adjusting the constraint bounds h to $\hbar$ (see Section 6.1.3) to obtain an initial solution to the network subproblems. Penalties would be attached to the objective coefficients of the corresponding artificial variables to progressively cancel them.

2. *Solve the k subproblems.*

   A network simplex algorithm could use the solution of the previous subproblem.

3. *Generate new cuts* $\sigma_k \geq g_k \gamma_k^{t+1} - \bar{x}_k \xi_k^{t+1}$ *for the master problem, using the dual prices* γ_k^{t+1} *and* ξ_k^{t+1} *from each subproblem k.*

   Since memory resources are usually limited, and increasing the size of the master problem tends to lengthen its solution time, it is common to restrict the number of constraints to some maximum size, r. The progressive construction of the master problem of Linear Program Formulation 23 is a process dual to that forming the master problem described in Section 5.1; redundant and nonbinding constraints will be the first to be eliminated; those that are most binding will be kept.

4. *Use an interior point method to find a new point in the master problem. If its objective value is within* ϵ *of the sum of the objective values of the preceding subproblems, STOP. Otherwise, go to Step 2.*

   There is some leeway in the choice in the stopping criterion – the algorithm could require improvements of either a scalar value of ϵ, or of a factor of ϵ.

5.6 Decomposition and Chain Formulation of MCNF

The master problem of Linear Program Formulation 1 represents a chain formulation of MCNF [22]. Hence, simplex decomposition can be viewed as building this formulation dynamically as it is optimized. Barrier and penalty decompositions can also be viewed as barrier and penalty solutions of the chain formulation.

Jones et al. [33] solve the chain formulation of MCNF using an Interior Point method; in effect, they solve the master problem explicitly.

Direct application of Interior Point methods to very large problems generated by columns has been implemented for crew scheduling [18] and other set covering problems [37].

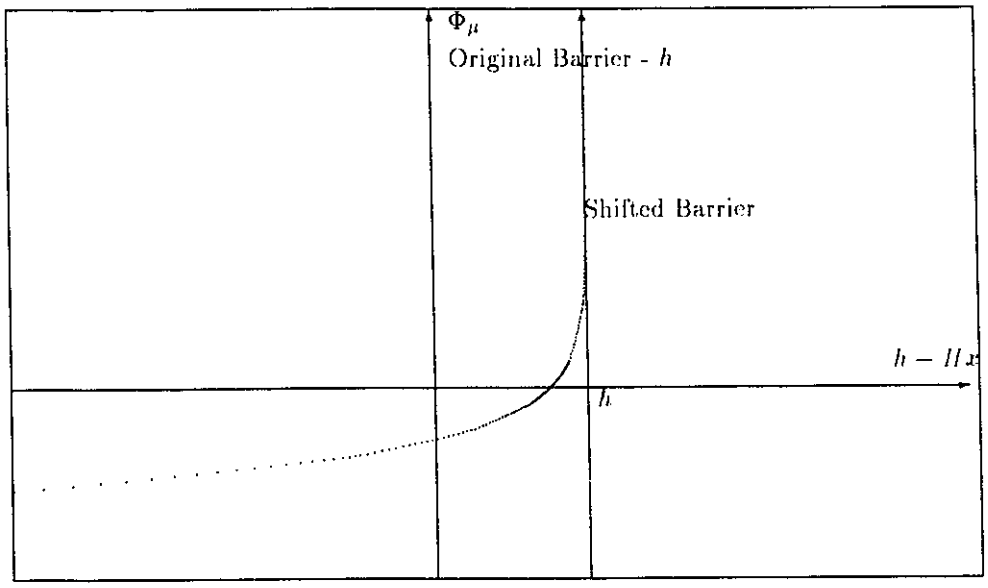


Figure 5-1: Shifted logarithmic barrier function

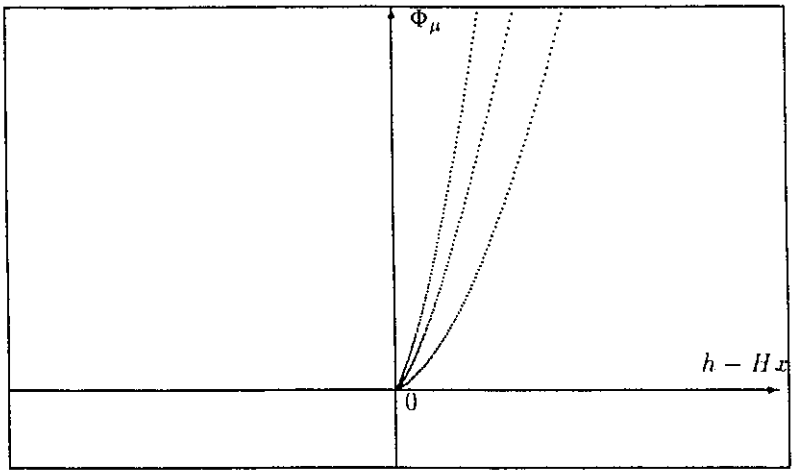


Figure 5-2: Quadratic penalty functions

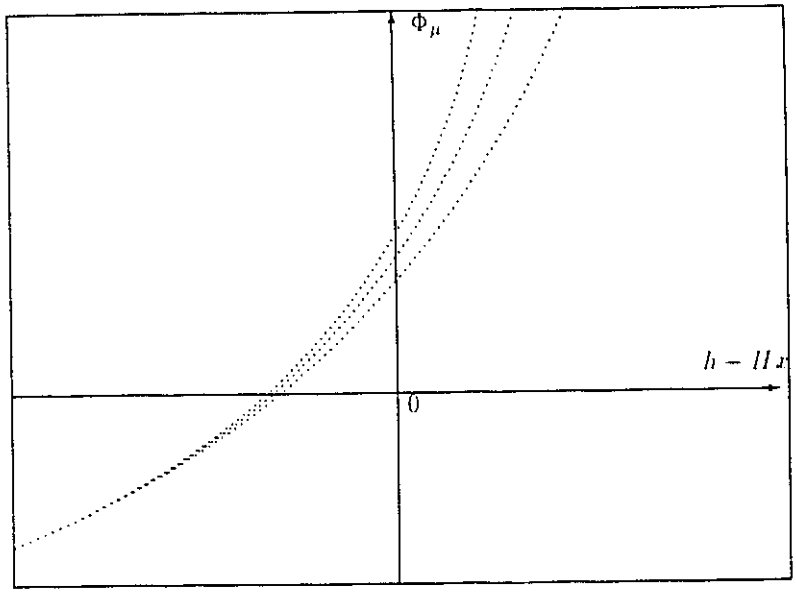


Figure 5-3: Quadratic penalties augmented by Lagrangian

Chapter 6

Implementation

This thesis proposes a simple implementation of the affine scaling method, as a basis for testing the benefit of a partitioning method. The affine scaling algorithm used in this thesis has not been designed for computational speed, because the hardware available (Sun Sparc 10, 64MB memory) does not allow testing of large scale problems over which the algorithm should have the best comparative performance. For example, Resende and Veiga [57] found that their Interior Point algorithm only outperformed the network simplex method for very large problems. Moreover, to attain this efficiency, they needed to use preconditioned conjugate gradient methods on parallel processors.
The CPU time taken by an affine scaling algorithm is roughly the product of the number of iterations and the time to perform one iteration. Improving the latter would require methods beyond the scope of this thesis.
On the other hand, comparisons based on the number of iterations are a useful indicator of the relative efficiency of the partitioning approach.

6.1 Algorithmic Strategies

6.1.1 Network Generation

It was necessary to write a network generator that would produce random multicommodity network flow problems in the form described in Appendix E.1.
Given the parameters:

- n - the number of nodes,
- K - the number of commodities,

- P - the number of paths desired for each commodity,
- L - the average path length,
- C_l, C_h - the desired range of costs for each arc,
- F_l, F_h - the desired range of flows for each path, and
- X - the number arcs added in order to augment the density of the network,

the following algorithm generates data stipulating a feasible MCNF problem, primarily using an augmenting path algorithm.

Algorithm 3 *Generate random Multicommodity Network Flow problem*

- *Set initial arc bounds to zero.*
- *For each commodity,*

  For each desired path, a random sequence of nodes $n_1, n_2, \ldots n_r$ is selected, where r is a random variable with a mean value $\bar{r} = L$, chosen from the exponential distribution and then adjusted into the integer range $[2, n]$. A flow f in the range $[F_l, F_h]$ is chosen along the path $[n_1, n_2, \ldots, n_r]$. The bound for each arc along the path is increased by f, and the supply at node n_1 and the demand at node n_r is increased by f.

- *X arcs are randomly selected and the corresponding flow bounds are augmented by a value chosen uniformly from the integer distribution $[F_l, F_h]$. In most of the test problems, no such arcs had their bounds augmented.*
- *If any nodes do not have any outgoing arcs, then one is randomly added to each such node, with a bound chosen uniformly from the integer distribution $[F_l, F_h]$, to avoid having nodes that are trivially redundant.*
- *The price for each arc is chosen uniformly from the integer distribution $[C_l, C_h]$.*

The code implementing this algorithm was written in the Perl language, and is presented in Appendix E.14.

6.1.2 Solution by Affine Scaling

The generic affine scaling step is described on page 16. Further details used in the implementation are presented in this section.

Stopping Rule

Optimality is determined using the properties of complementary slackness and dual feasibility. Let the reduced price $r(x) = c^T - A^T(AD_x^2A^T)^{-1}AD_x^2c^T$. The degree of complementary slackness is measured by setting $\gamma(x) = \max_i \{r_i(x)x_i\}$; dual feasibility is measured by $\delta(x) = -\min_i r_i(x)$.
It is shown in [64, PROPOSITION 6] that if the algorithm terminates when $\gamma(x^t) + \delta(x^t)M \leq \epsilon/n$, where M is an upper bound on the objective function value, then x^t is an ϵ-optimal feasible solution of the problem. In practice, M is an approximation of the upper bound, thus the stopping criterion is only approximate.

Upper Bounding

The algorithm can accomodate upper bounds u on the flow on each arc. It is modified using the following formulae:

$$D_x = \operatorname*{diag}_i(\min\{x_i, u_i - x_i\}),$$

$$\gamma(x) = \max_i \left\{ \max\left\{ \frac{(D_x c_p)_i}{x_i}, -\frac{(D_x c_p)_i}{u_i - x_i} \right\} \right\},$$

where e_i is a unit vector.

Initial Interior Point

In Section 3.1.2, the typical strategy used to initialize an interior point method was to choose some arbitrary vector x^0, and add an artificial variable $\dot{x}$. Typical choices for initial points are:

$$x^0 = (1, 1, 1, \ldots, 1, 1)^T \quad [64],$$

$$x^0 = \frac{1}{n}(1, 1, 1, \ldots, 1, 1)^T \quad [36],$$

$$x^0 = c\frac{||b||}{||Ac||} \quad [1].$$

For the affine scaling algorithm, Adler et al. [1] note that it is "desirable that (the) initial point be far from the facets of the polyhedral set defining the solution space." With the preceding values, x^0 may be near a facet, and convergence of the problem will suffer.
Based on the results in Section 6.3, in which several choices of initial points are examined, it is clear that the choice of the initial point affects the efficiency of the algorithm.

6.1.3 Alternative Feasibility Phases for MCNF

Several sets of artificial variables have been proposed to initialize MCNF.
In Table 6.1, I denotes the number of mutual capacity (and inequality) constraints, K the number of commodities, and n the number of nodes in an MCNF problem.

Single Artificial Variable

Many Interior Point implementations [25, 46, 64] try to alleviate the numerical burden of introducing a dense column by attempting to eliminate the artificial variable as soon as possible.

Shifted Barrier

As in Section 5.2, one artificial variable, v_i, is added for each inequality. Then the algorithm decreases it explicitly, canceling it when an interior point satisfies the constraint. The single artificial variable method of Section 6.1.3 can be used simultaneously to satisfy the equality constraints, unless a labelling procedure provides a positive flow for each arc, as proposed in [60].
In the implementation, non-slack variables are assigned the value 0.25, slack variables are initialized as:

$$y = \max\{0.25, \min\{h - 0.25He^T, u - 0.25\}\}$$

where u is a vector of upper bounds on the arcs.
Another initialization is $y = \frac{h}{n+1}$, if the value also satisfies the individual flow bound u. The performance of the alternative initializations are compared in Section 6.3.

6.1.4 Solution by Partitioning

Consider the partitioning method of Rosen [59], as discussed in Section 4.3. The following algorithm adapts it to the MCNF problem. Although provisions for upper bounding of variables are included in the computer programs, they are omitted from the equations and evoked briefly.

1. With a primal simplex algorithm, find a basic optimal solution to each of the individual minimal cost network subproblems, $\{\min c_k x_k : G_k x_k = g_k,\ x_k \geq 0\}$. The basic variables are denoted by x_{kB}, and the non-basic variables by x_{kN}.

2. Construct the reduced coupling problem, the constraints of which represent joint capacities from which the $n-1$ basic variables of each commodity network have been eliminated (see Linear Program Formulation 12).

$$\min \sum_k c_k' x_{kN}' = \min \sum_k c_k' x_{kN} = \sum_k H_k' x_{kN} = h_0',\ x_{kN} \geq 0$$

An affine scaling algorithm solves the reduced problem. This determines an estimate of values of the nonbasic variables, x_{kN} for $k = 1, \ldots K$. Let N^+ denote the index set of variables $x'_N > 0$ (actually $0 < x'_N < u$.).

3. If $x'_{kB} = x_{kB} - G_{kB}^{-1} G_{kN} x'_{kN} \geq 0$ for all $k = 1, \ldots, K$, then an optimal solution has been determined. Terminate with the optimal solution (x'_B, x'_N).

4. Otherwise feasibility of the subproblems is restored by including positive nonbasic variables $x_{kN+}(< u)$ of the reduced problem. A dual simplex method could be designed to restore subproblem feasibility. A primal pricing approach is implemented:

   For each negative component of $x'_{kB} = x_{kB} - G_{kB}^{-1} G_{kN} x'_{kN}$, let the cost $\bar{c}_{kB}$ in the subproblem be equal to a Big M value (for components x'_{kB} that exceeded the bound, let the cost $\bar{c}_{kB}$ be equal to $-M$); let the cost $\bar{c}_{kB}$ be zero otherwise. The modified subproblem k is therefore:

$$\{\min \bar{c}_{kB} x_{kB} + \bar{c}_{kN+} \cdot x_{kN+} : \; G_{kB} x_{kB} + G_{kN+} x_{kN+} = g_k, \; x_k \geq 0\}$$

   This problem can usually be solved using only a few pivot steps, based on the previous basic network solution. The primal algorithm, which for problems of non-trivial size accounted for less than 0.5% of the total solution time, provided quite adequate performance.

   Return to Step 2.

The algorithmic choices of Section 6.1.2 apply to the solution of the reduced problem, e.g. the treatment of the upper bound u.

6.2 Software Design

6.2.1 Network Generator

The network generator computer program, `mnetgen`, produces random feasible networks that were used for testing purposes, in the form described in E.1. The algorithm used is described in Section 6.1.1. `mnetgen` and several other data manipulation utilities were written in the Perl language[1], a semi-compiled scripting language that runs under UNIX and other sufficiently similar operating systems. Its user-transparent dynamic storage allocation and "associative arrays" allow easy implementation of the sparse data structures that are used in network problems. It is also a good tool for "rapid prototyping," allowing rapid development of computer programs, particularly those involving file manipulations.
The format of the output is described in Appendix E.1

[1] Practical Extraction and Reporting Language [67]

6.2.2 Affine Scaling

An affine scaling algorithm written in FORTRAN by Dr. Wun-Hwa Chen, which included routines from LINPACK, was translated to C, using a computer program called f2c, and is used to solve the reduced problem.

For comparison purposes, a computer program called solveaff was written which uses this affine scaling algorithm to solve general linear programs.

It accepts the following parameters options:

- x^0 – The default value for variables; this is also used to represent the distance that the sliding scheme backs off from infeasibility. The default value is 0.25.
- slideM – This price is used in the sliding scheme to pull the solution towards feasibility. The default value is slideM $= -1000000.0$.

6.2.3 Partitioning

Computer programs were developed in the C language to solve multicommodity network flow problems. The main computer program, called mcnf, uses the partitioning algorithm described in 6.1.4, and is listed in Appendix E.3. The functions used by mcnf include a network simplex function based on one written in FORTRAN by Shan [61]. This function was also translated to C by f2c, and then extensive work was done to make it better reflect the coding styles prevalent in C. The coordination problem was solved using the affine scaling code described in Section 6.2.2.

The default values for the parameters used by the affine scaling algorithm are as described in 6.2.2.

The format of input files are described in Appendix E.1.

6.3 Computational Experiments

The experiments assess the feasibility of affine scaling and partitioning, and the performance of algorithmic strategies for each method.

For the pure affine scaling method described in Section 6.2.2, three strategies were considered:

- Initialization of the Interior Point algorithm using the "Big M" scheme, using the point $0.25e^T$ as an initial point. This is denoted in Table 6.3 by NORMAL.
- Initialization of the Interior Point algorithm using the "Big M" scheme, using a midpoint based on the upper bound, u. The initial vector is $0.5u$. This is denoted in Table 6.3 by MIDPT.

- Initialization of the Interior Point algorithm using the "sliding barrier" scheme of Section 5.2 to initialize the slack variables, where all variables other than the slacks are initialized to 0.25. This is denoted in Table 6.3 by `SLIDE`. This initialization method combines the "sliding barrier" scheme to initialize inequalities with the "Big M" scheme to initialize the other constraints; this combined approach is unique and has not been presented elsewhere.

In the table, VARS refers to the number of variables (before any addition of artificial variables), CONS refers to the number of constraints, AI and II refer respectively to the total number of affine scaling iterations, and the number of such iterations within the infeasible region.
Similarly, three strategies were considered for solving problems using the partitioning scheme described in Section 6.1.4:

- Initialization of the Interior Point algorithm for solution of the reduced problem using the "Big M" scheme, using the point $0.25e^T$ as an initial point. This is denoted in Table 6.2 by `NORM`.

- Initialization of the Interior Point algorithm for solution of the reduced problem using the sliding barrier scheme of Section 5.2, using the point $0.25e^T$ as an initial point. This is denoted in the data table by `SLIDE`.

- Initialization of the Interior Point algorithm for solution of the reduced problem using the sliding barrier scheme of Section 5.2, using a weighted midpoint u/K as an initial point. This is denoted in the data table by `SMID`.

In Table 6.2, VARS and CONS refer respectively to the number of variables and constraints in the reduced problem. PI refers to the number of iterations of the partitioning algorithm (i.e. - the number of times that a reduced problem is constructed and solved). AI refers to the total number of affine scaling iterations required to solve the successive reduced problems. In most cases, problems were solved using all six options.

6.3.1 Data Tables

In the experimentation, it became clear that the behaviour of the affine scaling algorithm varied greatly depending on the choice of starting points. The variance was the most marked with the partitioning problems, where the "standard" sliding scheme was often several times faster than the other schemes. As was expected, increases in problem size resulted in increased iterations and CPU time for virtually all problems.

6.3.2 Pure Affine Scaling

- Some early experimentation indicated that the scheme proposed in Section 6.1.2 resulted in convergence about half as fast as when the standard starting point $e/4$ was used with the "Big M" method.

- Using an initial midpoint between lower and upper bounds provided performance very similar to the performance when the value $x^0 = e/4$ was used.

- In the sliding scheme, slack variables were treated as in Section 6.1.3. In most cases, the other two schemes outperformed the sliding scheme.

- On the average, the "midpoint" initialization scheme did best, followed by the "Big M" scheme initialized by $e/4$. The sliding scheme was consistently slowest.

- Contrary to expectations, the number of iterations before reaching a feasible point (II) is not proportional to the overall number of iterations and CPU time. For instance, in problem `c02n10b` for the "Big M" scheme, only 2 of the 29 iterations were performed in the feasible region. In contrast, for the other two schemes, virtually none of the iterations were performed in the feasible region, and yet the performance was very similar. In fact, the "midpoint" scheme, which produced points in the infeasible region up to the very end, actually found the optimum faster, and in fewer iterations.

6.3.3 Partitioning Problems

- Early partitioning experiments using either e or a random vector as an initial point had approximately the same performance, unlike the varied performances of pure Affine Scaling implementations.

- The sliding scheme converged faster than the normal "Big M" method in most cases, and solved all of the problems, contrary to either the "Big M" or the midpoint sliding schemes.

- A midpoint sliding scheme in which arc values were initialized to a weighted midpoint $u_i/(K+1)$ rarely performed as well as the "standard" sliding scheme, and tended not to work as well as the normal "Big M" method. This may be explained by the nature of the reduced problem, which supplies adjustments to the uncapacitated network flow problem, and can be expected to have more zero variables than the linear program for the complete MCNF problem.

- The experiments confirm that the partitioning scheme is a dual ascent algorithm, where $C^t_{ADJ} \geq C^{t-1}_{ADJ}$ for each iteration t, where C^t_{ADJ} is the cost of the network flows as adjusted by the solution

to the reduced problem. At each iteration t of the partitioning algorithm, $C^t_{ADJ} = C^t_N + C^t_R$ where C^t_N is the objective value associated with the latest feasible solution of the pure network flow problem, and C^t_R is the cost of the solution to the ensuing reduced problem.

The two relationships indicate that the solution estimates begin with an approximation based primarily on the optimal pure network flow solution, and converge monotonically towards the optimal feasible solution.

- In certain cases, the three initialization schemes generated different numbers of reduced problems (Linear Programming Formulation 12). This may be caused caused by degeneracy of the reduced problem, or by the lack of accuracy of the affine scaling algorithm, which yielded slightly different "optimal" solutions. These cases required more iterations to find an optimal MCNF solution, (or failed because of round-off errors). The first iteration of the partitioning algorithm was a good indicator of performance; the scheme that achieved the lowest value of C^1_P converged to the optimum faster than the other schemes.

- The number of partitioning iterations (denoted PI in Table 6.2) tended to increase with the number of commodities.

6.3.4 Comparing Partitioning to Pure Affine Scaling

- The subproblems are solved much faster than the coordination problem. For the small problem presented in Appendix E.1, only 2.75% of the time was spent solving the subproblem. In the set of test problems, as the size of the problems increased, the proportion of time spent solving the subproblems fell to less than 0.5% of the total time. Other coordination problems may yield similar results.

- The experiments showed that Interior Point methods are quite amenable to progressive changes in the cost function while searching for feasible solutions. With an Interior Point method, revising some costs will have a global and immediate effect on the direction of travel.

  In contrast, with the simplex method, in any iteration, changing costs influences the basis with a grainier effect. Thus, changes to the objective function may require a number of iterations to take effect.

  Changing the costs associated with the slack variables was used quite successfully in the implementation Section 6.1.3 of the Sliding Barrier method for initializing the partitioning method. It was not very successful when used to solve MCNF problems with the pure affine scaling method.

It may be practical to use Interior Point methods to solve problems with nonlinear objective functions by, at each iteration, presenting the method with a linear approximation of the objective function.

- Surprisingly, the number of affine iterations required to solve the reduced problem tended to be larger than the number required to solve the MCNF LP formulation. This may be explained by the high degree of degeneracy in the reduced problem. Since the reduced problem is much smaller than the full MCNF LP formulation, the CPU time was nonetheless lower. A method that better exploits the solution of the previous reduced problem might yield faster solutions.

The following graph depicts the ratio of the relative speed of fastest strategy for each method, as a function of the number of partitioning iterations. The regression line ($R^2 = 0.67$) indicates that the performance of partitioning degrades with the number of partitioning iterations. Again, a warm start that decreases the number of affine iterations required to solve the reduced problems would be necessary in order for the partitioning method to be competitive with pure affine scaling.

6.3.5 Conclusions

There was no evidence to establish clear superiority of the partitioning method over pure affine scaling or vice versa.

Table 6.4 indicates that there were cases in which partitioning was much faster than pure affine scaling, and vice versa. No conspicuous relationship between relative performance of the methods and the different aspects of problem size can be detected with the data sets tested, but the number of partitions performed strongly influences the overall computing time.

The use of differing initialization methods had clear influence on the convergence of solutions. For the partitioning reduced problem, which tends to be extremely degenerate, the shifted barrier method improved performance greatly. The shifted barrier method was less effective with the full LP formulation of MCNF.

Method	Number of Artificial Variables	Constraints Relaxed	Section
Single Artificial Variable [36]	1	All simultaneously	3.1.2
Shifted Barrier [60]	I	Mutual capacities	5.2
Backflow Arcs [35]	$\leq 2Kn$	Supply and demands	3.2.2

Table 6.1: Methods to Initialize MCNF

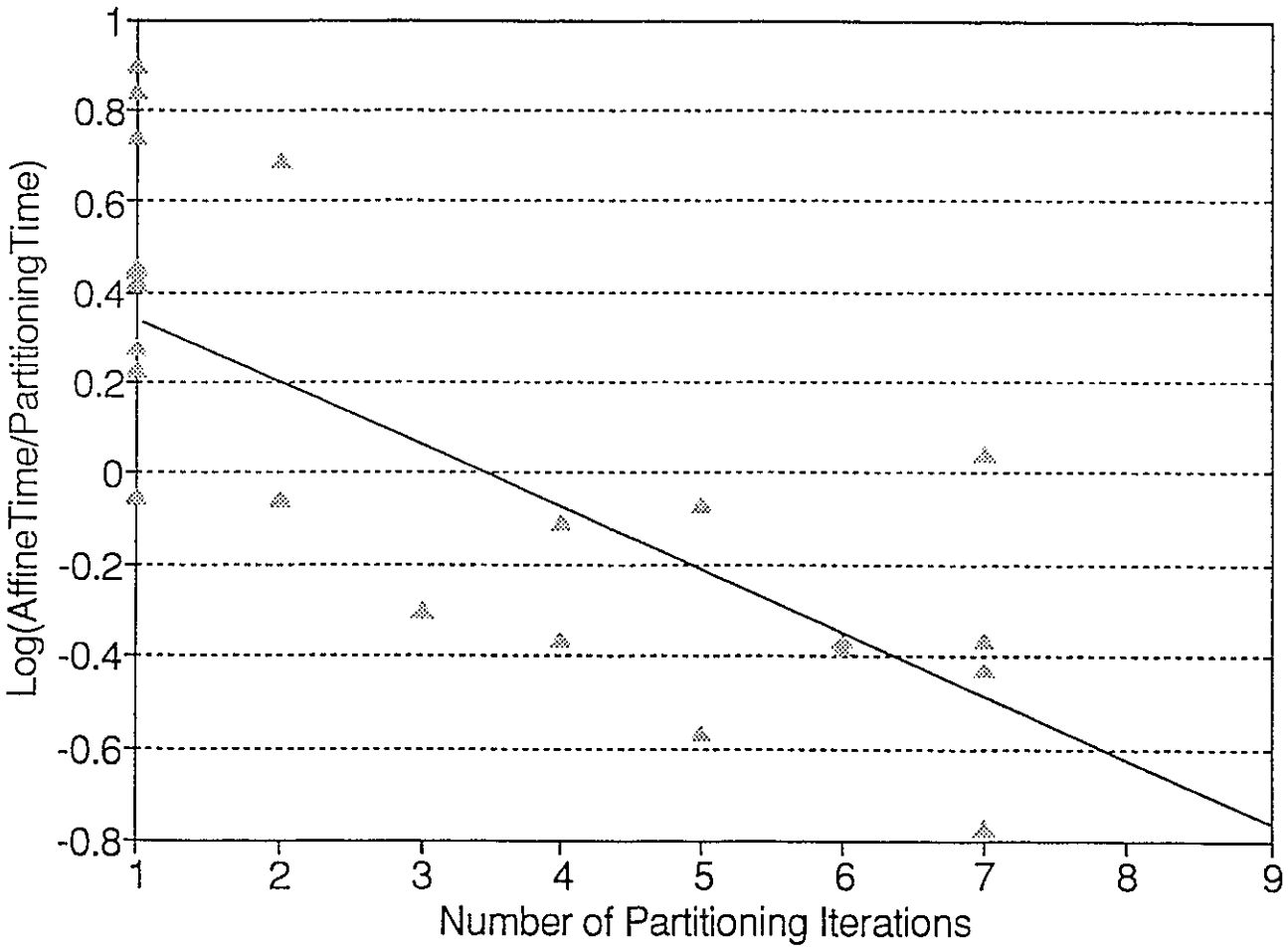
Table 6.2: Partitioning Results

Problem	Vars	Cons	NORM (s)	PI	AI	SLIDE (s)	PI	AI	SMID (s)	PI	AI
c02n03	4	4	0.031	1	10	0.028	1	9	0.043	1	9
c02n10	70	44	4.246	1	49	1.861	1	21	2.897	1	33
c02n10b	78	48	7.642	1	66	2.987	1	24	3.851	1	31
c02n14	104	65	22.445	1	80	7.703	1	27	12.647	1	44
c03n05	30	14	0.434	2	41	0.570	3	70	0.197	1	28
c03n06	42	19	0.684	2	53	1.340	3	106	1.519	4	119
c03n10	120	49	21.279	2	115	6.271	2	35	13.041	2	72
c03n10b	66	31	7.514	3	90	7.592	3	88	7.616	3	85
c03n15	138	60	74.890	4	225	70.810	4	224	104.637	6	336
c04n05	56	18	0.539	1	34	0.502	1	36	0.661	1	39
c04n08	132	40	44.076	5	264	58.029	5	289	52.488	5	307
c05n06	120	29	5.058	1	87	1.446	1	22	2.414	1	40
c05n06b	115	28	11.284	5	207	13.254	4	248	10.942	4	201
c05n10	175	44	20.450	1	86	6.226	1	25	8.209	1	34
c05n15	350	84	1661.598	9	484	1536.177	6	475	2175.337	10	704
c06n08	234	46	142.102	7	335	203.448	7	466	153.309	6	353
c07n10	336	57	650.242	7	494	1057.019	8	787	835.598	6	613
c07n10b	497	80	6083.438	7	1296	86.293	1	19	183.531	1	40
c08n08	344	50	261.813	5	289	517.556	5	565	335.512	4	368
c08n10	424	62	1306.574	9	588	1864.037	9	813	2674.599	12	1243
c08n10b	424	62	1395.538	6	650	1823.721	5	817	2288.857	11	1065
c08n15	872	123	Not Run	n/a	n/a	19399.614	7	958	FAILED	n/a	n/a
c10n10	600	69	2023.603	7	499	3844.205	10	957	5896.447	12	1473
c10n10b	820	91	5405.511	3	540	485.971	1	41	588.496	1	61
c10n15	610	75	2960.175	7	609	1978.583	7	405	2895.730	7	566
c1n100	950	1049	0.353[2]	1	0	0.388	1	0	0.366	1	0

Table 6.3: Pure Affine Scaling Results

PROBLEM	VARS	CONS	NORMAL (s)	AI	II	MIDPT (s)	AI	II	SLIDE (s)	AI	II
c02n03	8	10	0.026	9	2	0.044	19	19	0.025	8	1
c02n10	88	64	5.256	27	2	5.196	26	1	5.357	27	3
c02n10b	96	68	6.697	29	2	5.082	22	22	6.983	28	25
c02n14	130	93	23.574	31	3	20.241	26	1	21.887	29	6
c03n05	42	29	0.672	27	3	0.552	26	1	0.752	40	6
c03n06	57	37	1.390	35	2	1.284	33	1	3.546	90	4
c03n10	147	79	36.907	30	2	30.829	27	1	34.194	28	5
c03n10b	93	61	8.984	36	1	6.611	27	1	14.234	51	5
c03n15	180	105	40.703	37	1	35.606	26	0	152.537	110	3
c04n05	72	38	1.444	29	1	1.640	33	2	1.997	40	6
c04n08	160	72	21.646	33	2	18.920	29	1	50.844	78	5
c05n06	145	59	8.057	35	2	8.742	38	1	9.153	39	5
c05n06b	140	58	8.528	31	2	9.014	33	1	18.483	68	4
c05n10	220	94	53.537	42	3	49.305	41	2	FAILED	n/a	n/a
c05n15	420	159	952.548	65	1	411.994	27	1	2547.135	171	3
c06n08	276	94	78.552	38	3	61.366	30	1	298.527	146	4
c07n10	399	127	314.219	37	2	280.339	34	2	964.825	115	5
c07n10b	560	150	870.350	49	49	936.484	53	53	601.023	34	34
c08n08	400	114	444.620	63	2	223.658	31	2	744.367	113	4
c08n10	496	142	513.268	38	2	486.163	36	2	1714.296	127	5
c08n10b	496	142	693.238	48	3	575.017	39	2	3087.138	222	10
c08n15	984	243	FAILED	n/a	n/a	3297.714	36	2	FAILED	n/a	n/a
c10n10	690	169	871.055	35	3	946.588	38	2	2239.269	90	5
c10n10b	910	191	2780.472	59	59	1269.020	29	29	1968.496	45	45
c10n15	750	225	4040.062	60	2	2183.713	38	3	3237.058	56	4

Table 6.4: Performance ratio, Pure Affine vs. Partitioning, as a function of the number of Partitioning iterations (Semilogarithmic scale)



Chapter 7

Directions for Further Development

Suggestions for improving the efficiency of the algorithms have been ranked from speed-up techniques, to needs for methodological advances.

- Introducing a feasible solution using backflow arcs

  The method proposed by Kapoor and Vaidya [35] uses one artificial variable for each commodity, reflecting a "backflow" from sink to source. The addition of up to $2Kn$ arcs (where K represents the number of commodities, and n the number of nodes) increases the size of the problem, but preserves its network structure. Kapoor and Vaidya suppose that a positive circulation has been found by a labeling method. An alternative strategy is to use an Interior Point method to find a positive circulation. The following text describes a method by which the network conservation constraints may be perturbed while satisfying the joint capacity constraints.

  To each arc in the network, $Gx = g$, a nominal flow level, $x^\bullet$ is assigned. The flows are scaled to satisfy the joint capacity constraints by multiplying by a factor, ϵ, that allows $x' = \epsilon x^\bullet$ to satisfy the joint capacity constraints, $Hx' \leq h$. This represents a relaxation of the network flow conservation constraints. The value of ϵ may be determined by $\epsilon = \min_i \frac{H_{i*}^{\top}}{h_i}$. For the H structure described by Kapoor and Vaidya this may be simplified to $\epsilon = \min_i \frac{K}{h_i}$. This results in a set of flows $x' = \epsilon e$ that satisfy the joint constraints, but do not necessarily satisfy the flow conservation constraints, which have an imbalance at each node i of δ_{ik}.

  A "super-node" is then introduced for each commodity, transforming the problem into a circulation problem.

  Two arcs are then attached to the nodes that connect to the appropriate "super-node:"

- An arc is directed towards the super-node if the node is a demand node, or away from the super-node if the node is a supply node.
- A "backflow" arc maintains the "circulation" property. If the node reflects a net supply after adding the first arc, then this arc is directed to the super-node, and carries the appropriate flow to balance the flow of commodity k in node i to zero.

This results in an initial interior solution to the problem. This method could be combined with the "sliding barrier" method, allowing the backflows and the arcs representing supplies and demands to share an arc. Because of the enormous problem size that results, this method was not considered for implementation in this thesis.

- Further work to determine a "good" initial interior point.

  It was found that the performance of the interior point method depended greatly on the initial interior point. Using some of the structure of the problem to determine the initial point may assist greatly in improving convergence to the optimum.

- Warm start

  Since the reduced problem is not expected to vary tremendously between partitioning iterations, it may be possible to perform a warm start, starting the algorithm near the optimum. Warm start in the simplex and Interior Point methods are usually fledgling dual algorithms, full versions of which may be efficient.

- Determination of the barrier parameter μ

  A way of determining μ_t for Meyer & Schultz's primal formulation that does not require dual values is $\mu_t = \sum_i \hat{h}_i - h_i$. It exhibits the desirable characteristic that as the solution becomes "more feasible," μ correspondingly decreases towards a limiting value of zero. If the problem remains infeasible, μ remains correspondingly large. This represents an alternative to the formulae found in Section 5.2.

- Use of more advanced interior point methods to solve the reduced problem

  The current code is a relatively straightforward application of the Householder transformation to perform a QR reduction of the problem matrix. It does not take particular advantage of matrix sparsity; the Cholesky decomposition may take more advantage of sparsity.

  The code also does not take advantage of the fact that D_x does not change very much between iterations. Karmarkar [36, Section 6.1] describes a method to update $[AD_x^2A^T]^{-1}$ rather than recomputing it between iterations of the Interior Point algorithm.

- Use of more advanced algorithms to solve the network subproblems

  As written, the network simplex code is quite effective at solving network problems with several hundred arcs; for much larger problems, it may be more effective to use a dual simplex network algorithm, or one of the modified network simplex algorithms that has been proven to run in polynomial time, or an interior point method.

- Simplification of the coordination problem

  While it seems of little value to optimize the solution of the subproblems since they take up so little of the solution time, it seems correspondingly more worthwhile to simplify the coordination problem, perhaps first by improving the bounds for individual variables, and perhaps by eliminating redundant variables or constraints altogether.

  Chang [16] documents a set of heuristics that can be used to make matrices "more sparse," by performing a variety of reductions. This may reduce the complexity of the coordination problem.

- Interpretation of results with infeasibility

  This thesis highlights the importance of choosing an adequate initial feasible point. On the other hand, if the problem is not feasible, the various methods provide different information about infeasibility.

  If a particular MCNF problem proves to be infeasible, the values of the artificial variables may indicate why the infeasibility occurred, which may make it easier to reformulate the problem so that it is feasible. If the formulation of Kapoor & Vaidya is used, one can determine which *commodities* were involved in the infeasibility. If the formulation of Meyer & Schultz [60] is used, the results will show which *joint constraints* were forcibly violated. Both sets of information may be useful in determining the causes of infeasibility. If only a single artificial variable is introduced, there may be little interpretable information available.

Appendix A

Primal Dual Method

Megiddo [47,48] introduces the idea of combining the primal and dual problems, and using logarithmic barrier functions to solve the resulting set of constraints. Using these properties, Kojima et al [42] propose a primal-dual interior point method. The paper by McShane et. al [46] discusses the development of computer software using this method.

First consider the primal problem: [46]

Linear Program Formulation 24

$$\{\min cx : Ax = b,\ x \geq 0\}$$

Its dual is:

Linear Program Formulation 25

$$\{\min yb : yA + z = c,\ z \geq 0\}$$

The logarithmic barrier function for Formulation 24 is

$$f(x, \mu) = cx - \mu \sum_{i=1}^{n} \ln x_i$$

The optimum can be characterized for a given value of μ in the combined primal and dual system as follows:

$$D_x D_z e = \mu e$$

$$Ax = b$$

$$yA + z = c$$

The first equation along with the second assures primal feasibility; the first and third assure dual feasibility, and as $\mu \rightarrow 0$, optimality is assured.

First, let

$$v(\mu) = D_x D_z e - \mu e.$$

Using Newton's method, corrections are found to x, y and z, called $\Delta x, \Delta y, and \Delta z$, respectively.

$$\Delta x(\mu) = (D_z^{-1} - D_z^{-1} D_x A^T (A D_x^{-1} D_x A^T)^{-1} A D_z^{-1}) v(\mu)$$

$$\Delta y^T(\mu) = -(A D_z^{-1} D_x A^T)^{-1} A D_z^{-1} v(\mu)$$

$$\Delta z^T(\mu) = A^T (A D_z^{-1} D_x A^T)^{-1} A D_z^{-1} v(\mu)$$

or:

$$\Delta y^T(\mu) = -(A D_z^{-1} D_x A^T)^{-1} A D_z^{-1} v(\mu)$$

$$\Delta z(\mu) = \Delta y(\mu) A$$

$$\Delta x(\mu) = D_z^{-1} v(\mu) - D_z^{-1} D_x \Delta z^T(\mu)$$

x, y, z are updated by subtracting some multiple α of the Δ values, choosing α such that x and z remain positive. μ must converge towards zero; to that end, the authors use

$$\mu_{n+1} = \frac{by - cx}{n^2}$$

which is a scalar duality gap. As the solution approaches optimality, the primal cost cx approaches by, so $\mu \rightarrow 0$.

A.1 Initial Solution

In most methods, an initial interior point is forced by adding an artificial variable corresponding to a "known" interior point (in interior methods, this point is often e/n). Then, either a two phase method or the "Big M" Method is used to find the optimum.

McShane et al. need to add a constraint for the dual formulation. They then use the "Big M" method in order to find the optimum (See Section 6.1.3), and propose using values of:

$$M_c = 30n^2 \max |c_i|$$

$$M_b = 30n^2 \max |b_i|$$

A.2 Comparing Primal Affine Scaling with Primal-Dual Methods

Vanderbei et al. [64] use the directions in the Affine Scaling algorithm:

$$c_p = \left(I - D_x A^T (A D_x^2 A^T)^{-1} A D_x\right) D_x c^T$$

The computation is dominated by finding the solution w of the system:

$$(A D_x^2 A^T)^{-1} w = A D_x^2 c^T \tag{A.1}$$

McShane et al. [46] use the following directions in the primal-dual affine scaling algorithm:

$$c_{yp} = (-A D_z^{-1} A)^{-1} A D_z^{-1} v(\mu)$$

$$c_{zp} = -c_{yp} A$$

$$c_{xp} = D_z^{-1} v(\mu) - D_z^{-1} D_x c_{zp}$$

For the primal-dual method, the corresponding problem is to solve for s in:

$$(A D_x^2 D_z^{-2} A^T) s = A D_z^{-1} (D_x D_z e - \mu e),$$

which has approximately the same difficulty as solving Equation A.1. Therefore, the primal-algorithm can solve a larger system in the same order of computational steps as the primal affine scaling algorithm.

Appendix B

A Classical Affine Transformation to Canonical Form

This appendix presents a review of Bazaraa's [8] affine transformation that transforms an LP into *canonical* form. Consider the LP $\{\min cx : Ax = b,\ x \geq 0\}$.

B.1 Step 1: Regularizing the problem

Using Beale's classical method [9], add a constraint:

$$\sum_{j=1}^{n} x_j \leq Q$$

where Q is some known bound on the sum of the variables. Alternatively, adding a slack variable $x_{n+1} \geq 0$:

$$\sum_{j=1}^{n+1} x_j = Q \tag{B.1}$$

B.2 Step 2: Homogenization of the constraints

Add an artificial variable, x_{n+2}, identical to 1, in order to homogenize Equation B.1.

$$Ax - b^T x_{n+2} = 0 \tag{B.2}$$

$$e^T x + x_{n+1} - Q x_{n+2} = 0 \tag{B.3}$$

Note that the column based on b that is introduced to the constraint matrix will in general be dense, which may make the problem more difficult to solve by algorithms that expect sparse matrices (see Section 6.1.3).

The value $x_{n+2} = 1$ can be enforced by the additional equation:

$$e^T x + x_{n+1} + x_{n+2} = Q + 1 \tag{B.4}$$

B.3 Step 3: Variable Rescaling

The variables are normalized by $x_j^{'} = \frac{x_j}{Q+1}$ in order to transform the final constraint to

$$\sum_{j=1}^{n+2} x_j^{'} = 1 \tag{B.5}$$

Since the other equations are homogeneous, their coefficients are unchanged by this transformation. The original has then been transformed to the canonical form.

B.4 Step 4: Introducing an initial feasible solution

In order for the problem to have an initial feasible solution of the form $\frac{e}{n}$, another artificial variable $x_{n+3}^{'}$ is added such that a feasible point is $x' = \frac{e}{n+3}$. The final formulation is as follows:

Linear Program Formulation 26

Minimize

$$cx' + Mx_{n+3}^{'}$$

subject to

$$Ax' - bx_{n+2}^{'} - [Ae - b]x_{n+3}^{'} = 0,$$

$$e^T x' + x_{n+1}^{'} - Qx_{n+2} - (n + 1 - Q)x_{n+3}^{'} = 0,$$

$$e^T x' + x_{n+1}^{'} + x_{n+2}^{'} + x_{n+3}^{'} =, 1$$

$$x_j^{'} \geq 0 \; \forall j.$$

A value of M of order $O(2^Q)$ will ensure that $x_{n+3}^{'}$ is zero for an optimal solution, and therefore that the optimal solution of Formulation 26 is also a feasible solution of Formulation 2. [1]

[1] The algorithm could alternatively start with $M = \max_i (c_i)$, and double M any time that x_{n+3} does not diminish in an iteration of the algorithm.

B.5 Finding a value for Q

Theoretically, Q can be given the value

$$Q = \lceil 1 + \lg(1 + \max|c_i|) + \lg|\max(\det A)| \rceil$$

where c_i are the cost coefficients, and a's are square submatrices of the constraint matrix A. Unfortunately, the time and memory space required to compute the determinant matrices make this determination of Q impractical.

A practical value can often be determined by examining the structure of the problem under consideration. In the case of Kapoor and Vaidya's MCNF formulation, a minimal value for Q is readily available, as $Q = ce + 1$. If a particular formulation involves upper bounding, i.e. $0 \leq x \leq u$, one choice would be $Q = e^T u$.

If the value of Q cannot be easily determined by some examination of the linear program formulation, the following method could be used to find an appropriate Q:

B.5.1 Exponential Search

1. Choose an initial estimate of Q, say Q_0
2. Attempt to solve the system, using the estimate for Q_i
3. If x_{n+1} converges towards zero, then the estimate Q_i was too small. Generate a $Q_{i+1} > Q_i$, and resolve the system. A practical choice is to set $Q_{i+1} = rQ_i$, with $r > 1$. The interim results will be usable, since the only variable that needs to change to ensure feasibility of the current solution in the changed system is x_{n+1}, which is a free variable whose value depends primarily on the value of Q.

B.5.2 Optimal value of zero

There are two major approaches in the literature to achieve an optimal value of zero.

Using the Dual Problem

According to duality theory, if there is an optimal solution to the original problem, then the dual problem has the same optimal solution. The primal may therefore be combined with the dual, and by subtracting the objective functions from each other, the optimal value of 0 is achieved. Solving this problem yields both the primal and dual optimal solutions.

Unfortunately, this makes the problem much larger, since n constraints and m variables[2] are added.

Sliding Objective Function

A more attractive method of achieving an optimum of zero would be to modify the objective function so that its optimum becomes zero.

A general scheme for adjusting the objective function is as follows:

Suppose an upper bound u and a lower bound l are known for the optimal value v. An easily known upper bound u would be the best iterate of the primal LP, cx^i, and an l would be the smallest coefficient c_j. [3]

The objective function is modified by subtracting $\frac{1}{2}(u-l)$ from it; since $\sum_{j=1}^{n} x_i = 1$, $\frac{1}{2}(u-l)(\sum_{j=1}^{n})$ may instead be subtracted. This allows the objective function to remain homogeneous.

The linear program is next solved using an interior point method. There are now two possibilities:

1. The optimum lies between u and $\frac{1}{2}(u+l)$

   In this case, the search algorithm terminates, with a violation of the expectation that the optimum was $\frac{1}{2}(u+l)$. An optimal point has been found.

2. The optimum lies between $\frac{1}{2}(u+l)$ and l

   In this case, the search algorithm finds an optimum value of zero for the sliding objective function. It is not known if this value is optimal, so the upper bound is adjusted to $u_{new} = \frac{1}{2}(u_{old}+l)$, which generates a new objective function. If the value *was* optimal, then there will be no improvement in the next iteration.

[2] Where m and n are respectively the number of constraints and variables in the original problem.

[3] $\min_j (c_j)$ is a lower bound since it is trivially a minimum value of the related problem:

Linear Program Formulation 27
Minimize

$$cx$$

subject to

$$e^T x = 1$$
$$x \geq 0$$

Appendix C

Newton's Method with Steepest Ascent

This appendix describes a modified Newton's Method which has a close relationship to the partitioning method implemented in 6.1.4, in which the coordination problem is solved using an affine scaling algorithm. In this case also, Newton's Method would be used to solve the coordination problem, while a steepest ascent algorithm would be used to solve the network subproblems, providing an equivalent to the network simplex algorithm.

Consider the equality constrained problem:

Problem Formulation 5

$$\min_x F(x)$$

subject to

$$\mathcal{G}(x) = 0$$

where $\mathcal{G}(x) = Gx - g$. An unconstrained penalty problem corresponding with this is:

$$\min \mathcal{F}(x) = F(x) + \frac{1}{2} p \mathcal{G}^2(x)$$

Given an iterate x^t, let $M(x^t)$ be the subspace tangent to

$$S = \{x \ : \ \mathcal{G}(x) = \mathcal{G}(x^t)\}$$

$N(x^t)$ is the subspace generated by the gradients of $M(x^t)$, as evaluated at x^t.

Newton's method is applied to x^t over the subspace $N(x^t)$, obtaining w^t (determined below).

From w^t, a steepest descent step is taken over $M(x^t)$ in order to obtain x^{t+1}.

Algorithm 4 *Quasi-Newton Method, with a partial Steepest Ascent Step*

1. *Calculate* $\delta^t = -\frac{1}{p}\nabla\mathcal{G}(x^t)^T[\nabla\mathcal{G}(x^t)\nabla\mathcal{G}(x^t)^T]^{-2}\nabla\mathcal{G}(x^t)\nabla\mathcal{F}(x^t)^T$ [1] *as the approximation to the Newton direction.*

2. *Find* $\min_{\beta^t}\mathcal{F}(w^t)$ *where* $w^t = x^t + \beta^t\delta^t$ *(usually,* $\beta^t = 1$ *will provide an improved value). This represents the approximated Newton step in* $N(x^t)$.

3. *Calculate* $\partial^t = -\nabla\mathcal{F}(w^t)^T$. *This is the Steepest Ascent direction.*

4. *Find* $\min_{\alpha^t}\mathcal{F}(x^{t+1})$ *where* $x^{t+1} = w^t + \alpha^t\partial^t$. *This represents the Steepest Ascent step in* M.

This can be applied to the MCNF problem: $\{\min_{x,y}\{cx\} : Gx = g,\ Hx + y = h,\ x, y \geq 0\}$, after transforming into the barrier form:

$$\mathcal{F}(x,y) = cx + \mu\left[\sum_j \ln(x_j) + \sum_i \ln(h_i - H_i x)\right] + p/2(Gx - g)^2$$

[1] This involves an estimate of $L(x^t)$, which is costly to compute explicitly, since it involves all of the constraints

Appendix D

Random Choice of Starting Point

This appendix proposes a method of choosing a random starting point x^0 for the affine scaling algorithm applied to the linear program $\{\min cx : \ Ax = b, \ u \geq x \geq 0\}$. It is augmented by introducing an artificial variable, $\dot{x}$, resulting in the following formulation:

$$\{\min cx + M\dot{x} : \ Ax + a\dot{x} = b, \ u \geq x \geq 0, \dot{x} \geq 0\}$$

$\dot{x}^0$ is chosen to be 1, and receives a very large cost, M, such that at optimality, $\dot{x}^* = 0$.
In most cases, this should provide a point that is distant from the facets of the feasible region.
The initial point will be chosen from a uniform random distribution. An appropriate range must be selected to ensure that floating point overflow is avoided. Consider a maximum permissible floating point value of $F_{\max}$. The arithmetic operations involved in most interior point methods, including the method used in this thesis, are summarized by:

$$x^{k+1} = x^k - \frac{\alpha}{\lambda} D_{x^k} P_{x^k} D_{x^k} c,$$

$$P_x = I - D_x A^T (A D_x^2 A^T)^{-1} A D_x.$$

Let $MAX = \sqrt{F_{\max}}/n$. The following conditions are sufficient for x_i^0 to avoid overflow for the calculation of $(AD_x^2 A^T)$ and $D_{x^k} P_{x^k} D_{x^k} c$:

$$|A_{ij}| x_i^0 \leq MAX \text{ for all } j$$

$$|c_i| x_i^0 \leq MAX$$

Since $\dot{x}^0 = 1$, the value chosen for M is MAX.

To respect the additional bound,

$$x_i^0 \leq u_i,$$

an upper range for the uniform distribution should be $x_i^0 < m_i$, where

$$m_i = \min\left(u_i, \frac{MAX}{\max\{|c_i|, \max_j |a_{ij}|\}}\right).$$

It is similarly necessary that x_i^0 not be too small, both to avoid overflow in the solution of $(AD_x^2A^T)s = AD_x$, and to prevent the initial point from being too close to the faces of the polytope $\{x_i^0 \geq 0\}$, therefore, the points must satisfy the lower bound condition:

$$x_i^0 \geq \epsilon_i$$

A reasonable choice for the lower bound ϵ_i would be:

$$\epsilon_i = \frac{m_i}{n^2}$$

a, the vector of constraint coefficients of the artificial variable, is typically chosen as $a = b - Ax^0$. This method was implemented in the function `InitialSolution()` described in Appendix E.13, and tested on various problems. Since it did not perform as well as the "normal" initialization, $x^0 = (1, 1, 1, \ldots, 1, 1)^T$, on any of the problems solved, it was not considered for the final implementation.

Appendix E

Software Documentation

E.1 Data Format

Numbers may be input in a free-form format. For instance, the following inputs would be treated as entirely equivalent:

```
1 2 3 4
```

```
1.0 2.0e03.0000000000400e-02
```

Note that combinations of white space and line breaks are used to delimit numbers.

A problem to be solved by `MCNF` appears in the following form, illustrated by reference to a problem found in Helgason & Kennington [38, page 147].

- Number of arcs, nodes, and commodities.

  For example:

  ```
  9 6 2
  ```

- For each arc:

  - Head node, tail node
  - Costs for flow of each commodity
  - Flow bound for the arc

  For example:

  ```
  0 3 1 4 2
  ```

This describes an arc going from node 0 to node 3, with an upper bound of 2, with costs of 1 and 4 associated with the respective commodities.

- For each node, net supply (demand) for each commodity. For example:

```
2 2
```

This describes a node which supplies 2 units of both commodities.

The sample problem appears as follows:

```
9 6 2
 0 3 1 4 2
 0 4 8 2 3
 0 5 9 8 3
 1 3 10 3 3
 1 4 1 3 3
 1 5 4 2 3
 2 3 4 18 3
 2 4 10 4 3
 2 5 4 3 3
 2 2
 2 2
 2 2
 -2 -2
 -2 -2
 -2 -2
```

The output for this problem appears as follows[1]:

```
mcnf - MultiCommodity Network Flow Solver
(C) 1992 by Christopher Browne
cbbrowne@csi.uottawa.ca
$Id: proglist.tex,v 8.1 1993/03/31 18:22:56 cbbrowne Exp cbbrowne $
Determined the optimal solution
```

[1]Note that debugging information that is typically printed out has been omitted.

```
Arc: Head Tail Bound Flow:     0     1  Slack
   0 0    3   2.00   1.50  0.50  0.00
   1 0    4   3.00   0.00  1.50  1.50
   2 0    5   3.00   0.50  0.00  2.50
   3 1    3   3.00   0.00  1.50  1.50
   4 1    4   3.00   2.00  0.00  1.00
   5 1    5   3.00   0.00  0.50  2.50
   6 2    3   3.00   0.50  0.00  2.50
   7 2    4   3.00   0.00  0.50  2.50
   8 2    5   3.00   1.50  1.50  0.00
Total cost: 33.000000 (Optimum)


Time report:
Breakdown: Networks  Affine Scaling   Building Red.   Other
%     2.481      77.910 0.755    18.854
Seconds     0.002       0.075 0.001     0.018
Elapsed time: 0.095965
Size of problem:
Nodes: 6  Arcs: 9  Commodities: 2
Size of Affine Scaling Problem:
  Variables: 17  Constraints: 9
Number of partitioning iterations: 2
Number of affine scaling iterations: 27
Number of affine scaling iterations in which
the problem is infeasible : 1
Optimal cost: 33.000000
dynamic: done!
```

E.2 SOLVEAFF.C

This computer program uses the affine scaling algorithm implemented in [E.13] to solve linear programs described in matrix form in terms of A, b, c and u, solving the problem:

Linear Program Formulation 28

$$\min_x cx$$

subject to

$$Ax = b$$

$$u \geq x \geq 0$$

In order to use this solve multicommodity network flow problems, `MCNFAFF` (see [E.15]) translates problem formulations in the form described in E.1 into matrix form.

E.3 MCNF.C

This computer program is the "main program" for the partitioning algorithm described in 6.1.4. It calls a function implementing a network simplex algorithm [E.12] to solve the individual networks, constructs a reduced problem using the function `buildreducedproblem()` [E.4] and then calls an affine scaling interior point method [E.13] in order to solve the reduced problem. The results of the reduced problem either provide an optimal solution, or allow the network subproblems to be modified and resolved.

```
/* $Id: mcnf.c,v 8.0 93/01/20 18:40:35 cbbrowne Exp Locker: cbbrowne $ */
#define MAIN
#include "mcnf.h"
#include "boolean.h"
#include "linpro.h"
#include "soltype.h"
#include <sys/time.h>
#include <time.h>
/*****************************************************************
 **                          mcnf.c                             **
 **                    By Christopher Browne                    **
 **                    University of Ottawa                     **
 **          Prepared for Master of Systems Science Thesis      **
 *****************************************************************
 ** This program solves MultiCommodity Network Flow problems.   **
 ** It initially uses a network simplex method to solve the     **
 ** individual subproblems. This partitions the problem into    **
 ** sets of basic and nonbasic variables. It then uses an       **
 ** affine scaling algorithm to minimize a reduced joint        **
 ** capacity formulation. This determines a set of basic        **
 ** variables that ought to be eliminated from the basis.       **
```

```
**********************************************************************/
/*
 * PARAMETERS: - Defined in mcnf.h NODES        =        NUMBER OF NODES IN
 * NETWORK ARCS  = NUMBER OF LINKS IN NETWORK MLNK  = THE MAXIMUM
 * NUMBER OF LINKS NEEDED TO BE HANDLED =  ARCS + NODES VNODES
 *       TOTAL NODES INCLUDING "SUPERNODE" =  NODES + 1 RVARS  =
 * Number of variables in the reduced subproblem (Multiplied by NOK for
 * total) =        ARCS-NODES+1 NOK    =        COMMODITIES
 *                                                                          30
 * 2 INPUT FILES REQUIRED:
 *
 * (I) NETWORK DATA: 1 line for each "arc" (each of which involves NOK
 * commodities) Each line contains NOK+3 numbers:
 *
 * A                        <-- Node at the head B                      <--
 * Node at the tail 1 .. NOK                <-- Costs for each commodity Joint
 * Capacity        <-- Aggregate capacity of the arc
 *
 * (II) SOURCE FILE: NOK column matrix indicating supplies/demands for each  40
 * commodity Positive      ->      Supply node
 *
 * 0               ->      No net supply/demand Negative     ->      Demand node
 */

double
  *ubound = NULL, *bound = NULL, *duali = NULL, *supplyi = NULL,
  *x = NULL, *flowest = NULL, *rduals = NULL, *rbounds = NULL,
  *redcosts = NULL, *c = NULL, *b = NULL, *flowi = NULL, *costi =
  NULL;                                                                   50

double
  **flow = NULL, **mubound = NULL, **dual = NULL, **supply = NULL,
  **altsupply = NULL, **altflow = NULL, **cost = NULL, **newcost =
  NULL,          **d = NULL;

int
  *point = NULL, *headnode = NULL, *link = NULL, *tailnode = NULL;

boolean                                                                   60
  * basici = NULL, **basic = NULL, **eliminate = NULL;

/*
 * Here are some arrays used by the subproblems that need to be preserved
```

```
 * between iterations
 */
int
  *predl = NULL, *pred = NULL, *depth = NULL, *thread = NULL,
  *rthrd = NULL;

boolean
  * down = NULL;

/* The BIG arrays in which they are preserved */
int
  **spredl = NULL, **spred = NULL, **sdepth = NULL, **sthread = NULL,
  **srthrd = NULL;

boolean
  ** sdown = NULL;

int
  NOK = 0, ARCS = 0, NODES = 0;

int
  infeasibleiters = 0;

/* Important parameters */
double          mu = 0.25;
double          slideM = -10000000.0;
double          artM = 10000000.0;
double          netM = 100.0;

boolean         sliding = FALSE;
boolean         midpoint = FALSE;
boolean         randompoint = FALSE;

void
  main(int argc, char *argv[])
{
  void              mcnf(void);
  extern char       *optarg;
  int               arg;
  char              c;
  while ((c = getopt(argc, argv, "su:mrh?")) != EOF)
    switch (c) {
    case 's':    /* Sliding option ON */
```

```
      sliding = TRUE;
      break;
    case '?':                                                                    110
    case 'h':
      printf("$Id: mcnf.c,v 8.0 93/01/20 18:40:35 cbbrowne Exp Locker: cbbrowne $\n");
      printf("Options:\n");
      printf("-? or -h   Display this option list.  Depending on your shell, -? may be\na little difficult to generate.\
      printf("-s  Use the sliding option for slack variables (default = FALSE)\n");
      printf("-u  Set the value of mu (distance slid back from infeasibility\n");
      printf("       Default value: mu=%lf\n", mu);
      printf("-m  Initialize x[] with a bound midpoint (default = FALSE)\n");
      printf("-r  Initialize x[] with a random point (default = FALSE)\n");
                                                                                 120
      exit(0);
      break;
    case 'u':   /* Modify mu value */
      sscanf(optarg, "%lf", &mu);
      if (mu <= 0.0) {
        printf("Error: mu <= 0.0 - value of %lf\n", mu);
        exit(0);
      }
      if (mu > 10.0)
        printf("Warning: mu = %lf > 10.0.\n", mu);                            130
      break;
    case 'm':   /* Choose "midpoint" initialization */
      midpoint = TRUE;
      break;

    case 'r':   /* Choose "random" initialization */
      printf("Warning:   random initial point option not currently implemented!\n");
      randompoint = TRUE;
      break;
                                                                                 140
    default:
      break;
    }

  if ((randompoint && midpoint)) {
    printf("Can't use both -m and -r options.\n");
    exit(0);
  }
  mcnf();
}                                                                                150
```

```
void
  mcnf(void)
{
  /* Local variables */
  static
    int
      i, j, commodity, k, n, arc, node;

  static
    double
      optimalcost, adjustment, tflow, r_1, totalcost,
      totalflow;

  static
    int
      istate, iters, pricedout, dstart, nonbasics;

  static
    boolean
      optimalnetwork, optimal;


#include "initnetw.h"
#include "lpcode.h"
#include "dualtree.h"
#include "nsimplex.h"
#include "savenetw.h"
#include "shownet.h"
#include "dynamic.h"
#include "buildred.h"
#include "decision.h"

  extern
    void
      Round_Answer(
                   double *x,
                   double *redcosts,
                   int _MAXVARS,
                   double *rduals,
                   int _ARCS);

  extern
```

```
void
  Adjust_Network(
                boolean **basic,
                double **flow,
                double *x,
                double *bound,
                double **mbound,                                    200
                double **cost,
                double **newcost,
                double *rduals,
                double **altflow,
                int *headnode,
                int *tailnode,
                int **sdepth,
                int **spred,
                int **spredl,
                boolean **sdown,                                    210
                double *rbounds,
                double *redcosts,
                double **newsupply);

/* Time info. for use in timing... */
double
  Begintime, Endtime, nettime, buildtime, affinetime,
  othertime, totaltime;

struct timeval                                                      220
  starttime, endtime;

struct timezone
  tzp;

int             beginsecs;
int             affineiterations = 0;
int             iterations = 0;

/* Output header information */                                     230
printf("mcnf - MultiCommodity Network Flow Solver\n");
printf("(C) 1992 by Christopher Browne\n");
printf("cbbrowne@csi.uottawa.ca\n");
printf("$Id: mcnf.c,v 8.0 93/01/20 18:40:35 cbbrowne Exp Locker: cbbrowne $ \n");
printf("\n\n");
```

```
#ifdef DEBUGGING
  printf("Reading in Network:\n");
#endif
  /* READ IN NETWORK DATA FILE */                                          240
  /*
   * Starts with # of arcs # of nodes # of commodities
   */

  scanf("%d %d %d", &ARCS, &NODES, &NOK);
  printf("ARCS:        %3d\n", ARCS);
  printf("NODES:       %5d\n", NODES);
  printf("NOK:         %3d\n", NOK);

  /* Now, we malloc the double arrays */                                   250
  printf("mcnf: Allocate memory!\n");
  dynamic(ALLOCATE);

  /* Set up the decision table */
  init_Decision_table(VNODES);

  printf("Done allocation\n");

  for (node = 0; node < VNODES; node++)
    point[node] = NOTHING;                                                 260

  /*
   * Format: Head Tail   Cost1 Cost2 Cost3 ... CostK TotalBound
   */
  for (arc = 0; arc < ARCS; arc++) {
    scanf(" %d %d", &headnode[arc], &tailnode[arc]);
    /* Test for illegal values: */
    if ((headnode[arc] >= ARCS) || (headnode[arc] < -0)) {
      printf("ERROR: headnode[%d] value = %d - Out of range 0..%d\n",
             arc, headnode[arc], ARCS - 1);                                270
      exit(0);
    }
    if ((tailnode[arc] >= ARCS) || (tailnode[arc] < -0)) {
      printf("ERROR: tailnode[%d] value = %d - Out of range 0..%d\n",
             arc, tailnode[arc], ARCS - 1);
      exit(0);
    }
    for (commodity = 0; commodity < NOK; commodity++) {
      scanf("%lf", &cost[arc][commodity]);
```

```
    if (cost[arc][commodity] < 0)                                          280
      newcost[arc][commodity] = 0.0;
    else
      newcost[arc][commodity] = cost[arc][commodity];
  }

  scanf("%lf", &ubound[arc]);
  /* Check for error! */
  if (ubound[arc] < 0.0) {
    printf("ERROR: Upper bound for arc %d = %lf - Less than zero error\n",
           ubound[arc]);                                                   290
    exit(0);
  }
  'ound[arc] = ubound[arc];
  for (commodity = 0; commodity < NOK; commodity++) {
    mubound[arc][commodity] = ubound[arc];
  }

  /* Link the node in... I almost understand this... */

  link[arc] = point[headnode[arc]];                                        300
  point[headnode[arc]] = arc;

  for (commodity = 0; commodity < NOK; ++commodity) {
    basic[arc][commodity] = TRUE;
  }
}                        /* For all arcs */

/* READ IN SUPPLY/DEMAND DATA FILE */
for (node = 0; node < NODES; node++) {
  for (commodity = 0; commodity < NOK; ++commodity)                        310
    scanf("%lf", &supply[node][commodity]);
  /*
   * Put in the linking information for the artificial node -
   * this information is not, and SHOULD NOT be actually used.
   * I'm just completing the array for completeness purposes.
   */

  link[ARCS + node] = point[NODES];
  point[NODES] = node + ARCS;
}                                                                          320
/* Test supply/demands for summing to zero */
for (commodity = 0; commodity < NOK; commodity++) {
```

```
  totalflow = 0.0;
  for (node = 0; node < NODES; node++)
    totalflow += supply[node][commodity];
  if ((totalflow > 0.001) || (totalflow < -0.001)) {
    printf("ERROR: Net supply for commodity %d is %lf - not zero\n",
          commodity, totalflow);
    exit(0);
  }                                                                    330
}
/*
 * Output the matrices as input - just see if they're as they should
 * be!
 */
printf("Supply/Demands:\n");
for (node = 0; node < NODES; node++) {
  for (commodity = 0; commodity < NOK; commodity++)
    printf(" %7.2lf", supply[node][commodity]);
  printf("\n");                                                        340
}

printf("Node links:\n");
for (arc = 0; arc < ARCS; arc++) {
  printf(" From %d to %d : Max: %8.2lf Costs:   ",
        headnode[arc], tailnode[arc], ubound[arc]);
  for (commodity = 0; commodity < NOK; commodity++)
    printf(" %8.2lf", cost[arc][commodity]);
  printf("\n");
}                                                                      350

/* Set up starting time */

gettimeofday(&starttime, &tzp);

beginsecs = starttime.tv_sec;
Begintime = starttime.tv_usec / 1000000.0;

printf("Beginning time: %d %d\n", starttime.tv_sec, starttime.tv_usec);

                                                                       360
nettime = 0.0;
buildtime = 0.0;
affinetime = 0.0;
```

```
  /*
   * STEP 1: Optimize the network subproblems without consideration of
   * the joint constraints
   */
  printf("Starting Phase I - Finding feasible subproblem solutions\n");                370

  for (commodity = 0; commodity < NOK; commodity++) { /* For all commodities */
    /* Set up initial costs */
    for (arc = 0; arc < ARCS; arc++) {
      costi[arc] = 0.0;
    }
    /* Set up initial node values */
    for (node = 0; node < NODES; node++)
      supplyi[node] = supply[node][commodity];

                                                                                    380
    Initialize_Net(flowi, duali, costi, bound, supplyi, thread, rthrd,
                   depth, pred, predl, basici, down, link, point,
                   headnode, tailnode);

    /* Optimize the network */
    printf("Ready to make Network # %3d feasible \n", commodity);

#ifdef VERBOSE
    Show_Network(MLNK, VNODES, headnode, tailnode, bound,
                 flowi, costi, basici, down, pred, predl,                           390
                 depth, thread, rthrd, duali);
#endif

    /* Do the "Stage I" optimization, seeking feasibility. */
    nsimplex(MLNK, VNODES, point, link, headnode, tailnode,
             costi, bound, flowi, duali, basici, predl, pred,
             depth, thread, rthrd, down);

    printf("Optimized Network # %3d \nResults:\n", commodity);

                                                                                    400
#ifdef VERBOSE
    Show_Network(MLNK, VNODES, headnode, tailnode, bound,
                 flowi, costi, basici, down, pred, predl,
                 depth, thread, rthrd, duali);
#endif

    printf("Saving network %d\n", commodity);
```

```
  Save_Network(basic, basici, spredl, predl, flow, flowi, mubound,
               bound, sdown, down, spred, pred, sdepth, depth,
               sthread, thread, srthrd, rthrd, dual, duali,
               commodity);

  printf("Saved network %d\n", commodity);

}                       /* For all commodities */

/*
 * Zero out bounds for the feasibility arcs. This prevents them from
 * re-entering the basis. There's a specific test for this in
 * nsimplex()
 */

for (commodity = 0; commodity < NOK; commodity++) {
  for (node = 0; node < NODES; node++) {
    mubound[ARCS + node][commodity] = 0.0;
    cost[ARCS + node][commodity] = 0.0;
    newcost[ARCS + node][commodity] = 0.0;
    altsupply[node][commodity] = 0.0;
  }
}
for (node = 0; node < NODES; node++) {
  ubound[ARCS + node] = 0.0;
  bound[ARCS + node] = 0.0;
}

/*
 * Now, use the REAL costs, and reoptimize the networks, if
 * necessary. If there aren't any costs (i.e. - they're all zero),
 * this step will in fact get skipped.
 */

printf("Starting Phase II - Finding optimal subproblem solutions\n");

set_up_all_duals(dual, headnode, tailnode, sdepth, spredl, spred,
                 sdown, cost, altsupply, basic);

/* Resolve networks */
for (commodity = 0; commodity < NOK; commodity++) {
  optimalnetwork = TRUE;
```

```
    Restore_Network(basic, basici, spredl, predl, flow, flowi,
                    mubound, bound, sdown, down, spred, pred,
                    sdepth, depth, sthread, thread, srthrd, rthrd,
                    headnode, tailnode, point, link, dual, duali,
                    altsupply, commodity);

    for (arc = 0; arc < ARCS; arc++) {
      costi[arc] = cost[arc][commodity];
      if (costi[node] != 0)                                              460
        optimalnetwork = FALSE;
    }

    if (optimalnetwork) {
      printf("Network %d is optimal.  No need to resolve.\n", commodity);
    } else {
      printf("Network %d not at optimality.  Resolving.\n", commodity);

      nsimplex(MLNK, VNODES, point, link, headnode, tailnode, costi,
               bound, flowi, duali, basici, predl, pred, depth,          470
               thread, rthrd, down);

      printf("Optimized Network # %d \nResults:\n", commodity);

#ifdef VERBOSE
      Show_Network(MLNK, VNODES, headnode, tailnode, bound,
                   flowi, costi, basici, down, pred, predl,
                   depth, thread, rthrd, duali);
#endif

                                                                         480
      Save_Network(basic, basici, spredl, predl, flow, flowi,
                   mubound, bound, sdown, down, spred, pred, sdepth,
                   depth, sthread, thread, srthrd, rthrd, dual,
                   duali, commodity);

    }           /* If (optimalnetwork) */
  }                     /* for all commodities */

  printf("Step 0 complete - Found optimal solutions to the individual\n");
  printf("networks.\n");                                                 490
  /* End of Step 0 */

  gettimeofday(&endtime, &tzp);
```

```
  nettime += endtime.tv_sec - starttime.tv_sec -
    starttime.tv_usec / 1000000.0 + endtime.tv_usec / 1000000.0;

  printf("Beginning: Time: %d %d\n", endtime.tv_sec, endtime.tv_usec);

#ifdef DEBUGGING                                                          500
  printf("Solved all of the initial network subproblems\n");
  printf("Now we need to apply the joint capacity constraints\n ");
#endif

  /* Phase III: Main program loop */
  do {
    /* This gets repeated until an optimum is determined */

    /* Display and test the current network solution. If it's */
    /*                                                                    510
     * feasible, then it is necessarily optimal, and we may
     * terminate
     */

    totalcost = 0.0;

    printf("Interim solution at iteration %d\n", iterations);
    printf("Arc: Head Tail Bound          Flow: ");
    for (commodity = 0; commodity < NOK; commodity++)
      printf("%7d", commodity);                                            520
    printf("       Slack\n");

    optimal = TRUE;

    for (arc = 0; arc < ARCS; arc++) {
      totalflow = 0.0;
      printf("%4d %4d %4d %7.2lf                    ", arc, headnode[arc],
            tailnode[arc], ubound[arc]);
      for (commodity = 0; commodity < NOK; commodity++) {
        printf("%6.2lf ", flow[arc][commodity]);                           530
        totalflow += flow[arc][commodity];
        totalcost += flow[arc][commodity] * cost[arc][commodity];
      }
      printf(" %6.2lf\n", ubound[arc] - totalflow);
      if (totalflow > ubound[arc])
        optimal = FALSE;
    }
```

```
    printf("Total cost: %lf (Pure network simplex solution)\n", totalcost);

    iterations++;                                                          540

    /* If the initial solution was optimal, then take benefit. */
    if ((iterations == 1) && optimal) {
      for (commodity = 0 ; commodity < NOK ; commodity ++) {
        for (arc = 0 ; arc < ARCS ; arc ++) {
          allflow[arc][commodity] = flow[arc][commodity];
        }
      }
      break;
    }                                                                      550

    /*
     * STEP 1:    Construct the reduced coupling problem,
     * using the
     */
    /* flow and basis information generated in the subproblems. */
    /*
     * \min \sum_{i}{d^{'}_{i} x_{i2}} : \; {i}{ M_{i} x_{i2} } =
     * b^{'}_{0},              \; x_{i2} \geq 0
     */                                                                    560
    /*
     * Compute A = D_{n} - D_{b} "A_{b}^{-1}" "A_{n}"
     */
    /*
     * Compute c = c_{n} - c_{b} "A_{b}^{-1}" "A_{n}"
     */

    /* Compute b = d - D_{b} x_{b} */

#ifdef DEBUGGING                                                           570
    printf("Constructing Reduced problem\n");
#endif

    gettimeofday(&starttime, &tzp);

    /*
     * Note that we use the ORIGINAL bounds for the reduced
     * problem...
     */
    buildreducedproblem(b, d, c, basic, headnode, tailnode, cost,          580
```

```
                    flow, ubound, spredl, spred, sdepth, sthread,
                    srthrd, sdown, rbounds, x, altflow);

    gettimeofday(&endtime, &tzp);

    buildtime += endtime.tv_sec - starttime.tv_sec -
      starttime.tv_usec / 1000000.0 +
        endtime.tv_usec / 1000000.0;

    printf("Reduced problem Constructed\n");                                    590

#ifdef DEBUGGING
    printf("Looks like:\n");
    printf("D matrix:\n");
    for (arc = 0; arc < ARCS; arc++) {
      for (j = 0; j < NOK * (RVARS); j++)
        printf("%4.1lf ", d[arc][j]);
      printf("\n");
    }
                                                                                600
    printf("\nb - RHS vector\n");
    for (arc = 0; arc < ARCS; arc++) {
      if (arc % 5 == 0)
        printf("\n");
      printf("%lf ", b[arc]);
    }

    printf("\n\nCost Vector\n");
    for (i = 0; i < NOK * (RVARS); i++) {
      if (i % 5 == 0)                                                           610
        printf("\n");
      printf("%lf ", c[i]);
    }

    printf("\n\nBound vector\n");
    for (i = 0; i < NOK * RVARS + 1; i++) {
      if (i % 5 == 0)
        printf("\n");
      printf("%lf ", rbounds[i]);
    }                                                                           620
#endif

    printf("\n\nSolve Reduced problem\n");
```

```
gettimeofday(&starttime, &tzp);

lpcode(d, b, c, x, rduals, &istate, &iters, rbounds, redcosts,
        VARS, CONSTRAINTS, sliding);

affineiterations += iters;                                          630

gettimeofday(&endtime, &tzp);

affinetime += endtime.tv_sec - starttime.tv_sec -
  starttime.tv_usec / 1000000.0 +
    endtime.tv_usec / 1000000.0;

printf("Solved Reduced problem\n");

Round_Answer(x, redcosts, MAXVARS, rduals, ARCS);                   640

/* This determines x_{n} */
/*
 * Compute estimate of x_{b} based on x_{n} x_{b}^{'} =
 * x_{b} - "A_{b}^{-1}" "A_{n}" x_{n}
 */

Adjust_Network(basic, flow, x, ubound, mubound, cost, newcost, rduals,
                altflow, headnode, tailnode, sdepth, spred,
                spredl, sdown, rbounds, redcosts, altsupply);       650

/* Report prevalence of each price: */
reportprevalences();

printf("Reset dual values in networks\n");

set_up_all_duals(dual, headnode, tailnode, sdepth, spredl, spred,
                  sdown, newcost, altsupply, basic);

printf("Now, resolve the networks\n");                              660

gettimeofday(&starttime, &tzp);

/* Resolve networks */
for (optimal = TRUE, commodity = 0; commodity < NOK; commodity++) {
  printf("Restoring Network #%3d\n", commodity);
```

```
    optimalnetwork = TRUE;

    Restore_Network(basic, basici, spredl, predl, flow, flowi,
                    mubound, bound, sdown, down, spred, pred,
                    sdepth, depth, sthread, thread, srthrd, rthrd,
                    headnode, tailnode, point, link, dual, duali,
                    altsupply, commodity);

    for (node = 0; node < MLNK; node++) {
      costi[node] = newcost[node][commodity];
      if (altflow[node][commodity] < - NONZERO)
        optimalnetwork = FALSE;
    }

#ifdef DEBUGGING
    Show_Network(MLNK, VNODES, headnode, tailnode, bound,
                 flowi, costi, basici, down, pred, predl,
                 depth, thread, rthrd, duali);
#endif
    if (optimalnetwork) {
      printf("Network %d is optimal.  No need to resolve.\n", commodity);
    } else {
      printf("Network %d not at optimality.  Resolving.\n", commodity);

      optimal = FALSE;        /* The solution is
                               * proven here to be
                               * suboptimal */

      nsimplex(MLNK, VNODES, point, link, headnode, tailnode, costi,
               bound, flowi, duali, basici, predl, pred, depth,
               thread, rthrd, down);

      printf("Optimized Network # %d \nResults:\n", commodity);

#ifdef VERBOSE
      Show_Network(MLNK, VNODES, headnode, tailnode, bound,
                   flowi, costi, basici, down, pred, predl,
                   depth, thread, rthrd, duali);
#endif

      Save_Network(basic, basici, spredl, predl, flow, flowi,
                   mubound, bound, sdown, down, spred, pred, sdepth,
                   depth, sthread, thread, srthrd, rthrd, dual,
```

```
                    duali, commodity);                                          710

    } /* If (optimalnetwork) */
  }             /* for all commodities */

  gettimeofday(&endtime, &tzp);

  nettime += endtime.tv_sec - starttime.tv_sec -
    starttime.tv_usec / 1000000.0 +
      endtime.tv_usec / 1000000.0;

                                                                              720
}
while (!optimal);

gettimeofday(&endtime, &tzp);

totaltime = -Begintime - beginsecs + endtime.tv_sec +
  endtime.tv_usec / 1000000.0;

/* Determined an optimal solution! */
printf("Determined the optimal solution\n");                                  730

/* Print header */
printf("Arc: Head Tail Bound   Flow: ");
for (commodity = 0; commodity < NOK; commodity++)
  printf("%7d", commodity);
printf("       Slack\n");
totalcost = 0.0;
for (arc = 0; arc < ARCS; arc++) {
  totalflow = 0.0;
  printf("%4d %4d %4d %7.2lf            ", arc, headnode[arc],              740
        tailnode[arc], ubound[arc]);
  for (commodity = 0; commodity < NOK; commodity++) {
    printf("%6.2lf ", altflow[arc][commodity]);
    totalflow += altflow[arc][commodity];
    totalcost += altflow[arc][commodity] * cost[arc][commodity];
  }
  printf(" %6.2lf\n", ubound[arc] - totalflow);
}
printf("Total cost: %lf (Optimum)\n", totalcost);

                                                                              750
printf("\nTime report:\n");
othertime = totaltime - (nettime + affinetime + buildtime);
```

```
  printf("Breakdown: Networks     Affine Scaling   Building Red.   Other \n");

  printf("%%          %6.3lf            %6.3lf           %6.3lf      %6.3lf\n",
         100.0 * nettime / totaltime,
         100.0 * affinetime / totaltime,
         100.0 * buildtime / totaltime,
         100.0 * othertime / totaltime);                                          760

  printf("Seconds     %6.3lf            %6.3lf           %6.3lf      %6.3lf\n",
         nettime,
         affinetime,
         buildtime,
         othertime);

  printf("Elapsed time: %lf\n", totaltime);
  printf("Size of problem:\n");
  printf("Nodes: %d  Arcs: %d    Commodities: %d\n",                            770
         NODES, ARCS, NOK);
  printf("Size of Affine Scaling Problem:\n");
  printf("  Variables: %d  Constraints: %d\n",
         VARS, CONSTRAINTS);
  printf("Number of partitioning iterations: %d\n", iterations);
  printf("Number of affine scaling iterations: %d\n", affineiterations);
  printf("Number of affine scaling iterations in which \nthe problem is infeasible : %d\n", infeasibleiters);
  printf("Optimal cost: %lf\n", totalcost);
  printf("Options chosen:\n");
  printf("Sliding: ");                                                            780
  if (sliding)
    printf("ON\n");
  else
    printf("OFF\n");
  printf("Initialization: ");
  if (midpoint)
    printf("MIDPOINT\n");
  else
    printf("NORMAL\n");

                                                                                  790
  /* Free up the dynamically allocated memory */
  dynamic(FREE);
  exit(0);
}
```

E.4 BUILDRED.C

The function `buildreducedproblem()` is used to produce the reduced problem. Since the joint constraint matrix H takes on the simple form $[II \ldots I]$, the internal representation of the reduced matrix H_n is particularly simple. Much of the work in modifying the software to use more complex joint constraints would lie in modifying this function.

```
/* $Id: buildred.c,v 8.0 93/01/20 18:40:23 cbbrowne Exp $ */
#include "boolean.h"
#include "mcnf.h"
/*****************************************************************
**                        buildred.c                             **
**                   By Christopher Browne                       **
**                   University of Ottawa                        **
**          Prepared for Master of Systems Science Thesis        **
*****************************************************************
** This function takes the solutions to the MCNF network        **
** subproblems, and constructs a "reduced problem."  This       **
** problem takes the joint capacity constraints, and reduces    **
** the number of variables by removing those variables that     **
** were basic in the subproblems.                               **
*****************************************************************/
int
        *db,            /* These describe the matrices D_b and D_n. */
/* D_b[i,db[i]] =1, and D_b[i,*] = 0.0 */
        *dn;            /* dn[] is used in the same way */

extern double   mu;
extern double   slideM;
extern double   artM;

extern boolean  sliding;
extern boolean  midpoint;
extern boolean  randompoint;

void
buildreducedproblem(
                double *b,
                double **d,
                double *c,
                boolean ** basic,
                int *headnode,
```

```
                int *tailnode,
                double **cost,
                double **flow,
                double *bound,
                int **spredl,                                                    40
                int **spred,
                int **sdepth,
                int **sthread,
                int **srthrd,
                boolean ** sdown,
                double *rbounds,
                double *x,
                double **oldflow
)
{                                                                                50

#include "getabian.h"

        int
                node, arc, i, j, var, commodity;        /* Assortment of loop
                                                         * counters */

        int
                bas, nonbas;    /* Counters used to partition basic
                                 * and nonbasic */                              60

        static
        double
                sum, lo, hi;

        static
        double
                cost_adjustment;        /* This is the amount by
                                         * which the TOTAL cost gets
                                         * adjusted when a variable
                                         * was at the bounds */                 70

        /*
         * The structure of the computations of this function is as follows:
         *
         * We need to determine
         *
         * A = D_{n} - D_{b} "A_{b}^{-1}" "A_{n}" c = c_{n} - c_{b} "A_{b}^{-1}"
```

```
 * "A_{n}" b = d - D_{b} x_{b}
 *
 * D = [ I_1 I_2 ... I_k ]
 *
 * b is simply equal to the slacks achieved when the values of the
 * subproblems are introduced into the mutual capacity constraints.
 * Note some b[i] values could be negative.
 */

cost_adjustment = 0.0;

/* Set up the b[] vector */
for (arc = 0; arc < ARCS; arc++)
        b[arc] = bound[arc];

for (commodity = 0; commodity < NOK; commodity++) {
        /*
         * At this point, D refers to a matrix made up of [NOK]
         * identity matrices.
         *
         * Use of a D matrix of more general form is left to later
         * implementation
         */

        /* Determine the partitioning */
        bas = nonbas = 0;
        for (arc = 0; arc < ARCS; arc++) {
                if (basic[arc][commodity]) {        /* Arc is in the basis */
                        db[bas++] = arc;
                } else {
                        dn[nonbas] = arc;
                        rbounds[nonbas + commodity * RVARS] = bound[arc];    /* Insert value into
                                                                              * bound array */
                        /*
                         * Here is where the initial value is put
                         * into the x[] array
                         */

                        hi = bound[arc] - mu;
                        lo = mu;

                        if (hi - lo < mu) {
                                /*
```

```
                        * hi & lo are too near to one
                        * another
                        */
                        printf("BuildReducedProblem(): Arc %d: bound: %lf  mu: %lf  Error: Bounds on flo
                }
                /* Choose value */
                if (midpoint) {
                        x[nonbas + commodity * RVARS] = bound[arc] / (NOK + 1);
                } else {                                                                    130
                        x[nonbas + commodity * RVARS] = mu;
                }

                /*
                 * x[nonbas+commodity*RVARS] =
                 * oldflow[arc][commodity]; This scheme needs
                 * to adjust oldflow[][] if it gives <= 0 or
                 * >= bound
                 */

                                                                                            140
                nonbas++;
        }       /* Endif */
        rbounds[RVARS * NOK + arc] = bound[arc];
}               /* For all ARCS */

/* compute D' */
/* First, fill in D_n matrix with zeros */
for (j = 0; j < RVARS; j++) { /* For each column */
        var = RVARS * commodity + j;
        for (arc = 0; arc < ARCS; arc++)        /* Zero out the column */               150
                d[arc][var] = 0.0;

        /* Fill in the appropriate 1 */
        d[dn[j]][var] = 1.0;

        /* Get a column of A_b^{-1} A_n */
        abinvan(
                headnode[dn[j]],
                tailnode[dn[j]],
                commodity,                                                              160
                sdepth,
                spred,
                spredl,
                sdown,
```

```
                    NODES);
#ifdef DEBUGGING
                printf("abian[%d --> %d]: ", headnode[dn[j]], tailnode[dn[j]]);
                for (node = 0; node < NODES; node++) {
                        printf(" %lf ", abian[node]);
                }                                                              170
                printf("\n");
#endif
                /* adjust the cost -> c[j] */
                sum = 0.0;
                for (node = 0; node < NODES; node++)
                        sum += abian[node] * cost[spredl[node][commodity]][commodity];

                /*
                 * Note: No need to test here to see if the arc is
                 * valid, since if it is not, abian[i] will be ZERO              180
                 */
                c[var] = cost[dn[j]][commodity] - sum;

                /* Adjust the column of D */
                for (node = 0; node < NODES; node++) {
                        arc = spredl[node][commodity];
#ifdef DEBUGGING
                        printf("Adjusting arc %d for variable %d by %lf\n",
                                arc, var, abian[node]);
#endif                                                                         190
                        if (arc < ARCS) {       /* Is the arc a valid
                                                 * one? */
                                d[arc][var] -= abian[node];
                        }
                        /*
                         * Note that this was REALLY simple since D
                         * is real simple
                         */
                }
        }               /* For all variables (of reduced subproblem) */        200

        /* Adjust the columns if necessary */
        for (j = 0; j < RVARS; j++) {
                var = RVARS * commodity + j;
                if (flow[dn[j]][commodity] > NONZERO) { /* var. at bound */
                        for (arc = 0; arc < ARCS; arc++) {
                                d[arc][var] = -d[arc][var];
```

```
                        }
                        c[var] = -c[var];
                        cost_adjustment += c[var] * rbounds[var];
                    }
                }
            }                       /* For all commodities */

        printf("Constant by which the solution to the reduced problem is \
adjusted: %lf\n",
                cost_adjustment);
        /* Fill in an identity matrix for the slack variables */
        for (i = NOK * RVARS; i < NOK * RVARS + ARCS; i++) {
                for (j = 0; j < ARCS; j++) {
                        d[j][i] = 0.0;
                }
                d[i - NOK * RVARS][i] = 1.0;
                c[i] = 0.0;
                x[i] = mu;
                if (x[i] >= rbounds[i]) {
                        printf("Error: Slack variable for arc %d has bound=%lf, less than mu = %lf\n",
                                i - NOK * RVARS, bound[i], mu);
                        exit(-1);
                }
        }

        /* Now, remove basic flows from b[] */
        for (commodity = 0; commodity < NOK; commodity++) {
                nonbas = 0;
                for (arc = 0; arc < ARCS; arc++) {      /* For each arc */
                        b[arc] -= flow[arc][commodity];
                }                       /* For each arc */
        }                               /* For each commodity */

        /* Now, insert flow values into the x[] matrix */

        return;
}                                       /* End function buildreducedproblem() */

void
Initialize_build()
{
#ifdef DEBUGGING
        printf("buildred: allocating memory\n");
```

```
#endif
        db = (int *) malloc(NODES * sizeof(int));
        dn = (int *) malloc(RVARS * sizeof(int));
#ifdef DEBUGGING
        printf("buildred: allocated memory\n");
#endif
}

void
Close_build()                                                    260
{
        free(db);
        free(dn);
}
```

E.5 DUALTREE.C

The functions in this file are used to rebuild the dual values for the networks in preparation for re-optimization. The function `set_up_all_duals()` is the only one that is to be used externally. The methodology is as follows:

1. A tree is built to represent the basis of the network.

   The data structures that are used within `nsimplex()` allow travel towards the root of the tree, but not from the root to the leaves. The arrays `basis_tree[VNODES][VNODES][NOK]` and `successors[VNODES][NOK` describe the basis tree in a form that allows easy traversal from the root towards the leaves.

   This takes $O(NODES)$ time, and $O(NODES^2)$ space.

2. Each arc cost is applied to the the appropriate node of the tree.

   The cost for each arc is initially attached just to the root of the subtree that is affected, and a tree traversal is performed to apply all costs to the respective subtrees.

   This takes $O(ARCS)$ time, and $O(NODES)$ space.

3. Search for the root.

   This is simply done by starting (arbitrarily) at the first node, and climbing the tree until a node of depth 1 is reached.

   An alternative heuristic could be faster: Pick the better of the current node and the node following it (i.e. — pick the one that has the least depth).

The current method takes $O(depth(NODES))$ time, and is not worth optimizing further.

4. Lastly, the dual values are adjusted by applying the cost adjustments found in the second step to:

   - The current node; and
   - Recursively, to its successors

   This takes $O(NODES)$ time, and $O(\max_n depth(n))$ space.

The overall complexity is $O(ARCS)$ in time, and $O(NODES^2)$ in space.

```
/* $Id: dualtree.c,v 8.0 93/01/20 18:40:29 cbbrowne Exp $ */
#include "boolean.h"
#include "mcnf.h"
/*************************************************************
** dualtree.c **
** By Christopher Browne **
** University of Ottawa **
** Prepared for Master of Systems Science Thesis **
*************************************************************
** This set of functions builds a basis tree; inserting each **
** basic arc into the tree, linking it to its successors. **
** This information is required when modifying the dual **
** values; we need a list of all successors for a given node. **
** It may also be useful in producing the A_{N}A_{B}^{-1} **
** matrix. **
*************************************************************
** Notable Statistics: **
** Time complexity: **
** for build_tree: O(VNODES) **
** for get_subtree: O(VNODES) **
** Memory consumption: **
** NODES*NODES integers **
** + up to NODES recursive calls within **
** build_partial_subtree **
*************************************************************/

int
        **successors, ***basis_tree;

double
        **dcost;
```

```
void
build_tree(
        int **predl,
        int **pred,
        int **depth,
        int commodity)
{                                                                    40
    static
    int
            node, a, b, arc, tmp;

    /* Initialize to zero - NOTHING is in the tree */
    for (node = 0; node < VNODES; node++)
        successors[node][commodity] = 0;

    for (node = 0; node < VNODES; node++) {
        if (depth[node][commodity] == 0)                             50
            continue;       /* Node has no predecessor */
#ifdef DEBUGGING
        printf("Node: %d  Depth: %d\n", node, depth[node][commodity]);
#endif
        arc = predl[node][commodity];
        a = node;
        b = pred[node][commodity];
        if (depth[a][commodity] > depth[b][commodity]) {
            tmp = a;
            a = b;                                                   60
            b = tmp;
        }
#ifdef DEBUGGING
        printf("buildtree(): inserting %d-> %d\n", a, b);
#endif
        tmp = successors[a][commodity]++;
        basis_tree[a][tmp][commodity] = b;
#ifdef DEBUGGING
        printf("succeeded.\n");
#endif                                                               70
    }
    return;
}

void
```

```
determine_dual_adjustments(
                        int *headnode,
                        int *tailnode,
                        int **depth,
                        double **cost,                                          80
                        int commodity,
                        boolean ** basic)
{
        static int      arc, node;

        for (node = 0; node < VNODES; node++)
                dcost[node][commodity] = 0.0;
        for (arc = 0; arc < MLNK; arc++) {
                if (basic[arc][commodity]) {        /* Insert dual costs only for
                                                     * basic arcs */              90
                        if (depth[headnode[arc]][commodity] > depth[tailnode[arc]][commodity]) {
                                dcost[headnode[arc]][commodity] -= cost[arc][commodity];
                        } else {
                                dcost[tailnode[arc]][commodity] += cost[arc][commodity];
                        }
                }
        }
#ifdef DEBUGGING
        printf("\nDual adjustments to distribute over subproblem %d\n",
                commodity);                                                     100
        for (node = 0; node < VNODES; node++) {
                printf("Node: %d  Adjustment: %lf \n", node, dcost[node][commodity]);
        }
#endif
        return;
}

int
find_root(int **depth,
          int **pred,                                                           110
          int commodity)
{
        static int      root = 0;

        while (depth[root][commodity] != 0)
                root = pred[root][commodity];

        return (root);
```

```
}

                                                                        120
int
adjust_duals_of_subtree(int node,
                        int commodity,
                        double delta,
                        double **dual)
{
        int             i, s;       /* Can NOT be static! This function
                                     * recurses! */
#ifdef DEBUGGING
        printf("Adjusting tree rooted at %d by %lf (%d successors)\n", node,   130
                delta + dcost[node][commodity], successors[node][commodity]);
#endif
        s = successors[node][commodity];
        for (i = 0; i < s; i++)
                adjust_duals_of_subtree(basis_tree[node][i][commodity], commodity,
                                        delta + dcost[node][commodity], dual);

        dual[node][commodity] += delta + dcost[node][commodity];
}

                                                                        140
void
set_up_all_duals(double **dual,
                 int *headnode,
                 int *tailnode,
                 int **depth,
                 int **predl,
                 int **pred,
                 boolean ** down,
                 double **cost,
                 double **altsupply,                                    150
                 boolean ** basic)
{
        static int      root, commodity, node;
        extern void     Adjust_Arc_Names(
                                                double **altsupply,
                                                boolean ** basic,
                                                int *headnode,
                                                int *tailnode,
                                                int commodity);

                                                                        160
        for (commodity = 0; commodity < NOK; commodity++) {
```

```
                Adjust_Arc_Names(altsupply, basic, headnode, tailnode, commodity);
                for (node = 0; node < VNODES; node++)
                        dual[node][commodity] = 0.0;
                build_tree(pred, pred, depth, commodity);
#ifdef DEBUGGING
                printf("Tree built.\n");
#endif
                determine_dual_adjustments(headnode, tailnode, depth, cost,
                                                commodity, basic);                    170
#ifdef DEBUGGING
                printf("Adjustments allocated to nodes.\n");
#endif
                root = find_root(depth, pred, commodity);
#ifdef DEBUGGING
                printf("Root: %d \n", root);
#endif
                adjust_duals_of_subtree(root, commodity, 0.0, dual);
        }
}                                                                                     180

void
Initialize_dualtree()
{
        int             i, j;

        successors = (int **) malloc(VNODES * sizeof(int *));
        for (i = 0; i < VNODES; i++)
                successors[i] = (int *) malloc(NOK * sizeof(int));
                                                                                      190
        dcost = (double **) malloc(VNODES * sizeof(double *));
        for (i = 0; i < VNODES; i++)
                dcost[i] = (double *) malloc(NOK * sizeof(double));

        basis_tree = (int ***) malloc(VNODES * sizeof(int **));
        for (i = 0; i < VNODES; i++) {
                basis_tree[i] = (int **) malloc(VNODES * sizeof(int *));
                for (j = 0; j < VNODES; j++)
                        basis_tree[i][j] = (int *) malloc(NOK * sizeof(int));
        }                                                                             200
}

void
Close_dualtree()
```

```
{
        int             i, j;

        for (i = 0; i < VNODES; i++)
                free(successors[i]);
        free(successors);                                                    210

        for (i = 0; i < VNODES; i++)
                free(dcost[i]);
        free(dcost);

        for (i = 0; i < VNODES; i++) {
                for (j = 0; j < VNODES; j++)
                        free(basis_tree[i][j]);
                free(basis_tree[i]);
        }                                                                    220
        free(basis_tree);
}
```

E.6 GETABIAN.C

The function `abinvan()` computes a column of the matrix $G_b^{-1}G_n$, placing the result in the array `abian[NODES]`. These columns, that with the arc passed in to the function describe the cycle involving the basic arcs and the parameter, are used in the production of the reduced problem, and in interpreting the results. The approach is to choose a `(headnode, tailnode)` combination, and then climb the basis tree in order to determine the set of basic arcs involved in the cycle between the two nodes.[2]

```
/* $Id: getabian.c,v 8.0 93/01/20 18:40:32 cbbrowne Exp $ */
#include "boolean.h"
/*************************************************************************
** abinvan.c **
** By Christopher Browne **
** University of Ottawa **
** Prepared for Master of Systems Science Thesis **
*************************************************************************
** This function takes two nodes (defining an arc) as input, **
** and computes the corresponding column of F_{b}^{-1} F_{n}. **          10
*************************************************************************/
```

[2]The `(headnode, tailnode)` combination normally is used in this context to describe a nonbasic arc, but the algorithm will produce an appropriate cycle array for basic arcs and for non-existent arcs as well.

```
double          *abian;         /* The vector that contains the computed
                                 * column */

void
abinvan(
        int head,
        int tail,
        int commodity,                                                    20
        int **depth,
        int **pred,
        int **predl,
        boolean ** down,
        int NODES
)
{
        int             left, right, sign;
        int             i;

                                                                          30
        for (i = 0; i < NODES; i++)
                abian[i] = 0;

        /*
         * Determine which node is at the greatest depth, and put it on the
         * left side
         */
        if (depth[head][commodity] > depth[tail][commodity]) {
                sign = -1;
                left = head;                                              40
                right = tail;
        } else {
                sign = 1;
                right = head;
                left = tail;
        }

        /* Climb up the left branch until the two sides are at equal depths */
        while (depth[left][commodity] > depth[right][commodity]) {
                if (down[predl[left][commodity]][commodity]) {            50
                        abian[left] = sign;
                } else {
                        abian[left] = -sign;
                }
                left = pred[left][commodity];
```

```
        }

        /* Now, climb up both branches simultaneously until they coincide */
        while (left != right) {
                /* Climb the left branch */
                if (down[predl[left][commodity]][commodity]) {
                        abian[left] = sign;
                } else {
                        abian[left] = -sign;
                }
                left = pred[left][commodity];

                /* Climb the right branch */
                if (down[predl[right][commodity]][commodity]) {
                        abian[right] = -sign;
                } else {
                        abian[right] = sign;
                }
                right = pred[right][commodity];
        }
        return;
}

void
Initialize_abian(int NODES)
{
#ifdef DEBUGGING
        printf("getabian: Allocating space\n");
#endif
        abian = (double *) malloc(NODES * sizeof(double));
#ifdef DEBUGGING
        printf("getabian: Allocated space\n");
#endif
}

void
Close_abian()
{
        free(abian);
}
```

E.7 ADJUSTNT.C

The function `Adjust_Network()` performs adjustments on the parameters of the individual subproblems, based on the values of the estimated flows x_n determined in the reduced problem. It refers to the function `newprice()` (described in E.8) for new prices for the network subproblems.

```
/* $Id: adjustnt.c,v 8.0 93/01/20 18:40:22 cbbrowne Exp Locker: cbbrowne $ */
#include "mcnf.h"
#include "boolean.h"
#include "decision.h"
/*****************************************************************
**                        adjustnt.c                            **
**                   By Christopher Browne                      **
**                   University of Ottawa                       **
**          Prepared for Master of Systems Science Thesis       **
*****************************************************************                    10
** The function Adjust_Network() produces modified prices (and  **
** possibly adjusts flows) for the network subproblems, as      **
** products of the solution of the Reduced problem, x_n (and    **
** possibly the dual values/reduced prices associated with that **
** solution)                                                    **
*****************************************************************/

extern double   netM;

void                                                                   20
Adjust_Network(
               boolean ** basic,
               double **flow,
               double *x,
               double *bound,
               double **mbound,
               double **cost,
               double **newcost,
               double *rduals,
               double **altflow,                                       30
               int *headnode,
               int *tailnode,
               int **sdepth,
               int **spred,
               int **spredl,
               boolean ** sdown,
```

```
        double *rbounds,
        double *redcosts,
        double **newsupply)
{                                                                    40
    static
    int
            commodity, arc, node, nonbasics, method;

    static
    double
            flowe, rcoste, deltaflow, totalflow, totalcost;

#include "getabian.h"

                                                                     50
    /* Zero/infinity out the costs */
    for (commodity = 0; commodity < NOK; commodity++) {
        for (arc = 0; arc < ARCS + NODES; arc++) {
            newcost[arc][commodity] = 0.0;
        }
    }

    /* Now, based on x[], determine the "alternative flows". */
    for (commodity = 0; commodity < NOK; commodity++) {
        nonbasics = 0;
        /*                                                           60
         * Adjust x_{n} for the nonbasic arcs that were at the upper
         * bounds
         */
        for (arc = 0; arc < ARCS; arc++) {
            if (basic[arc][commodity]) {
                altflow[arc][commodity] = flow[arc][commodity];
            } else {
                altflow[arc][commodity] = x[nonbasics + RVARS * commodity];

                                                                     70
                /*
                 * This IF statement may not be necessary -
                 * to test!
                 */
                if (flow[arc][commodity] > NONZERO)
                    altflow[arc][commodity] = bound[arc] - altflow[arc][commodity];


                nonbasics++;
```

```
                }                                                              80
        }               /* For all arcs */

        /* Adjust the flows through basic arcs by the x_n values */
        for (arc = 0; arc < ARCS; arc++) {
                if (!basic[arc][commodity]) {
                        abinvan(headnode[arc], tailnode[arc], commodity,
                                sdepth, spred, spredl, sdown, NODES);
                        /* Compute alternative flows in the network */
                        for (node = 0; node < NODES; node++) {
                                if (spredl[node][commodity] < ARCS) {          90
#ifdef DEBUGGING
                                        printf("Adjust node %d by %lf\n", spredl[node][commodity],
                                                abian[node] * (flow[arc][commodity] -
                                          altflow[arc][commodity]));
#endif
                                        altflow[spredl[node][commodity]][commodity] += abian[node] *
                                                (flow[arc][commodity] - altflow[arc][commodity]);
                                }       /* if */
                        }       /* for node */
                }       /* if */                                               100
        }               /* for arc */
}                       /* For commodity */

/* Display the current network solution. */
totalcost = 0.0;
printf("MCNF solution as adjusted by the reduced problem:\n");
printf("Arc: Head Tail Bound     Flow: ");
for (commodity = 0; commodity < NOK; commodity++)
        printf("%7d", commodity);
printf("      Slack\n");                                                       110

for (arc = 0; arc < ARCS; arc++) {
        totalflow = 0.0;
        printf("%4d %4d %4d %7.2lf                ", arc, headnode[arc],
                tailnode[arc], bound[arc]);
        for (commodity = 0; commodity < NOK; commodity++) {
                printf("%6.2lf ", altflow[arc][commodity]);
                totalflow += altflow[arc][commodity];
                totalcost += altflow[arc][commodity] * cost[arc][commodity];
        }                                                                      120
        printf(" %6.2lf\n", bound[arc] - totalflow);
```

```
        }
        printf("Total cost: %lf (Network adjusted by reduced problem)\n",
                totalcost);


        /*
         * Original: if f_i is basic, and x'_i < 0, then c_i = INFINITY if
         * f_i is basic, and x'_i > bound, then c_i = -INFINITY c_i = 0                    130
         * otherwise also, if altflow[i] != 0, set the bound to 0.
         */

        printf("From this, determine which members of x_b would be negative,\n");
        printf("and ought to be eliminated from the basis set.\n ");
        for (commodity = 0; commodity < NOK; commodity++) {
                printf("Alternative x_b flows:\n");
                printf("Determine new cost function for network subproblems\n");
                printf("NODES = %d\n", NODES);
                for (arc = 0; arc < ARCS; arc++) {                                      140
                        printf("Arc: %3d  ", arc);
                        printf("NetS. flow: %5.2lf ",
                                flow[arc][commodity]);
                        printf("Alt.flow: %5.2lf  ",
                                altflow[arc][commodity]);

                        newcost[arc][commodity] = newprice(basic[arc][commodity],
                                                        flow[arc][commodity],
                                                    altflow[arc][commodity],
                                                        bound[arc]);                    150
                }
        }                       /* for commodity */

        printf("Adjustments made: Returning\n");
        return;
}
```

E.8 DECISION.C

The function `newprice()` determines the revised costs that are used to resolve the network subproblems.

```
/* $Id: decision.c,v 8.0 93/01/20 18:40:25 cbbrowne Exp Locker: cbbrowne $ */
#include "boolean.h"
#include "mcnf.h"
```

```
/*****************************************************************
**                        decision.c                           **
**                    By Christopher Browne                    **
**                    University of Ottawa                     **
**          Prepared for Master of Systems Science Thesis      **
*****************************************************************
** This function determines the new prices for the network     **
** subproblems, based on a decision table.                     **
*****************************************************************/

extern double   netM;

#define T 1
#define F 0

#define ZERO 0
#define IN 1
#define UP 2
#define LTZ 3
#define GTU 4

#define ERROR (-1)

double          price[2][3][5];
int             prevalence[2][3][5];

#ifdef NEVERDEFINED
Key             set:

                BASIC =
{T | F}
T means that the arc was in the basis
F means that the arc was not in the basis

NETVALUE =
{ZERO | IN | UP}
ZERO means that the arc flow in the network problem was 0
IN means that the arc flow in the network problem was in the range
0 < FLOW < Upper bound
UP means that the arc flow in the network problem was at the upper
bound

ALTVALUE =
```

```
{LTZ | ZERO | IN | UP | GTU}
LTZ means that the alternative arc flow estimate is less than 0
ZERO,          IN, and UP have analogous interpretations to those for NETVALUE
               GTU means that the alternative arc flow estimate is greater than the
               upper bound

#endif                          /* NEVERDEFINED */
void
init_Decision_table(int Nodes)
{
  int           tf, net, aff;
  double        Basic_violation_price, NonBasic_price1, M3, NonBasic_price2;

  Basic_violation_price = netM;
  NonBasic_price1 = Basic_violation_price * (Nodes + 1.0);
  NonBasic_price2 = NonBasic_price1 * (Nodes + 1.0);
  M3 = 0.0;                     /* NonBasic_price * (Nodes + 1.0); */

  /*
   * A cost of netM * (Nodes+2) is sufficent to make this cost overpower any
   * path at cost -netM, even if such a path passed through all nodes in the
   * network, which is the longest possible such path
   */

  /* Set up data table */
  /* BASIC     NETVALUE ALTVALUE NEWCOST */

  price[F][IN][ZERO] = ERROR;
  price[F][IN][IN] = ERROR;
  price[F][IN][UP] = ERROR;

  price[F][IN][LTZ] = ERROR;
  price[F][IN][GTU] = ERROR;

  price[F][ZERO][LTZ] = ERROR;
  price[F][UP][LTZ] = ERROR;
  price[F][ZERO][GTU] = ERROR;
  price[F][UP][GTU] = ERROR;

  price[T][ZERO][ZERO] = M3;
  price[T][ZERO][IN] = 0.0;
  price[T][ZERO][UP] = -M3;
  price[T][ZERO][LTZ] = Basic_violation_price;
```

```
price[T][ZERO][GTU] = -Basic_violation_price;                                    90

price[T][IN][ZERO] = M3;
price[T][IN][IN] = 0.0;
price[T][IN][UP] = -M3;
price[T][IN][LTZ] = Basic_violation_price;
price[T][IN][GTU] = -Basic_violation_price;

price[T][UP][ZERO] = M3;
price[T][UP][IN] = 0.0;
price[T][UP][UP] = -M3;                                                          100
price[T][UP][LTZ] = Basic_violation_price;
price[T][UP][GTU] = -Basic_violation_price;

price[F][ZERO][ZERO] = NonBasic_price2;
price[F][ZERO][IN] = 0.0;
price[F][ZERO][UP] = 0.0;       /* - NonBasic_price1; */

price[F][UP][ZERO] = NonBasic_price1;
price[F][UP][IN] = 0.0;
price[F][UP][UP] = 0.0;                  /* - NonBasic_price2; */                110

for (aff = 0; aff < 5; aff++) {
  for (net = 0; net < 3, net++) {
    for (tf = 0; tf < 2; tf++) {
      printf("price");
      if (tf == T) {
        printf("[T]");
      } else if (tf == F) {
        printf("[F]");
      } else {                                                                   120
        printf("ERROR!\n");
        exit(-1);
      }

      if (net == ZERO) {
        printf("[ZERO]");
      } else if (net == IN) {
        printf("[IN]");
      } else if (net == UP) {
        printf("[UP]");                                                          130
      } else {
        printf("ERROR!\n");
```

```
            exit(-1);
          }

          if (aff == ZERO) {
            printf("[ZERO]");
          } else if (aff == LTZ) {
            printf("[LTZ]");
          } else if (aff == GTU) {
            printf("[GTU]");
          } else if (aff == IN) {
            printf("[IN]");
          } else if (aff == UP) {
            printf("[UP]");
          } else {
            printf("ERROR!\n");
            exit(-1);
          }
          printf(" = %lf\n", price[tf][net][aff]);
          prevalence[tf][net][aff] = 0;
        }
      }
    }
    return;
}

double
    newprice(
            boolean basic,
            double netflow,
            double altflow,
            double bound)
{
    static int      basici, neti, alti;

    static double   chosenprice;
    printf("price[");
    if (basic) {
      basici = T;
      printf("T");
    } else {
      basici = F;
      printf("F");
    }
```

```
    printf("][");

    if (netflow < NONZERO) {
      neti = ZERO;                                                    180
      printf("ZERO");
    } else if (netflow > bound - NONZERO) {
      neti = UP;
      printf("UP");
    } else {
      neti = IN;
      printf("IN");
    }
    printf("][");
    if (altflow < 0.0) {                                              190
      alti = LTZ;
      printf("LTZ");
    } else if (altflow < NONZERO) {
      alti = ZERO;
      printf("ZERO");
    } else if (altflow < bound - NONZERO) {
      alti = IN;
      printf("IN");
    } else if (altflow > bound) {
      alti = GTU;                                                     200
      printf("GTU");
    } else {
      alti = UP;
      printf("UP");
    }
    chosenprice = price[basici][neti][alti];
    prevalence[basici][neti][alti]++;
    printf("] = %lf\n", chosenprice);
    if (chosenprice == ERROR) {
      printf("Error! Contradiction in network repricing!\n");         210

      printf("Netflow: %4.1lf      Altflow: %4.1lf\n", netflow, altflow);
      exit(-1);
    } else {
      return (chosenprice);
    }
}
```

```
void
  reportprevalences(void)                                                    220
{
  static int
    aff, net, tf;
#ifdef VERBOSE
  for (aff = 0; aff < 5; aff++) {
    for (net = 0; net < 3; net++) {
      for (tf = 0; tf < 2; tf++) {
        printf("prevalence");
        if (tf == T) {
          printf("[T]");                                                     230
        } else if (tf == F) {
          printf("[F]");
        } else {
          printf("ERROR!\n");
          exit(-1);
        }

        if (net == ZERO) {
          printf("[ZERO]");
        } else if (net == IN) {                                              240
          printf("[IN]");
        } else if (net == UP) {
          printf("[UP]");
        } else {
          printf("ERROR!\n");
          exit(-1);
        }

        if (aff == ZERO) {
          printf("[ZERO]");                                                  250
        } else if (aff == LTZ) {
          printf("[LTZ]");
        } else if (aff == GTU) {
          printf("[GTU]");
        } else if (aff == IN) {
          printf("[IN]");
        } else if (aff == UP) {
          printf("[UP]");
        } else {
          printf("ERROR!\n");                                                260
          exit(-1);
```

```
        }
        printf("of: %lf", price[tf][net][aff]);
        printf(" = %d\n", prevalence[tf][net][aff]);
      }
    }
  }
#endif
}
```

E.9 SAVENETW.C

The function `Save_Network()` copies the values determined in the network simplex code, `nsimplex()` in [E.12], from the singly dimensioned arrays to arrays in which they are saved so that the interim solutions can be modified and passed back to `nsimplex()` using `Restore_Network()` so that they can be resolved.

E.10 SHOWNET.C

The function `Show_Network()` is used to display solutions of the network subproblems. This is mostly useful for debugging purposes.

E.11 INITNETW.C

The function `initializenet()` is used to generate an initial feasible solution to a network problem. This initial solution is created by adding an artificial node, and connecting arcs from each of the "real" nodes to this node, with "Big M" costs. The initial solution is to allow *all* flows from sources to go to the artificial node, and for all flows to sinks to come from the artificial node. If the problem is feasible, the high costs applied to the artificial arcs will force their flows to zero.

E.12 NSIMPLEX.C

The function `nsimplex()`, based on Shan's `submk.f` [61], solves the single commodity network subproblems using the network simplex method.

E.13 Affine Scaling Code

E.13.1 Overall Structure

The affine scaling functions have the following overall structure:

- `affine.h`

  This file contains various parameters that are used throughout the affine scaling algorithms.

- `lpcode.c`

  This contains the function, `lpcode()` that drives the solution of the linear program.

- `firsts.c`

  The function `InitialSolution()` is used to introduce an interior point for the initial solution. It adds an artificial variable to the LP model such that the initial point $1/2 \times e^T$ is a feasible solution. An extremely high cost is attached to the artificial variable so that it will be "driven out" of the solution as quickly as possible.

  Some experimentation has been done with alternative methods for introducing initial solutions, specifically:

  - Random generation of an initial point, as described in Appendix [6.1.2]
  - Generation of a point using a "sliding barrier method" similar to that of Meyer and Schultz [60]. This method requires a number of modifications of the Interior Point method:

    1. The constraints that slide *must* be of the form $Ax \leq b$; that is, there must be slack variables. In the MCNF reduced problem, this requirement is satisfied by all of the constraints. More general Linear Programs are likely to have equality constraints.
       If there are strict equality constraints, they may be dealt with using the "standard" Big M method with the addition of an artificial variable. This correspondingly complicates the code, since there are two different forms of feasibility being sought simultaneously.
    2. The right hand side, b, is modified, and may be remodified in each iteration until feasibility is reached. This required the introduction of an array, `borig[]` in which the original b is stored. A new function, `ChangeDelta`, is used to perform the modifications to both b and the objective function, c.

  It would be possible to use a combination of the initialization schemes, choosing a random initial point within the "sliding barrier" method.

- `rprice.c`

  The function `OptimalDirection()` is used to determine the optimal direction of travel, c_p, stored in the vector `r[]`. It uses quadratic decomposition methods developed for solving least-squares problems.

  - `dqrdc.c`

    The function `QRFactorize()` computes a QR decomposition using Householder transformations. It uses the sub-function `EuclideanNormal()`.

  - `dqrsl.c`

    The function `SolveLeastSquares()` takes the matrices determined using `QRFactorize()`, and determines the minimizing value x^k. For our problem, this value becomes the optimal direction, c_p that reduces the value of the objective function, while remaining feasible for all of the constraints.

- `chekup.c`

  The function `StoppingRule()` determines whether or not the current solution x^k is epsilon-optimal. That is, it provides one of three answers:

  - The solution is not optimal, and more work must be done.
  - The solution is optimal. Terminate.
  - An optimal solution cannot be found. Terminate.

- `updatx.c`

  The function `UpdateX()` updates the solution x^k, by adding an appropriate multiple of c_p^k. It attempts to eliminate the artificial variable that was added to ensure feasibility.

E.14 MNETGEN

This computer program, written in the Perl language, produces random feasible networks that were used for testing purposes, in the form described in E.1. The algorithm used is described in Section 6.1.1.

```
eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}' # -*-Perl-*-
    if 0;

# $Id: mnetgen,v 7.0 93/01/04 23:09:04 cbbrowne Exp Locker: cbbrowne $
# By: Christopher Browne
```

```
# Date: June 8, 1992
# This program generates a random multicommodity network
# Parameters:
# Coming from <stdin>
# (1) COMMODITIES - number of commodities
# (2) NODES - number of nodes
# (3) ROUTES - number of sources/sinks per commodity
# (4) EXTRAS - number of additional links
# added "just for fun"
# (6) LO, HI - minimum, max. price for arcs
# (8) MINQ, MAXQ - minimum, maximum quantity of flows
# (9) AVG - Average path length (exponential dist'n)
#
# Output:
# (I) NETWORK DATA:
# 1 line for each "arc" (each of which involves NOK commodities)
# Each line contains NOK+3 numbers:
# A <— Node at the head
# B <— Node at the tail
# Costs 1 .. NOK <— Costs for each commodity
# Joint Capacity <— Aggregate capacity of the arc
#
# (II) SOURCE FILE: NOK column matrix indicating supplies/demands for each
# commodity
# -> For each node
# Positive -> Supply node
# 0 -> No net supply/demand
# Negative -> Demand node

# First: Get the parameters
if ($#ARGV < 8) {
    print "Insufficient parameters\n";
    print "Needed: COMMODITIES  NODES  ROUTES  AVG EXTRAS\n";
    print "COSTLO COSTHI  MINQ MAXQ\n";
    die(-1);
}
$COMMODITIES =$ARGV[0];
$NODES       =$ARGV[1];
$ROUTES      =$ARGV[2];
$AVG         =$ARGV[3];
$EXTRAS      =$ARGV[4];
$COSTLO      =$ARGV[5];
$COSTHI      =$ARGV[6];
```

```
$MINQ       =$ARGV[7];
$MAXQ       =$ARGV[8];                                                    50

# Second: Set up paths
foreach $COMM (1 .. $COMMODITIES) {

    foreach $p (1..$NODES) {
        $supply[&key($p,$COMM)]=0; # Zero out the supply matrix
    }

    foreach $p (1..$ROUTES){
        &randomizepath;                                                  60
        $flow=$MINQ + int(0.5 + rand($MAXQ-$MINQ));
        # We'll use an exponential distribution for the route length...
        $length = int(1 - $AVG * 2.3025851 * log(1-rand(1)));
        # 2.3025851 is a "magic number", specifically equal to ln(10)
        if ($length > $NODES) {  # Max. length is $NODES...
            $length = $NODES;
        }
        $supply[&key($path[1],$COMM)] += $flow;
        $supply[&key($path[$length],$COMM)] -= $flow;
        foreach $n (1..$length) {                                        70
            &link($path[$n],$path[$n+1],$flow);
        }                           # foreach $n
    }                               # foreach $p
}                                   # foreach $COMM

# Third: Add some random arcs
foreach $k (1..$EXTRAS) {
    $head=0;
    $tail=0;
    while ($head==$tail) {                                               80
        $head=int(1+rand($NODES));
        $tail=int(1+rand($NODES));
    }
# Link in this node
    $size= $MINQ+int(1+rand($MAXQ-$MINQ));
    &link($head, $tail, $size);
}

# Fourth: See if there are any nodes that aren't linked to anything
foreach $head (1..$NODES) {                                              90
  NODE: {
```

```
        foreach $tail (1..$NODES) {
            last NODE if ($arc{&key($head,$tail)} > 0);
        }
        # If it gets here, then the node wasn't linked to anything.
        # This code forces there to be at least one arc going out of
        # each node.

        $tail=$head;                    # Initialize the Tail so that there's
                                        # at least one iteration of the while
                                        # loop

        while ($tail==$head) {
            $tail=int(1+rand($NODES));
        }
        $size = $MINQ + int(0.5+rand($MAXQ-$MINQ));
        &link($head, $tail, $size);
    }
}

# Now, everything is linked. Count the arcs.
$ARCS = 0;

foreach $head (1..$NODES) {
    foreach $tail (1..$NODES) {
        if ($arc{&key($head,$tail)} > 0) {
            $ARCS ++ ;
        }
    }
}

# Print out the size of the problem
print "$ARCS\n$NODES\n$COMMODITIES\n";

foreach $head (1..$NODES) {
  foreach $tail (1..$NODES) {
    if ($arc{&key($head,$tail)} > 0) {
      print " ", $head-1, " ", $tail-1 , " "; # Print head, tail
      # Note that 1 is subtracted - This program starts node
      # numbering at 1, whereas the other software wants the
      # numbering to start at 0.
      foreach $n (1..$COMMODITIES) {
          print int(1+rand($COSTHI-$COSTLO)+$COSTLO)," ";
      }
```

```
      print $arc{&key($head,$tail)}, "\n";
    }
  }
}

# Now, supply/demand                                                    140
foreach $node (1..$NODES) {
  foreach $comm (1..$COMMODITIES) {
    print " $supply[&key($node,$comm)]";
  }
  print "\n";
}

# Display the parameters as input
print "Nodes: $NODES\n";
print "Commodities: $COMMODITIES\n";                                     150
print "Number of routes: $ROUTES\n";
print "Number of additional arcs: $EXTRAS\n";
print "Cost Range: $COSTLO $COSTHI\n";
print "Supply/Demand Range: $MINQ: $MAXQ\n";
print "Average path length: $AVG\n";

exit(0);

sub randomizepath {
# This creates a random permutation of the nodes so that up to $NODES of  160
# them can be chosen without replacement
   foreach $kk (1..$NODES) {
     $path[$kk]=$kk;
   }
   foreach $kk (1..$NODES) {
     $swap = int(1+rand($NODES));
     $tmp = $path[$kk];
     $path[$kk]=$path[$swap];
     $path[$swap]=$tmp;
   }                                                                    170
# print "Path: ";
   foreach $kk (1..$NODES) {
# print $path[$kk]," ";
   }
# print "\n";
 }
```

```
sub key {
# This converts the (head,tail) pair into ONE value
     return($_[0]*($NODES+$COMMODITIES)+$_[1]);                                    180
}

sub link {
   # connect $_[0] to $_[1], with a capacity of $_[2]
   $arc{&key($_[0],$_[1])} += $_[2];
}
```

E.15 MCNFAFF

This computer program, written in the Perl language, translates the networks produced by `MNETGEN` [E.14] into the matrix form

$$A \mid b$$

$$c$$

$$u$$

The computer program, `SOLVEAFF`, (see [E.2]) is then used to solve this problem using an affine scaling algorithm.

Bibliography

[1] Ilan Adler, Mauricio G. C. Resende, and Geraldo Veiga. An Implementation of Karmarkar's Algorithm for Linear Programming. March 1986.

[2] Ravindra K. Ahuja, Andrew V. Goldberg, James B. Orlin, and Robert E. Tarjan. Finding minimum cost flows by double scaling. *Mathematical Programming*, 53(3), February 1992.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network Flows. Technical Report 2059-88, Alfred. P. Sloan School of Management, M.I.T., August 1988.

[4] Agha Iqbal Ali and Jeff L. Kennington. A Primal Partitioning Code for Solving Multicommodity Network Flow Problems (Version 1). Technical Report IEOR 77003, Southern Methodist University, August 1977.

[5] I. Ali, D. Barnett, K. Farhangian, J. Kennington, B. McCarl, B. Patty, B. Shetty, and P. Wong. Multicommodity Network Problems: Applications and Computations. Technical Report 82-OR-4, Southern Methodist University, September 1983.

[6] Earl R. Barnes. A variation on karmarkar's algorithm for solving linear programming problems. *Mathematical Programming*, 36, 1986.

[7] Richard S. Barr, Keyvan Farhangian, and Jeff L. Kennington. Networks with Side Constraints: An LU Factorization Update. Technical Report 83-OR-4, Southern Methodist University, June 1983.

[8] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows.* Wiley, second edition, 1990.

[9] E. M. L. Beale. An Alternative Method for Linear Programming. *Proccedings of the Cambridge Philosophical Society*, 50(4), 1954.

[10] P. Beck, L. Lasdon, and M. Engquist. A Reduced Gradient Algorithm for Nonlinear Network Problems. *ACM Transactions on Mathematical Software*, 9(1), May 1983.

[11] M. Bellmore, G. Bennington, and S. Lubore. A Multivehicle Tanker Scheduling Problem. *Transportation Science*, 5, 1971.

[12] D. P. Bertseka and P. Tseng. Relaxation Methods for Minimum Cost Network Flow Problems. Technical Report MIT Report LIDS-P-1339, Massachusetts Institute of Technology, October 1983.

[13] D. P. Bertseka and P. Tseng. The Relax Codes for Linear Minimum Cost Network Flow Problems. *Annals of Operations Research*, 13, 1988.

[14] Dimitri P. Bertsekas and Eli M. Gafni. Projected Newton Methods and Optimization of Multicommodity Flows. *IEEE Transactions on Automatic Control*, AC-28(12), December 1983.

[15] Stephen P. Bradley. Solution Techniques for the Traffic Assignment Problem. Technical Report ORC 65-35, University of California at Berkley, 1965.

[16] S. Frank Chang and S. Thomas McCormick. A hierarchical algorithm for making sparse matrices sparser. *Mathematical Programming*, 56(1-3), 1992.

[17] A. Charnes, W. W. Cooper, and A. Henderson. *An Introduction to Linear Programming.* Wiley, New York, 1953.

[18] Lai Chi-Yen and Jun-Min Liu. Computational Results of Karmarkar-based Interior Point Methods on Crew Planning Applications. In *Third SIAM Conference on Optimization*, Boston, MA, April 1989. AT & T, SIAM.

[19] S. Clark and J. Surkis. An Operations Research Approach to Racial Desegregation of School Systems. *Socio-Econ. Plan. Sci.*, 1, 1968.

[20] G. B. Dantzig and P. Wolfe. A Decomposition Principle for Linear Programs. *Operations Research*, 8(1), 1960.

[21] I. I. Dikin. Iterative Solution of Problems of Linear and Quadratic Programming. *Soviet Math Docl.*, 8(3), 1967.

[22] L. R. Ford and D. R. Fulkerson. A Suggested Computation for Maximal Multi-Commodity Network Flows. *Management Science*, 5(1), 1958.

[23] M. Frank and P. Wolfe. An Algorithm for Quadratic Programming. *Naval Research Logistics Quarterly*, 3. 1956.

[24] A. M. Geoffrion and G. W. Graves. Multicommodity Distribution System Design By Benders Decomposition. *Management Science*, 20(5), 1974.

[25] Philip E. Gill, Walter Murray, Michael A. Saunders, J.A. Tomlin, and Margaret H. Wright. On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method. *Mathematical Programming*, 1986.

[26] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[27] J. L. Goffin, A. Haurie, and J. P. Vial. Decomposition and Nondifferentiable Optimization with the Projective Algorithm. Technical Report 91-01-17, McGill University, January 1991.

[28] M. D. Grigoriadis. An Efficient Implementation of the Network Simplex Method. *Mathematical Programming Study*, 26, 1986.

[29] M. D. Grigoriadis and W. W. White. A Partitioning Algorithm for the Multicommodity Network Flow Problem. *Mathematical Programming*, 3, 1972.

[30] Arnoldo C. Hax and Dan Candea. *Production and Inventory Management*. Prentice-Hall, 1984.

[31] D. W. Hearn, S. Lawphongpanich, and J. A. Ventura. Restricted Simplicial Decomposition: Computation and Extensions. Technical Report 84-38, Department of Industrial and Systems Engineering, University of Florida, Gainsville, Florida, October 1984.

[32] T. C. Hu. *Integer Programming and Network Flows*. Addison-Wesley, 1969. Chapter 11, Page 177.

[33] K. L. Jones, I. J. Lustig, J. M. Farvolden, and W. B. Powell. Multicommodity Network Flows: The Impact of Formulation on Decomposition. Technical Report SOR 91-23, Princeton University, 1991.

[34] R. E. Kalaba and M. L. Juncosa. Optimal Design and Utilization of Communication Networks. *Management Science*, 3(1), October 1956.

[35] Sanjiv Kapoor and Pravin M. Vaidya. Fast Algorithms for Convex Quadratic Programming and Multicommodity Flows. 1986.

[36] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 1984.

[37] Narendra Karmarkar, Maruicio G. C. Resende, and K. G. Ramakrishnan. An interior point algorithm to solve computationally difficult set covering problems. To appear in Mathematical Programming.

[38] J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming*. Wiley, 1980.

[39] Jeff Kennington. A primal partitioning code for solving multicommodity network flow problems (version 1). Technical Report OR 79008, Southern Methodist University, June 1979.

[40] Jeff Kennington and Mohamed Shalaby. An Effective Subgradient Procedure for Minimal Cost Multicommodity Flow Problems. *Management Science*, 23(9), 1977.

[41] L. G. Khachian. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20, 1979.

[42] Masakazu Kojima, Shinji Mizuno, and Akiko Yoshise. A Primal-Dual Interior Point Algorithm for Linear Programming. Technical report, Tokyo Institute of Technology, February 1987.

[43] Leon S. Lasdon. *Optimization Theory for Large Systems.* MacMillan Publishing, 1970.

[44] David G. Luenberger. *Linear and Nonlinear Programming.* Addison Wesley, second edition, 1984.

[45] Irvin J. Lustig and Guangye Li. An Implementation of a Parallel Primal-Dual Interior Point Method for Multicommodity Flow Problems. Technical Report SOR 92-02, Princeton University, January 1992.

[46] Kevin A. McShane, Clyde L. Monma, and David Shanno. An Implementation of a Primal-Dual Interior Point Method for Linear Programming. November 1988.

[47] N. Megiddo. Pathways to the Optimal Set in Linear Programming. In *Proceedings of the 6th Mathematical Programming Symposium of Japan*, 1986.

[48] N. Megiddo and M. Shub. Boundary Behavior of Interior Point Algorithms in Linear Programming. Technical Report RJ 5319, IBM Almaden Research Center, September 1986.

[49] John E. Mitchell and Michael J. Todd. Solving Combinatorial Problems Using Karmarkar's Algorithm. Part I: Theory. January 1989.

[50] John E. Mitchell and Michael J. Todd. Solving Combinatorial Problems Using Karmarkar's Algorithm. Part II: Computational Results. January 1989.

[51] R. C. Monteiro and I. Adler. An $o(n^3l)$ primal-dual interior point algorithm for linear programming. May 1987.

[52] R. C. Monteiro and Ilan Adler. Limiting Behaviour of the Affine Scaling Continuous Trajectories for Linear Programming Problems. April 1988.

[53] B. A. Murtagh and M. A. Saunders. Large-Scale Linearly Constrained Optimization. *Mathematical Programming*, 14, 1978.

[54] J. B. Orlin. Genuinely Polynomial Simplex and Non-Simplex Algorithms for the Minimum Cost Flow Problem. Technical Report 1615-84, Alfred P. Sloan School of Management, M.I.T., 1984.

[55] R. B. Potts and R. M. Oliver. *Flows in Transportation Networks.* Academic Press, 1972.

[56] Mauricio G. C. Resende and Geraldo Veiga. A Dual Affine Scaling Algorithm for Minimum Cost Network Flow. February 1991.

[57] Mauricio G. C. Resende and Geraldo Veiga. Computational Investigation of an Interior Point Linear Programming Algorithm for Minimum Cost Network Flow. August 1991.

[58] R. Tyrrell Rockafellar. *Convex Analysis.* Princeton University Press, Princeton, New Jersey, 1972.

[59] J. B. Rosen. Primal Partition Programming for Block Diagonal Matrices. *Numerische Mathematik*, 6, 1964.

[60] Gary L. Schultz and Robert R. Meyer. An Interior Point Method for Block Angular Optimization. *SIAM Journal on Optimization*, 1(4), November 1991.

[61] Yen-Shwin Shan. *A Dynamic Multicommodity Network Flow Model for Real Time Optimal Rail Freight Car Management.* PhD thesis, Princeton University, October 1985.

[62] Cyrus James Staniec. *Solving the Multicommodity Transshipment Problem.* PhD thesis, Naval Postgraduate School, Monterey, California, June 1987.

[63] J. A. Tomlin. Minimum-Cost Multicommodity Network Flows. *ORSA*, 14(1), 1966.

[64] Robert J. Vanderbei, Marc. S. Meketon, and Barry A. Freedman. A Modification of Karmarkar's Linear Programming Algorithm. *Combinatorica*, 1986.

[65] V. Viswanathan. Yield Management at American Airlines. In *Optimization Days 1992*, Montreal, Canada, 1992. American Airlines.

[66] Balder von Hohenbalken. Simplicial Decomposition in Nonlinear Programming Algorithms. *Mathematical Programming*, 13, 1977.

[67] Larry Wall and Randal L. Schwartz. *Programming perl.* O'Reilly and Associates, Inc., August 1991.

[68] W. W. White. Mathematical Programming, Multicommodity Flows, and Communication Nets. In *Proceedings of the Symposium on Computer-Communications Networks and Teletraffic*, 1972.

[69] Phillip Wolfe. *Methods of Nonlinear Programming.* McGraw-Hill, 1963.

[70] Phillip Wolfe. Convergence theory in nonlinear programming. In *Integer and Nonlinear Programming*. North-Holland, 1967.

[71] W. I. Zangwill. The convex simplex method. *Management Science*, 14(7):429–450, March 1967.

[72] W. I. Zangwill. *Nonlinear Programming: A Unified Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1969.

[73] S Zenios. Decompositions of multicommodity network flow problems for parallel computations and parallel decomposition of multicommodity networks. undated technical note.

[74] S. A. Zenios, M. C. Pinar, and R. S. Dembo. Large Scale Nonlinear Network Models and their Application. Technical Report Report EES-86-18, Princeton University, 1986.

[75] S. A. Zenios, M. C. Pinar, and R. S. Dembo. Linear Quadratic Penalty Forms for Network Structured Problems. 1990. ORSA/TIMS Conference, Philadelphia.