The Minimum Cost Flow Problem and The Network Simplex Solution Method

By
Damian J. Kelly, B.Comm
and
Garrett M. O'Neill, B.Comm

a dissertation
presented to the National University of Ireland
in partial fulfillment of the requirements
for the degree of
Master of Management Science
at
University College Dublin

Supervised by
Dr. Fergus Gaines
and
Dr. Derek O'Connor

September 1991

NOTE : This is a re-TEX ing of the original TEX files. The only changes that have been made are (1) format and layout, and (2) EPS figures have been used instead of the original cut-and-paste figures.

Some diagrams and tables are missing but these are not too important because the data from them is summarised in other tables.

Derek O'Connor August, 2003

Abstract

The Minimum Cost Flow (MCF) Problem is to send flow from a set of supply nodes, through the arcs of a network, to a set of demand nodes, at minimum total cost, and without violating the lower and upper bounds on flows through the arcs. The MCF framework is particularly broad, and may be used to model a number of more specialised network problems, including Assignment, Transportation and Transshipment problems, the Shortest Path Problem, and the Maximum Flow problem.

The Network Simplex Method (NSM) is an adaption of the bounded variable primal simplex algorithm, specifically for the MCF problem. The basis is represented as a rooted spanning tree of the underlying network, in which variables are represented by arcs. The method iterates towards an optimal solution by exchanging basic and non-basic arcs. At each iteration, an entering arc is selected by some pricing strategy, and an arc to leave the basis is ascertained. The construction of a new basis tree is called the pivot. There are many strategies for selecting the entering arc, and these determine the speed of solution.

Acknowledgements

We would like to express our gratitude to both our supervisors, *Dr. Fergus Gaines* and *Dr. Derek O'Connor*, for their generous advice and frequent guidance, which was of immense assistance throughout the preparation of this thesis.

We also express our thanks to *Prof. David Judge* for his perserverence in obtaining some of the more obscure problem literature.

Finally, we thank *Mr. Peadar Keenan* and *Mr. Craig Kilroy* for their interest in, and assistance with, the practical application of our work.

Contents

Chapter 1

Introduction

1.1 Terms of Reference

Our guiding objectives in the preparation of this dissertation were to review the Minimum Cost Flow (MCF) Problem, and to develop an efficient implementation of the Network Simplex Method for its solution. The following presents a more detailed exposition of the topics which are the precursors to the achievement of these objectives.

1. Documentation of the historical and mathematical development of the Minimum Cost Flow Problem.
2. Demonstrate the broad applicability of the MCF framework in the formulation of other common network problems, and in the solution of practical problems.
3. Review the contemporary methods available for the solution of the MCF problem, extant in the literature.
4. Present a complete mathematical derivation of Network Simplex Method, demonstrating the correspondence between the graph theoretical and linear algebraic statements of the algorithm.
5. Examine the implementation and performance of contemporary NSM codes, with particular emphasis on data storage schemes and pricing strategies.
6. Design and test a range of pricing strategies, with a view to determining their suitability for the solution of different classes of problems.

1.2 Chapter Outline

Chapter 2 deals in general with the Minimum Cost Flow Problem. The historical background to the problem is traced, followed by a linear algebraic statement of the problem and of its specialisations. Contemporary analysis and empirical testing of solution methods is reviewed.

In Chapter 3, we discuss in detail the Network Simplex Solution Method. We present a comprehensive mathematical background to the method, including proofs of its properties, optimality conditions and techniques associated with its operation. We derive a succinct algorithmic statement of the method, and analyse its order of complexity.

The efficient implementation of the method is the subject of Chapter 4. We present alternative data storage schemes and procedures which may be availed of, and describe in detail those implemented in our own code. Particular emphasis is placed on pricing strategies.

Applications, and our empirical testing of NSM are discussed in Chapter 5. A comprehensive set of test problems are generated and solved. The results form the basis for analysis of the performance of our code. Documented practical applications of the MCF framework are reviewed, alongside one in which we are involved.

Finally, in Chapter 6, we present our conclusions.

Chapter 2

The Minimum Cost Flow Problem

2.1 Historical Background

The Minimum Cost Flow Problem (MCF) and the Network Simplex Solution Method (NSM) were initially developed quite independently. The MCF has its origins in the formulation of the classic *transportation* type problem, while NSM, as its name suggests is deduced from the *Simplex Method* for solving Linear Programming Problems.

2.1.1 Development of Transportation Type Problems

The group of problems now known as transportation problems was first studied by a Russian mathematician, L.V. Kantorovich, in a paper entitled *Mathematical Methods of Organizing and Planning Production* (1939). Largely ignored by his Russian contemporaries, his work remained unknown elsewhere until 1960, long after the seminal Western work in this field had been completed. Its value was however recognised in 1975 by the Nobel Academy , which awarded its Prize in Economic Science in equal shares to Kantorovich and T.C. Koopmans "for their contribution to the theory of optimal allocation of resources".

Kantorovich (1939) deals with nine different problem types, with the stated goal of aiding the third of Stalin's five-year plans to obtain the greatest possible usage of existing industrial resources. Principal among these problems were:

(i) the distribution of work among individual machines
(ii) the distribution of orders among enterprises
(iii) the distribution of raw materials, fuel and factors of production
(iv) the minimisation of scrap
(v) the best plan of freight shipments

A solution method was also included and was described by Marchenko, in a forward to the article, as "an original method going beyond the limits of classical mathematical analysis". In retrospect, the algorithm is rudimentary and incomplete, but its importance lies in the application of the method to "questions of organizing production". A more complete theory of the transshipment problem is to be found in *On the Translocation of Masses*, Kantorovich in 1942 and in Kantorovich and Gauvrin in 1949. The relations between primal and dual are recognized and an extension to capacitated networks is given.

In the West, the seminal work in the area was done by F. Hitchcock (1941) who outlines the standard form of the transportation problem for the first time. He proposes a $m \times n$ dimensional geometric interpretation of the transportation of goods from m factories to n cities, and constucts a "region of possibilities" on whose boundary the optimal solution must lie. A method for finding the fixed points on this boundary, called *vertices*, is proposed, and is shown to iteratively generate better solutions by expresssing the objective function in terms of variables with zero values. Hitchcock also notes the occurence of multiple optimal solutions and degeneracy.

At around the same time, T.C. Koopmans, as a member of the Combined Shipping Board during the Second World War, began research into reducing shipping times for the transportation of cargo. In *Optimum Utilisation of the Transportation System* in 1947, he

studies the transportation problem, developing the idea of node-potentials and an optimality criterion. He also shows that an extreme point, or vertex as Hitchcock terms it, is represented by a tree. It is because of the work of these two researchers that the classical transportation problem is often referred to as the *Hitchcock-Koopmans Transportation Problem.*

Much of the development of what we now refer to as the Minimum Cost Flow problem and network-based solution methods may be attributed to Ford and Fulkerson (1962). This monograph contains an extensive review and analysis of the MCF problem and its specialisations, and also includes an innovative primal-dual solution algorithm.

2.1.2 Development of the Simplex Method

The Simplex Method for solving Linear Programming Problems has a longer history than the Transportation Problem. The first instance of a Linear Programming Problem is attributed to the nineteenth century mathematician Fourier, who in 1826 studied linear inequalities to find the "least maximum deviation" fit to a system of linear equations. He reduced this problem to that of finding the optimal vertex in a solution space and suggested a method of iterative, vertex to vertex, improvement. This principle is the basis of the simplex algorithm as we know it.

The immediate predecessor of the Simplex Method was the *Input-Output* model of the national economy proposed by W. Leontief in 1936. This quantitative model was designed to trace and predict the effect of government policy and consumer trends on the various sectors of the economy, all of which were interlinked.

The accepted father of linear programming and the Simplex Method is G.B. Dantzig. From 1946 to 1947, Dantzig worked to mechanise the planning processes of the United States Air Force. He generalized the Input-Output model and, finding no method of solving the problem extant in the literature, he designed the Simplex Method.

2.1.3 Simplex Method Applied to Transportation

In 1951, Dantzig showed how the Simplex Method could be applied to transportation problems, and went on to develop a specialisation of his method for these problems. Having expressed the problem in terms of a source-destination tableau, he describes the calculation of simplex multipliers and reduced costs to iterate towards the optimal solution. The method is known as *MODI* - "modified for distribution". Dantzig noted at this point the triangularity of the basis matrix and integrality of solutions generated from problems with integer parameters. He also examines the possibilities of degeneracy and cycling, and proposes a perturbation rule to avoid them.

Alex Orden (1956) generalised Dantzig's method and applied it to the *transshipment problem* (or uncapacitated MCF), allowing the inclusion of nodes with zero supply and demand. This is achieved by representing every node as both a source of supply and a demand destination, and adding a buffer amount, greater than or equal to the total supply, to every node. The revised formulation is solved using Dantzig's transportation method with flows from a node to itself being ignored.

2.1.4 Bounded Simplex Methods

In 1955, Dantzig developed a *bounded variable* simplex method for linear programming. This method allowed the handling of upper bounds on variables without the inclusion of

explicit constraints. It was originally envisaged for the problem of scheduling requests for computing time with a restriction on the number of available computer-hours per day. The method operates in a similar fashion to the transportation method, but variables can leave when they reach either their upper bounds or lower bounds (generally zero). In 1962, Dantzig formulated this method in generic terms for the solution of a bounded transportation problem. Dantzig (1963) contains comprehensive coverage of all the above developments.

These developments led directly to the development of the Network Simplex Method as a primal solution method for Minimum Cost Flow Problems, which may be described as bounded transshipment problems. NSM is essentially the graph-theoretical (as opposed to linear-algebraic) representation of Dantzig's bounded simplex method.

2.2 Formulation and Specialisations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be a directed network consisting of a finite set of nodes, $\mathcal{N} = \{1, 2, \ldots, n\}$, and a set of directed arcs, $\mathcal{A} = 1, 2, \ldots, m$, linking pairs of nodes in $\mathcal{N}$. We associate with every arc of $(i, j) \in \mathcal{A}$, a flow x_{ij}, a cost per unit flow c_{ij}, a lower bound on the flow l_{ij} and a capacity u_{ij}.

To each node $i \in \mathcal{N}$ we assign an integer number $b(i)$ representing the available supply of, or demand for flow at that node. If $b(i) > 0$ then node i is a supply node, if $b(i) < 0$ then node i is a demand node, and otherwise, where $b(i) = 0$, node i is referred to as a transshipment node. Total supply must equal total demand.

The Minimum Cost Flow (MCF) Problem is to send the required flows from the supply nodes to the demand nodes (i.e. satisfying the demand constraints (2.1)), at minimum cost. The *flow bound* constraints, (1.3), must be satisfied. The demand constraints are also known as *mass balance* or *flow balance* constraints.

The formulation of the problem as a Linear Programming (LP) problem is as follows:

$$\textbf{minimise} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{1.1}$$

$$\textbf{subject to} \sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = b(i), \qquad \text{for all } i \in \mathcal{N} \tag{1.2}$$

$$0 \le l_{ij} \le x_{ij} \le u_{ij}, \qquad \text{for all } (i,j) \in \mathcal{A} \tag{1.3}$$

That the total net supply must equal zero can be seen by summing the flow balance equations over all $i \in \mathcal{N}$ resulting in

$$\sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = \sum_{i\in\mathcal{N}} b(i) = 0$$

The problem is described in matrix notation as

$$\textbf{minimize } \{cx | Nx = b \text{ and } 0 \le l \le x \le u\}$$

where N is a node-arc incidence matrix having a row for each node and a column for each arc.

The MCF formulation is particularly broad and can be used as a template upon which a number of network problems may be modelled. The following are some examples.

2.2.1 Shortest Paths

The shortest path problem is to find the directed paths of shortest length from a given *root* node to all other nodes. We assume, without loss of generality that the root node is 1, and set the parameters of MCF as follows: $b(1) = n - 1$, $b(i) = -1$ for all other nodes, c_{ij} is the length of arc (i, j), $l_{ij} = 0$ and $u_{ij} = n - 1$ for all arcs. The LP formulation is:

$$\textbf{minimise} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\begin{aligned}
\textbf{subject to} \quad & \sum_{j:(1,j)\in\mathcal{A}} x_{1j} - \sum_{j:(j,1)\in\mathcal{A}} x_{j1} = n - 1 \\
& \sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = -1, \qquad \text{for } i = 2, 3, \ldots, n \\
& 0 \le x_{ij} \le n - 1, \qquad \text{for all } (i, j) \in \mathcal{A}
\end{aligned}$$

The optimum solution to this problem sends unit flow from the root to every other node along a shortest path.

2.2.2 Maximum Flow

The maximum flow problem is to send the maximum possible flow from a specified *source* node (node 1) to a specified *sink* node (node n). The arc $(n, 1)$ is added with $c_{n1} = -1$, $l_{n1} = 0$ and $u_{n1} = \infty$. The supply at each node is set to zero, i.e. $b(i) = 0$ for all $i \in \mathcal{N}$, as is the cost of each arc $(i, j) \in \mathcal{A}$. Finally, u_{ij} represents the upper bound on flow in arc (i, j) to give the following LP formulation:

$$\begin{aligned}
& \textbf{minimise} \qquad -x_{n1} \\
\textbf{subject to} \quad & \sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = 0, \qquad \text{for all } i \in \mathcal{N} \\
& 0 \le x_{ij} \le u_{ij}, \qquad \text{for all } (i, j) \in \mathcal{A} \\
& 0 \le x_{n1} \le \infty
\end{aligned}$$

2.2.3 The Assignment Problem

The graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ is composed as follows: $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$ where $\mathcal{N}_1$ and $\mathcal{N}_2$ are two disjoint sets of equal size. The problem is to assign each element in $\mathcal{N}_1$ to one element in $\mathcal{N}_2$ (e.g. people to machines), at minimum cost. If element $i \in \mathcal{N}_1$ is assigned to element $j \in \mathcal{N}_2$, then $x_{ij} = 1$. Otherwise $x_{ij} = 0$. The cost of assigning $i \in \mathcal{N}_1$ to $j \in \mathcal{N}_2$ is represented by c_{ij}. For each $i \in \mathcal{N}_1, b(i) = 1$, and for each $j \in \mathcal{N}_2$, $b(j) = -1$. All lower bounds are 0, and all upper bounds are 1. The LP formulation is:

$$\textbf{minimise} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\textbf{subject to} \sum_{j:(i,j)\in\mathcal{A}} x_{ij} = 1, \qquad \text{for all } i \in \mathcal{N}_1$$

$$\sum_{i:(i,j)\in\mathcal{A}} -x_{ij} = 1, \qquad \text{for all } j \in \mathcal{N}_2$$

$$x_{ij} \in \{0,1\}, \qquad \text{for all } (i,j) \in \mathcal{A}$$

2.2.4 The Transportation Problem

As for the Assignment Problem, the Transportation Problem is based on a bi-partite graph, where $\mathcal{N}_1$ is the set of source nodes, and $\mathcal{N}_2$ is the set of sink or *destination* nodes, such that $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$, and the set of arcs is defined by $\mathcal{A} = \{(i,j)|i \in \mathcal{N}_1, j \in \mathcal{N}_2\}$

The objective is to find the least cost shipping plan from the sources of supply ($\mathcal{N}_1$) to the destinations ($\mathcal{N}_2$), where x_{ij} is the number of units shipped from source i to destination j, and c_{ij} is the cost of shipping one unit from i to j. The supply and demands at sources and destinations respectively are denoted by $b(i)$. There are no upper bounds on flows. (A problem with no upper flow bounds is said to be *uncapacitated*).

In LP form, the problem is stated:

$$\textbf{minimise} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\textbf{subject to} \sum_{j:(i,j)\in\mathcal{A}} x_{ij} = b(i), \qquad \text{for all } i \in \mathcal{N}_1$$

$$\sum_{i:(i,j)\in\mathcal{A}} x_{ij} = -b(j), \qquad \text{for all } j \in \mathcal{N}_2$$

$$x_{ij} \geq 0, \qquad \text{for all } (i,j) \in \mathcal{A}$$

2.2.5 The Transshipment Problem

The Transshipment problem is a generalisation of the above with the addition of intermediate, zero-supply nodes called transshipment nodes. We divide the nodes into three subsets: $\mathcal{N}_1$, a set of supply nodes, $\mathcal{N}_2$, a set of demand nodes and $\mathcal{N}_3$, a set of transshipment nodes. The problem is equivalent to the transportation problem in all other respects. The objective is to minimise shipping costs and the arcs are uncapacitated. The LP formulation is:

$$\textbf{minimise} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\textbf{subject to} \sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = b(i), \qquad \text{for all } i \in \mathcal{N}_1, \mathcal{N}_2$$

$$\sum_{i:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = 0, \qquad \text{for all } j \in \mathcal{N}_3$$

$$x_{ij} \geq 0, \qquad \text{for all } (i,j) \in \mathcal{A}$$

The only difference between the transshipment formulation and that of the general MCF is that the latter includes upper and lower bounds on the flows in the arcs.

2.3 Contemporary Solution Methods

A variety of solution methods for the MCF problem are described in the literature. These may be categorised into a number of principal approaches, namely *primal, dual, primal-dual* and *scaling* alogorithms. For example, *negative cycle* and *primal simplex* (as its name suggests) are both primal algorithms, while dual methods include *successive shortest paths* and a specialisation of the *dual simplex* method for LP problems. Ford's *Out-of-Kilter* algorithm is perhaps the best known primal-dual method. Finally, there exist a family of relaxation algorithms based on either *right-hand-side* or *cost* scaling. We briefly review each of these methods before analyzing the Network Simplex Method in greater detail.

2.3.1 Note on Complexity Analysis

Complexity Analysis of an algorithm is a method of providing performance guarantees by giving worst case upper bounds on the running time. The bound is stated in terms of the following parameters:

n, m: the number of nodes and arcs respectively,
C, U: an upper bound on the arc costs and capacities.

> Tarjan (1983) uses the following notation for asymptotic running times, which we have adopted:
>
> If f and g are functions of nonnegative variables $n, m, \ldots$ we write "f is $O(g)$" if there are positive constants c_1 and c_2 such that $f(n, m, \ldots) \leq c_1 g(n, m, \ldots) + c_2$ for all values of $n, m, \ldots$.

An algorithm is said to be a polynomial-time algorithm if its running time is bounded by a polynomial function of n, m, $\log C$, and $\log U$. It is strongly polynomial if polynomial functions of n and m only, bound the running time, and pseudo-polynomial if polynomially bounded in terms of n, m, C, and U. If we require the removal of C and U from the bound then we invoke the *Similarity Assumption*. This is the assumption that both C and U are polynomially bounded in n, i.e. that $C = O(n^k)$ and $U = O(n^k)$. The terms 'running time' and 'order of complexity' are used interchangeably.

As many MCF solution methods include the finding of shortest paths or maximum flows as procedures, we need to define the order of complexities of these procedures. The term $S(\cdot)$ is denotes the order of complexity of Shortest Path Problem, and $M(\cdot)$ is the corresponding running time for a Maximum Flow Problem. The following bounds are taken from Ahuja, Magnanti and Orlin (1989).

Polynomial-Time Bounds:

$$S(n, m, C) = \min\{m \log\log C, m + n\sqrt{\log C}\}$$
$$M(n, m, U) = mn \log\left(\frac{n\sqrt{\log U}}{m} + 2\right)$$

Strongly Polynomial-Time Bounds:

$$S(n,m) = m + n\log n$$
$$M(n,m) = nm\log(n^2/m)$$

2.3.2 Negative Cycle Algorithm

This algorithm was devised by Klein (1967) and is based on the observation that an optimal solution to the MCF problem admits no negative cost augmenting cycle. In outline form the algorithm involves finding an initial feasible flow through the network, and subsequently identifying any negative cost augmenting cycles. Such cycles are eliminated by augmenting flow in all arcs of the cycle by the maximum amount possible.

The method used by Klein to identify a negative cycle is based on a label-correcting shortest path algorithm. Given that mCU is an upper bound on the initial (feasible) flow cost, and that the lower bound for the optimal cost is 0, we derive a bound on the complexity of the algorithm as follows:

Since the algorithm reduces flow cost by at least one unit at each iteration, a bound on the number of iterations is $O(mCU)$. If $S(m,n)$ is the complexity of the shortest path problem, then a bound on the complexity of Klein's (1967) algorithm is

$$O\big(mCU\ S(n,m)\big) = O(nm^2CU) \text{ for } S(n,m) = O(nm)$$

Weintraub (1974) suggested a variant of this algorithm whereby the *most* negative cycle is introduced into the feasible flow at each stage. An analysis of this algorithm by Barahona and Tardes in 1987 yielded a bound of

$$O\big(m^2\log(mCU)\ S(n,m,C)\big)$$

This algorithm also yields a method, based on the negative incremental cost of the incoming cycle at each iteration, for determining an upper bound on the difference in cost between the current and optimal solutions. This gives the option of algorithm termination once a solution is reached which is close to optimal.

2.3.3 Successive Shortest Path (SSP) Algorithm

The SSP algorithm was developed independently by Jewell in 1958, Iri in 1960 and Busaker and Gowan in 1961. The algorithm maintains dual feasibility and strives to attain primal feasibility. The flow bound constraints are always satisfied while the supply/demand constraints are initially violated. The algorithm generally proceeds as follows:

At each stage, a node i with excess supply, and a node j with unfulfilled demand are selected. A shortest path (i.e. path of minimum per unit cost with excess capacity) from i to j is found, and flow is sent along this path from i to j. The algorithm terminates when the current solution satisfies all the supply and demand constraints.

Each iteration reduces the supply of a node by at least one unit. Thus nU is a bound on the number of iterations. This gives a complexity for the SSP algorithm of

$$O\big(nU\ S(n,m,C)\big)$$

Edmonds and Karp (1972) modified the algorithm slighlty so that all shortest path computations are performed on networks with non-negative arc costs. They then devised a scaling algorithm (see section 2.3.6 below) based on SSP with a complexity bound of

$$O\big((n+m)(\log U)\ S(n,m,C)\big)$$

2.3.4 Primal-Dual and Out-of-Kilter Algorithms

The primal dual algorithm for the MCF problem is described in Ford and Fulkerson (1962). It is similar to the successive shortest path algorithm but uses many paths to augment the flow at each iteration.

The MCF is transformed into a single source, single sink problem, generally by the addition of a super-source, s and super-sink, t. At each iteration, the node potentials are updated by finding a tree of shortest paths with the node s as its root. A maximum flow problem is then solved with the object of sending flow from s to t using only arcs with zero reduced cost. Thus the excess supply of some node strictly decreases as does the node potential of the sink. Upper bounds on these are nU and nC respectively. This gives an order of complexity for the Primal-Dual algorithm of

$$O\bigg(\min\big(nU\ S(n,m,C), nC\ M(n,m,U)\big)\bigg)$$

The Out-of-Kilter algorithm was developed by Minty (1960), who applied the it to the analysis of electrical networks, and by Fulkerson in 1961. It maintains a solution satisfying the mass balance constraints but violating the dual feasibility and flow bound conditions.

The idea behind the algorithm is to adjust the flow in an arc until it is consistant with the reduced cost of that arc with regard to the dual-derived optimality conditions, i.e.

$$\text{if } \bar{c}_{ij} \begin{cases} > 0 & \text{drive } x_{ij} \text{ to } 0 \\ < 0 & \text{drive } x_{ij} \text{ to } u_{ij} \\ = 0 & \text{allow any flow } 0 \le x_{ij} \le u_{ij} \end{cases}$$

The kilter number of an arc, k_{ij}, is defined as the minimum increase or decrease in the flow necessary so that the arc satisfies its flow bound and dual feasibility constraints. The algorithm iteratively reduces the kilter numbers of the arcs by augmenting flow in cycles, until all arcs are 'in kilter' and the MCF is solved to optimality.

2.3.5 Specialisation of LP Dual Simplex

A specialisation of the dual simplex LP method for Network Flow Problems has been studied and analysed by Helgason and Kennington (1977), and by Armstrong, Klingman and Whitman (1980).

Helgason and Kennington (1977) state that a Dual Simplex algorithm requires all the computational machinery of a Primal Simplex algorithm, in addition to the ability to generate rows of the inverse of the current basis matrix. They go on to show that there are at most two distinct elements in such a row. Their dual simplex code for the MCF problem is based on a simple algorithm to find these distinct elements.

On testing their algorithm, however, the authors conclude that its use should be reserved for situations where a good starting dual feasible basis is available.

Armstrong, Klingman and Whitman (1980), test various data structures and various pricing rules, including implementations of multiple pricing criteria. The most efficient combination was selected and the resulting algorithm implemented as DNET2. The authors tested their algorithm against PNET1, a then state of the art primal simplex code, devised by Glover,Karney and Klingman, and Glover, Klingman and Stutz in 1974. While faster than earlier dual implementations, their testing showed that, even for problems specially suited to the dual code, the special purpose, primal simplex based approaches are superior, in terms of both storage and speed, to both dual and Out-of-Kilter methods.

2.3.6 Scaling Algorithms

The basic idea underlying the scaling algorithms is that they commence with a feasible flow for a transformed or altered set of constraints, and iteratively, in successively smaller steps, move towards an optimal solution. Scaling algorithms are generally subdivided into *Right-Hand* and *Cost* scaling algorithms.

Right-Hand scaling is derived from the successive shortest path algorithm. Edmonds and Karp (1972) used this algorithm to derive the first polynomial bounded algorithm for the MCF problem. First, the problem is transformed into an uncapacitated one. The 'node imbalance' is defined as the excess supply or unfulfilled demand at each node. The smallest power of two, θ, for which all supply node imbalances are less than 2θ and/or all demand node imbalances are greater than -2θ, is found. We augment flow of θ along shortest paths, from supply nodes whose imbalances are greater than θ to demand nodes with imbalances less than $-\theta$, as often as possible. Set $\theta = \theta/2$ and repeat until θ is less than unity, at which point an optimal solution has been reached.

Cost scaling was developed independently by Bertsekas in 1979 and Tardos in 1985. In this case, the dual optimality conditions are relaxed, to form ϵ-optimality conditions. A pseudo-flow is ϵ-optimal if all the dual optimality conditions are within ϵ units of being satisfied. The cost scaling algorithm scales upon ϵ. Initially $\epsilon = C$, and at each stage the algorithm converts an ϵ-optimal flow into a $\frac{1}{2}\epsilon$-opitmal flow, until $\epsilon < \frac{1}{n}$, at which stage optimality is reached. A bound on the complexity of the algorithm, derived by Goldberg and Tarjan in 1987, is $O(n^2 m \log nC)$. Using a different pseudo-flow conversion scheme – called the 'wave' method – this can be reduced to $O(n^3 \log nC)$.

Double scaling algorithms have also been developed, specifically by Gabow and Tarjan in 1987, and by Ahuja, Golberg, Orlin, and Tarjan (1988). These consist of an overall cost scaling approach, within which right-hand scaling is used to ensure that a positive amount of flow augmentation is possible at each stage.

From a theoretical point of view, the scaling approach to solving the MCF problem is very important. Scaling type algorithms are the basis of virtually all polynomially-bounded MCF algorithms. Among the fastest of these is one due to Gabow and Tarjan, developed in 1987, with a bound of

$$O(nm \log n \log U \log nC)$$

Another, developed in the same year by Goldberg and Tarjan, has a bound of

$$O(nm \log n \log nC),$$

but is very expensive in terms of storage. Finally, an algorithm by Ahuja, Golberg, Orlin and Tarjan, developed in 1988, is bounded by

$$O(nm \log \log U \log nC)$$

Scaling algorithms also figure prominently in the development of *strongly* polynomial time algorithms for the MCF problem. The first strongly-polynomial MCF algorithm is due to Eva Tardos in 1985, with a bound of $O(m^4)$. The fastest scaling algorithm described in the literature to date is due to Orlin in 1988, with a bound

$$O\big(m \log nS(n,m)\big)$$

These algorithms are being used in some ground-breaking work by Bertsekas and Eckstein (1988), who are trying to develop an algorithm for the MCF problem that is particularly suited to implementation on a computer with the parallel processing capabilities.

Despite their theoretical pedigree, Ahuja, Magnanti and Orlin (1989) consider that the comparative performance of scaling and non-scaling algorithms in empirical tests has not been fully assessed. For this reason, the continued development of non-scaling algorithms must remain a priority in MCF problem solution.

2.3.7 Relaxation Algorithms

A variant on the primal-dual class of algorithms has been proposed by Bertsekas (1985) and Bertsekas and Tseng (1988). The approach of these relaxation algorithms is to maintain a pseudo-flow satisfying the flow bound constraints and to move towards satisfaction of the flow balance constraints. This is done through an iterative descent of the dual functional. The dual price vector, π, and the corresponding flow vector, x, which together satisfy complementary slackness, are maintained, and the algorithm iterates using procedures based on flow augmentation and price adjustment.

The code developed from the algorithm in 1985 was called RELAX, with RELAX-II and RELAXT-II developed in 1988. Bertsekas (1985) claims that the RELAX method is theoretically pseudo-polynomial. In both papers, the code is comprehensively tested using the NETGEN benchmark problems, developed by Klingman, Napier and Stutz (1974).

2.3.8 Network Simplex Method

The Network Simplex Method is an adaption of the bounded variable primal simplex algorithm, specifically for the MCF problem. The basis is represented as a rooted spanning tree of the underlying network, in which variables are represented by arcs, and the simplex multipliers by node potentials. At each iteration, an entering variable is selected by some pricing strategy, based on the dual multipliers (*node potentials*), and forms a cycle with the arcs of the tree. The leaving variable is the arc of the cycle with the least augmenting flow. The substitution of entering for leaving arc, and the reconstruction of the tree is called a *pivot*. When no non-basic arc remains eligible to enter, the optimal solution has been reached.

The first tree manipulation structure was suggested by Johnson in 1966. The first implementations are those of Srinivasan and Thompson (1973) and Glover, Karney, Klingman and Napier (1974). These papers develop the NSM for the transportation problem, and both result in running times which offer significant reductions over previous solution times for these problems. Srinivisan and Thompson (1973) tested a variety of pricing rules including arc-sampling and a precursor of a candidate list, both of which yield promising results.

In Glover, Karney, Klingman and Napier (1974), the pricing strategies used are more specific to the transportation problem. One significant finding was that a *satisficing* rule – i.e. selecting a 'good', though not necessarily the best entering arc – results in faster solutions. A series of different methods for generating a starting solution were also tested, as the code implemented a two-phase algorithm. The authors found that those methods allowing rapid commencement of the main part of the algorithm were preferable. Those which generated a lower cost starting solution were judged to take too long to do so.

In Bradley, Brown and Graves (1977), the basis is stored as a spanning tree defined by a parent array, i.e. an array of pointers to the unique immediate predecessors of each node. They consruct a general code for solving MCF problems, designed with speed of execution as a priority, and with the capablility to handle very large scale problems. The pricing strategy adopted by the authors involves a *candidate queue* of so-called 'interesting' nodes and arcs, and is performed in three stages. In the opening gambit an initial solution is quickly generated using artificial arcs. In the middle game, many arcs are eligible to enter, and so exhaustive pricing is unnecessary. Finally, the end game concentrates on the few remaining, favourably priced arcs, until the optimal solution is reached.

Another candidate list strategy is described by Mulvey (1978). A limited number of favourably priced entering arcs are stored on a list formed by scanning the non-basic arcs in a *major* iteration. In each *minor* iteration, the most favourably priced arc on the list is chosen to enter the basis and a pivot is performed. Minor iterations continue until either the candidate is empty, or until a preset limit on the number of minor iterations is reached. The speed of solution may be influenced by varying the limits on list-length and minor iterations. Mulvey (1978) concludes that partial suboptimization in the choice of entering arc is suited to sparse networks, and that worst case analysis of the number of iterations is a poor guide to performance.

Goldfarb and Reid (1977) proposed a *steepest edge* pricing criterion. Instead of selecting entering arcs on the basis of their reduced cost, they are selected by taking into account the changes in all arcs on the cycle. Grigoriadis (1986), however, found this approach to be a poor one. His own code is perhaps the fastest primal simplex code for the MCF problem yet developed. Having reviewed and dismissed candidate list pricing strategies as excessively complex and not sustainable by mathematical argument, the author opts for a simple 'best in arc block' pricing rule. A block of arcs of preset size is priced at each iteration, and the most favourably priced of these enters the basis.

The starting solution is completely artificial. With respect to this, Grigoriadis (1986) discusses a *gradual penalty* method, whereby the assigned cost of the flow in artificial arcs is gradually increased until just sufficient to eliminate these arcs from the solution. This, it is claimed, reduces the distortion of the cost function, and results in a lesser number of pivots.

2.3.9 NSM and Degeneracy

A *degenerate* pivot is one in which the flow augmentation is zero and thus the value of the objective function reamins unchanged. Degeneracy is noted by Bradley, Brown and Graves (1977) as a critical issue in solving the MCF problem, and particularly in its transportation and assignment specialisations. In some tests involving these problem types, up to 90% of pivots were degenerate. The authors suggest a method of detecting degenerate pivots, which triggers the abortion of such a pivot.

Gavish, Schweitzer and Shlifer (1977) also noted 90% – 95% degeneracy in transportation and assignment problems. They proposed selecting the leaving arc as the one with the greatest cost coefficient, which reduced the total number of pivots by 50%.

The most significant development, however, was the *strongly feasible basis* technique proposed by Cunningham (1976). A strongly feasible basis is defined as one in which a *positive* amount of flow can be sent from any node to the root without violating any of the flow bound constraints. An all-artificial starting solution generates such a basis, which is then maintained by imposing controls on the selection of the leaving arc. The author proves that his algorithm is finite.

This method was independently developed by Barr, Glover and Klingman (1977) and applied to assignment problems. They form a subset of bases that are capable of leading to an optimal solution, and select a new basis from this subset at each stage. Their algorithm is also shown to be finite.

Cunningham (1979) showed that although these methods prevent *cycling* – i.e. the occurence of an infinite number of degenerate pivots – a phenomenon known as *stalling* could still occur. Stalling is the occurence of an exponential, though not infinite, number of consecutive degenerate pivots. In a 1973 paper, Zadeh describes a set of pathological examples for which network algorithms, including NSM, perform an exponential number of iterations. By not allowing an arc to remain continually available to enter the basis, Cunningham (1979) shows that stalling can also be avoided. Thus each arc must be considered periodically as a candidate entering arc. The author proposes a number of rules to achieve this including 'Least Recently Considered Arc', and 'Least Recently Basic Arc', along with two other methods which cyclically traverse a fixed ordering of nodes looking at the arcs incident with those nodes.

Grigoriadis (1986), while noting the results of the previous studies, does not use the strongly feasible basis technique. Rather, he relies on the fact that cycling is rare in practical applications, and implements a rule designed to minimise the updating of the basis structure. It should be noted also that the "best in arc block" pricing strategy is one which cycles through the entire set of arcs, thus periodically checking every arc as Cunningham (1979) recommends.

2.4 Empirical Testing of Algorithms

The most recent and relevant testing of MCF algorithms is documented by Bertsekas (1985), Grigoriadis (1986), and Bertsekas and Tseng (1988). Bertsekas (1985) tests representative algorithms of various methods, in solving forty benchmark problems produced by NETGEN. The NETGEN program was devised by Klingman, Napier and Stutz (1974) with the following three objectives:

1. "To permit codes developed for a more general class of problems to be easily tested on special subclasses."
2. "To encourage standardization of data specification for all types of network problems."
3. "To facilitate computational studies of parameter variation."

The authors also specify the problem parameters of forty test problems. By using the same random seed as Klingman, Napier and Stutz (1974), a researcher may replicate the exact problems again and again. The forty benchmark problems are divided into the following into four classes:

Table I

NETGEN Problem Classes

Class	*Problem Type*	*No. of Problems*	*Range of n*	*Range of m*
I	Transportation	10	200- 300	1300- 6300
II	Assignment	5	400	1500- 4500
III	Transshipment	20	400-1500	1306- 5730
IV	Large MCF	5	3000-8000	15000-35000

The algorithms tested by Bertsekas (1985) were as follows:

RELAX	the author's own relaxation code
RNET	an implementation of NSM by Grigoriadis and Hsu in 1980
KILTER	an Out-of-Kilter code due to Aashtiani and Magnanti in 1976
VKILTER	a modification of KILTER including elements of RELAX
PDUAL	an implementation of the primal-dual algorithm

The testing was performed on a VAX 11/750 using standard FORTRAN, compiled under VMS version 3.7 in optimize mode. Table II shows a summary of the test results obtained. All times are stated in seconds, and represent the total solution for *all* problems in each class.

Table II

NETGEN Solution Times – Bertsekas (1985)

Class	RELAX	RNET	KILTER	PDUAL	VKILTER
I	45.22	94.30	251.85	367.71	75.19
II	14.02	39.21	56.89	71.60	34.7
III	138.50	156.66	609.20	819.50	209.08
IV	1124.96	1141.44	3252.59		

As the codes represent state of the art implementations of the various algorithms, Bertsekas (1985) concludes that only the relaxation and NSM codes are competitive, with RELAX performing slightly better than RNET.

Grigoriadis (1986) begins by setting out the superiority of RNET over other NSM implementations. He states that PNET, due to Klingman, Napier and Stutz, is an order of magnitude slower than RNET, while GNET, an implementation by Bradley, Brown and Graves (1977), is about one and a half times slower than RNET. The author then tests the original RNET along with an updated version, which we refer to as RNET II. The updated version includes a *Gradual Penalty Method* for the elimination of artificial arcs from an all artificial starting solution. The speed of both RNET implementations is compared to that of RELAX. These tests were performed on an IBM/370-168 using FORTRAN-H, and executed under VM/CMS. The author's results are summarised in the table below.

Table III

NETGEN Solution Times – Grigoriadis (1986)

Class	RNET-II	RNET	RELAX
I	13.23	28.12	18.76
II	5.40	13.90	4.17
III	26.08	53.34	66.08
IV	130.21	402.96	693.73

From these results, Grigoriadis (1986) concludes that the updated RNET offers the most efficient solution method for all but assignment problems. Bertsekas and Tseng (1989) refute these results by claiming that Grigoriadis (1986) had used an altered version of the original RELAX code. Two updates, RELAX-II and RELAXT-II, are described by the authors, the latter having larger memory requirements than the former. Both of these are then tested against RNET on a VAX 11/750. As in Bertsekas (1985), all codes are in standard FORTRAN, compiled under VMS version 3.7. The following results were obtained:

Table IV

NETGEN Solution Times – Bertsekas and Tseng (1989)

Class	RELAX-II	RELAXT-II	RNET
I	37.32	30.42	96.22
II	9.51	10.47	40.37
III	94.66	92.93	157.76
IV	295.55	232.65	882.07

Bertsekas and Tseng (1989) conclude that the relaxation method is several times faster than NMS on the standard benchmark problems, and that the speed-up factor increases with problem dimension.

The only firm conclusion which we can draw from these studies is that NSM and relaxation methods are the fastest methods presently available for the solution of MCF problems. We feel that it is impossible to definitively state which method is strictly better for the following reasons:

1. The testing by Grigoriadis (1986) was performed using costs expressed to a greater degree of accuracy than the testing by Bertsekas (1985) or Bertsekas and Tseng (1989). In all cases, this would bias testing in favour of the respective authors' codes, as the relaxation methods are more sensitive to cost changes than NSM.
2. Grigoriadis (1986) is accused of using an altered RELAX code.
3. It is not clear whether Bertsekas and Tseng (1989) used the updated RNET code, which we referred to as RNET-II.
4. The various tests were performed on different machines, with the codes generated using different compilers.
5. Glover and Klingman in 1982, as reported in Bertsekas and Tseng (1988) indicates that a commercial NSM code, called ARCNET, is slightly superior to RNET.

2.4.1 DIMACS Implementation Challenge 1991

The Rutgers University centre for *Discrete Mathematics and Computer Science* (DIMACS) has organized a 'challenge' in which researchers may implement any MCF algorithm, in any language and on any machine. The implementations will be tested on standard sets of problems, with the codes and results to be discussed and published at a workshop at DIMACS in October 1991. The purpose of the challenge is to resolve the controversy which surrounds the reported performances of the various MCF algorithms and their implementations.

Chapter 3

The Network Simplex Method

3.1 Mathematical Background

The following is an outline of the mathematical justification of the Network Simplex Method. We begin by deriving the optimality conditions for the Minimum Cost Flow problem, based on the concept of flow augmenting cycles. We then deduce the representation of the basis as a spanning tree of the underlying network. Next we show that an optimal solution is an integer-valued extreme point. Finally, problem transformations and dual optimality conditions are described. The material presented here draws mainly form the following sources: Bazarra, Jarvis and Scherali (1977), Chvátal (1983) and Ahuja, Magnanti and Orlin (1989).

3.1.1 Flow Decomposition and Optimality Conditions

Flow Decomposition Property:

Every directed path and cycle flow has a unique representation as non-negative arc flows. Every non-negative arc flow x can be represented as a directed path and cycle flow, though not necessarily uniquely.

Proof:

In the arc formulation, the decision variables are x_{ij}, the flow along each arc $(i,j) \in A$. In the path and cycle formulation, we enumerate the set $\mathcal{P}$ of all paths in the network, the set $\mathcal{Q}$ of all cycles in the network, and the decision variables are $h(p)$, the flow along path p and $f(q)$, the flow along cycle q, for all $p \in P$ and $q \in Q$. Define ϕ_{ij} as follows:

$$\phi_{ij}(p) = \begin{cases} 1 & \text{if arc } (i,j) \text{ is contained in path } p \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_{ij}(q) = \begin{cases} 1 & \text{if arc } (i,j) \text{ is contained in cycle } q \\ 0 & \text{otherwise} \end{cases}$$

Thus a set of path and cycle flows determines a set of arc flows as follows:

$$x_{ij} = \sum_{p \in P} \phi_{ij}(p) h(p) + \sum_{q \in Q} \phi_{ij}(q) f(q)$$

That arc flows can be represented as path and cycle flows is shown algorithmically:

Select i_0, a supply node. Some arc (i_0, i_1) carries positive flow. If i_1 is a demand node, stop. Otherwise, by the flow balance constraints some arc (i_1, i_2) carries positive flow. continue until we encounter a demand node (case 1) or a previously encountered node (case 2).

In case 1 we have a directed path p from i_0 to i_k consisting solely of arcs with positive flow.

$$\begin{aligned} \text{Let} \quad & h(p) = \min\big\{b(i_0), -b(i_k), \min\{x_{ij} | (i,j) \in p\}\big\} \\ & b(i_0) \leftarrow b(i_0) - h(p) \\ & b(i_k) \leftarrow b(i_k) + h(p) \\ \text{and} \quad & x_{ij} \leftarrow x_{ij} - h(p) \qquad \text{for all arcs } (i,j) \in p \end{aligned}$$

In case 2 we have a directed cycle q.

$$\begin{aligned} \text{Let} \quad & f(q) = \min\{x_{ij} | (i,j) \in q\} \\ \text{and} \quad & x_{ij} \leftarrow x_{ij} - f(q) \qquad \text{for all arcs } (i,j) \in q \end{aligned}$$

Continue until all $b(i) = 0$. Thus there exist no supply or demand nodes. Select a transshipment node with at least one outgoing arc of positive flow as a starting point and repeat the procedure. In this case cycles must be found. Terminate when $x = 0$.

The original flow is now the sum of the flows on the identified paths and cycles. Every time we identify a path, the supply of some node or the flow along some arc is reduced to zero, and every time we identify a cycle we reduce the flow along some arc to zero. Therefore the number of paths and cycles is less than or equal to $m + n$, and the number of cycles is less than or equal to m. ■

It is not necessary to show explicitly that this holds in the undirected case, since an undirected network may always be represented as a directed one with twice the number of arcs.

Augmenting Cycles:

A cycle q with flow $f(q) > 0$ is an *augmenting cycle* with respect to flow x if

$$0 \leq x_{ij} + \phi_{ij}(q) f(q) \leq u_{ij} \qquad \text{for each arc } (i,j) \in q$$

i.e. adding flow in cycle does not exceed flow bounds.

The *cost* of an augmenting cycle q is

$$c(q) = \sum_{(i,j) \in \mathcal{A}} \phi_{ij}(q) c_{ij}$$

i.e. the change in cost for one unit of flow augmented along the cycle.

Augmenting Cycle Property:

If x and y are two feasible solutions to an MCF problem, then y equals x plus the flow on at most m augmenting cycles with respect to x. The cost of y equals the cost of x plus the cost of flow along these augmenting cycles.

Proof:

Suppose x and y are two MCF solutions. Then

$$\begin{aligned} & Nx = b \text{ and } 0 \leq x \leq u \\ \text{and} \quad & Ny = b \text{ and } 0 \leq y \leq u \end{aligned}$$

(after problem transformation to set $l_{ij} = 0$ for all $(i,j) \in \mathcal{A}$)

Let z be the difference vector $z = x - y$. Then

$$\begin{aligned} Nz &= Nx - Ny \\ &= b - b = 0 \end{aligned}$$

Thus by the flow decomposition property we can represent z by $r \leq m$ cycle flows. For each arc $(i,j) \in \mathcal{A}$

$$z_{ij} = \sum_{k=1}^{r} \phi_{ij}(q_k) f(q_k)$$

Since $y = x + z$

$$0 \leq y_{ij} = x_{ij} + \sum_{k=1}^{r} \phi_{ij}(q_k) f(q_k) \leq u_{ij} \quad \text{for each } (i,j) \in \mathcal{A}$$

By the flow decomposition property,

$$\phi_{ij}(q_k) \geq 0 \text{ and } f(q_k) \geq 0 \quad \text{for each } k \text{ and each } (i,j) \in \mathcal{A}$$

Therefore adding any of the cycle flows q_k to x gives a feasible resultant flow and therefore each cycle $q_1, \ldots, q_r$ is an augmenting cycle with respect to flow x.

$$\begin{aligned}
\sum_{(i,j)\in\mathcal{A}} c_{ij} y_{ij} &= \sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij} + \sum_{(i,j)\in\mathcal{A}} c_{ij} z_{ij} \\
&= \sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij} + \sum_{(i,j)\in\mathcal{A}} c_{ij} \left(\sum_{k=1}^{r} \phi_{ij}(q_k) f(q_k) \right) \\
&= \sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij} + \sum_{k=1}^{r} c(q_k) f(q)
\end{aligned}$$

Thus the cost of y equals the cost of x plus the cost of flow along these augmenting cycles. ■

Optimality Conditions:

A feasible flow x is an optimum flow if and only if it admits no negative cost augmenting cycle.

Proof:

Let x be any feasible solution and x^* be the optimal solution such that $x \neq x^*$. The augmenting cycle property implies that $x^* - x$ can be decomposed into r augmenting cycles, and the sum of the costs of these cycles equals $cx^* - cx$.

If $cx^* < cx$ then $cx^* - cx < 0$ and therefore one of the cycles must have a negative cost.

If all cycles in the decomposition of $x^* - x$ are of non-negative cost then $cx^* - cx \geq 0$ which implies $cx^* \geq cx$.

But since x^* is optimum, we must have $cx^* = cx$, which means that we have obtained another optimum solution because $x \neq x^*$. ■

3.1.2 Cycle Free and Spanning Tree Solutions

Cycle Free Property:

If the objective function value of the MCF problem

$$\textbf{minimise } \{cx : Nx = b, l \leq x \leq u\}$$

is bounded from below on the feasible region and the problem has a feasible solution, then at least one cycle free solution solves the problem.

Proof:

With respect to a given flow x:

arc (i, j) is a *free* arc if $l_{ij} < x_{ij} < u_{ij}$

arc (i, j) is a *restricted* arc if $x_{ij} = l_{ij}$ or u_{ij}

A solution x has the *cycle free property* if it contains no cycles entirely consisting of free arcs.

If a solution x contains a cycle (i.e. it is not cycle free) calculate the cycle cost by traversing the cycle and summing the costs of all the forward arcs and subtracting all the costs of the reverse arcs.

If the cycle cost is negative we send a flow of θ around the cycle in the direction of traversal. If it is positive send a flow of θ around the cycle in the opposite direction. In either of these cases we make θ as large as possible, i.e. increasing θ until one of the arcs in the cycle, say arc (k, l), reaches its upper or lower bound. Set the flow at this level, thus making the arc (k, l) a *restricted* arc and eliminating the cycle. If the cycle cost is 0 we can similarly remove the cycle by increasing the flow in either direction around the cycle without any cost penalty.

Applying this argument repeatedly we can remove all cycles from the solution and establish the cycle free property. ■

Spanning Tree Property:

If the objective function value of the MCF problem

$$\textbf{minimise } \{cx : Nx = b, l \leq x \leq u\}$$

is bounded from below on the feasible region and the problem has a feasible solution then at least one spanning tree solution solves the problem.

Proof:

This is an extension of the cycle free property. Given that the network is connected, let $\mathcal{S}$ be the set of free arcs. If the free arcs connect all the nodes of $\mathcal{G}$, then the solution is connected and acyclic (from the cycle free property), thus forming a spanning tree. Otherwise, we add to $\mathcal{S}$ some restricted arcs until $\mathcal{S}$ connects all the nodes.

The set $\mathcal{S}$ now forms a spanning tree of $\mathcal{G}$, and the feasible solution x form which $\mathcal{S}$ is derived is referred to as a spanning tree solution. ■

Spanning Tree Triangularity:

The rows and columns of the node-arc incidence matrix of any spanning tree can be re-arranged to be lower triangular.

Proof:

Represent any spanning tree $\mathcal{S}$ by a node-arc incidence matrix i.e. a matrix with a row for every node and a column for each arc. Each column representing arc (i, j) has a $+1$ in row i, a -1 in row j, and 0 elsewhere.

Each spanning tree must have at least one leaf node, i.e. a node incident to only one arc in $\mathcal{S}$. The row corresponding to this node has only one non-zero entry, ($+1$ or -1) and the matrix can be permuted until this entry is in the top left hand corner.

Remove this node and its incident arc from $\mathcal{S}$ and from the node-arc incidence matrix. We again have a spanning tree. The procedure is repeated for the first $n - 1$ rows, the

resulting matrix being

$$L = \begin{pmatrix} \pm 1 & 0 & \dots & 0 \\ p_1 & \pm 1 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ p_{n-1} & q_{n-1} & \dots & \pm 1 \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

The final row is redundant as it equals -1 times the sum of all the other rows, and can thus be omitted, leaving a lower triangular matrix with ± 1 diagonal entries. ■

To illustrate the spanning tree and its triangularisation, we consider the following simple distribution network, based on the map of Ireland in Figure 1a. Dublin and Belfast are supply nodes, while Cork, Galway, Limerick and Waterford are demand nodes. Figure 1b shows a possible solution to the problem, represented as a spanning tree, rooted at an artificial node, r. The node-arc incidence matrix corresponding to this tree is shown in Figure 2a, alongside its triangularised equivalent (Figure 2b).

Town	Node	Supply
Dublin	1	12
Belfast	2	8
Cork	3	-6
Galway	4	-8
Limerick	5	-4
Waterford	6	-2

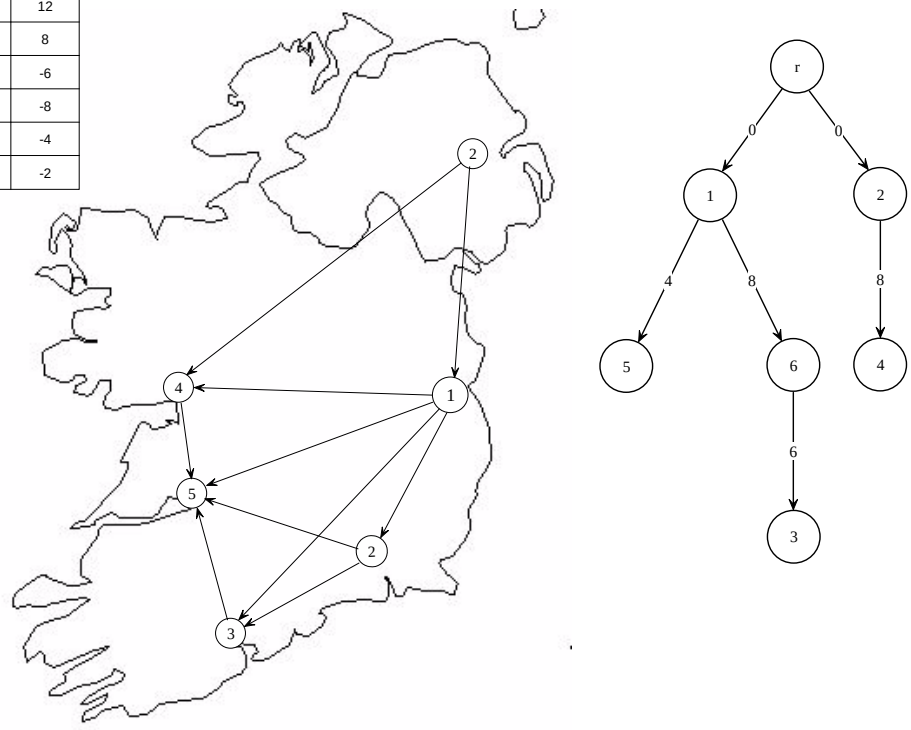

Figure 1. (a) Ireland Distribution Network. (b) Spanning Tree.

Integrality of Optimum Solution:

If the objective function of the MCF problem

$$\textbf{minimise } \{cx : Nx = b, l \leq x \leq u\}$$

is bounded from below on the feasible region, the problem has a feasible solution, and the vectors b,l and u are integer, then the problem has at least one integer optimum solution.

Proof:

Triangulate the constraint matrix N so that the first $n \times n$ portion corresponds to L above. Let the rest correspond to M, thus $N = [L, M]$. Partition the flow vector x so the first $n-1$ elements of x correspond to a the spanning tree. Let this portion be $x^{(1)}$ and the remainder be $x^{(2)}$.

$$\text{Thus } x = \begin{pmatrix} x^{(1)} \\ x^{(2)} \end{pmatrix}$$

$$Nx = (L, M) \begin{pmatrix} x^{(1)} \\ x^{(2)} \end{pmatrix}$$

$$= Lx^{(1)} + Mx^{(2)} = b$$

$$\text{Therefore} \qquad Lx^{(1)} = b - Mx^{(2)}$$

All the elements of b are integers. All the elements of $x^{(2)}$ equal upper or lower bounds, both of which are integers. Triangulation involves re-arranging rows and columns of the node-arc incidence matrix, therefore all the elements of L and M are $0, +1, -1$ and all the diagonal elements of L are non-zero. Thus by forward substitution we can solve for all $x_i \in x^{(1)}$ with all $x_i \in x^{(1)}$ being integral. ■

	x_{16}	x_{15}	x_{24}	x_{63}	x_{1r}	x_{2r}
1	1	1	0	0	1	0
2	0	0	1	0	0	1
3	0	0	0	−1	0	0
4	0	0	−1	0	0	0
5	0	−1	0	0	0	0
6	−1	0	0	1	0	0
r	0	0	0	0	−1	−1

Figure 2a : Node-Arc Incidence Matrix

	x_{16}	x_{15}	x_{24}	x_{63}	x_{1r}	x_{2r}
3	−1	0	0	0	0	0
6	1	−1	0	0	0	0
5	0	0	−1	0	0	0
4	0	0	0	−1	0	0
1	0	1	1	0	1	0
2	0	0	0	1	0	1
r	0	0	0	0	−1	−1

Figure 2b : Triangularised Matrix

Extreme Point Property:

For the MCF problem, every cycle free solution is an *extreme point*, and every extreme point is a cycle free solution. An extreme point solution is one which cannot be expressed as a weighted combination of two other feasible solutions i.e. as $x = \alpha y + (1-\alpha)z$ for $0 < \alpha < 1$.

Proof:

If x is not a cycle free solution we can generate two other feasible solutions y and z by sending a flow of $+\theta$ and $-\theta$ respectively around the cycle so that $x = \frac{1}{2}y + \frac{1}{2}z$, and therefore x is not an extreme point.

If x is not an extreme point, then $x = \alpha y + (1-\alpha)z$ with $0 < \alpha < 1$. Let x',y' and z' be the components of these vectors for which y and z differ. Thus

$$\begin{aligned} & l_{ij} \leq y_{ij} < x_{ij} < z_{ij} \leq u_{ij} \\ \text{or} \quad & l_{ij} \leq z_{ij} < x_{ij} < y_{ij} \leq u_{ij} \end{aligned}$$

for those arcs (i, j) for which y and z differ. Let N' denote the submatrix of N whose columns correspond to these arcs (i, j). Then since

$$\begin{aligned} Ny = Nz &= b \\ \text{so } N'y' &= N'z' \\ \text{i.e. } N'(z'-y') &= 0 \end{aligned}$$

Therefore treating $(z' - y')$ as a flow vector on these arcs, the flow out of each node is equal to the flow into each node. Then

$$\begin{aligned} x &= \alpha y + (1-\alpha)z \\ &= z - \alpha(z-y) \end{aligned}$$

Therefore x contains an undirected cycle composed entirely of free arcs by the definition of x_{ij} above. Hence if x is not an extreme point, it is not a cycle free solution. Hence any cycle free solution must be an extreme point and any extreme point must be a cycle free solution. ■

Basis Property:

Every spanning tree solution to the MCF is a basic solution and vice versa. A basic solution to a bounded LP problem with s variables and t constraints will have t basic variables and $(s - t)$ non-basic variables i.e. set to either their upper or lower bounds.

Proof:

A basic solution to the MCF problem has $s = m$ variables, one for each arc, and $t = n-1$ constraints, one for each node excluding the final redundant constraint as outlined above. It will therefore have $t = n - 1$ basic and $s - t = m - (n - 1)$ non-basic variables. This corresponds exactly to the spanning tree soluton which has $n - 1$ arcs, leaving $m - (n - 1)$ arcs not in the tree. These non-tree arcs are at either their upper or lower bounds.

Therefore, basic solutions, spanning tree solutions and by implication, extreme point solutions correspond. Hence every spanning tree solution to the MCF is a basic solution and vice versa. ■

3.2 MCF Problem Transformation

In order to simplify the operation of NSM some network transformations are often required. In particular these involve the removal of all non-zero lower flow bounds, and the additional of artificial arcs and an artificial root node to obtain a basic feasible solution.

3.2.1 Removing Non-Zero Lower Bounds

The standard NSM is based on a problem in which $l_{ij} = 0$ for all arcs $(i, j) \in \mathcal{A}$. In order to accomodate problems which include lower-bounded arcs, the following transformation is applied. For $(i, j) \in \mathcal{A}$ with $l_{ij} \neq 0$, we replace:

$$\begin{aligned} l_{ij} &\text{ with } 0 \\ u_{ij} &\text{ with } u_{ij} - l_{ij} \\ b(i) &\text{ with } b(i) - l_{ij} \\ b(j) &\text{ with } b(j) + l_{ij} \\ x_{ij} &\text{ with } x'_{ij} = x_{ij} - l_{ij} \end{aligned}$$

After problem solution the flow values for the original problem are obtained as $x_{ij} = x'_{ij} + l_{ij}$ for all arcs $(i, j) \in \mathcal{A}$.

3.2.2 Artificial Starting Solution

An artificial *root* node and artificial arcs are added in order to simplify the task of finding an initial BFS. The *root* we call r and introduce an additional arc for each node i of the original problem as follows:

$$\begin{aligned} (i, r) &\text{ if } b(i) \geq 0 \\ (r, i) &\text{ if } b(i) < 0 \end{aligned}$$

with $c_{ir}, c_{ri} = C$ and $u_{ir}, u_{ri} = \infty$ where C is a large cost.

3.3 Optimality Condition – The Dual Problem

As stated above, a feasible flow x is an optimum flow if and only if it admits no negative cost augmenting cycles. The *dual* of the MCF problem and the *Complementary Slackness Theorem* can be used to derive optimality conditions equivalent to those previously stated.

3.3.1 MCF Problem

The primal problem is:

$$\text{minimise} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{3.1}$$

$$\text{subject to} \sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} x_{ji} = b(i), \qquad \text{for all } i \in \mathcal{N} \tag{3.2}$$

$$0 \leq x_{ij} \leq u_{ij}, \qquad \text{for all } (i, j) \in \mathcal{A} \tag{3.3}$$

3.3.2 Dual Problem

Associate the dual variable $\pi(i)$ with the flow balance constraint for node i in (3.2) above, and δ_{ij} with the upper bound constraint of arc (i, j) in (3.3).

The Dual Problem is then:

$$\text{maximise} \sum_{i\in\mathcal{N}} b(i)\pi(i) \tag{3.4}$$

$$\text{subject to} \quad \pi(i) - \pi(j) - \delta_{ij} \leq c_{ij} \qquad \text{for all } (i,j) \in \mathcal{A} \tag{3.5}$$
$$\delta_{ij} \geq 0, \qquad \text{for all } (i,j) \in \mathcal{A} \tag{3.6}$$

3.3.3 Complementary Slackness Theorem

For a primal constraint and the corresponding dual variable (or a dual constraint and the corresponding primal variable) the following is true with respect to an optimal solution:

(i) If a constraint is oversatisfied (i.e. if the corresponding slack variable is greater than zero), then the corresponding dual variable equals zero.

(ii) If a dual variable is positive, then the corresponding primal constraint is exaclty satisfied.

The complementary slackness conditions for the MCF problem are:

$$x_{ij} > 0 \Rightarrow \pi(i) - \pi(j) - \delta_{ij} = c_{ij}$$
$$\delta_{ij} < 0 \Rightarrow x_{ij} = u_{ij}$$

which translate into the following optimality conditions:

$$x_{ij} = 0 \Rightarrow \pi(i) - \pi(j) \leq c_{ij}$$
$$0 \leq x_{ij} \leq u_{ij} \Rightarrow \pi(i) - \pi(j) = c_{ij}$$
$$x_{ij} = u_{ij} \Rightarrow \pi(i) - \pi(j) \geq c_{ij}$$

The reduced cost of arc (i,j), $\bar{c}_{ij}$ is defined as:

$$\bar{c}_{ij} = c_{ij} - \pi(i) + \pi(j)$$

giving the optimality conditions:

(i) x is feasible.

(ii) If $\bar{c}_{ij} > 0$ then $x_{ij} = 0$.

(iii) If $\bar{c}_{ij} = 0$ then $0 \leq x_{ij} \leq u_{ij}$.

(iv) If $\bar{c}_{ij} < 0$ then $x_{ij} = u_{ij}$.

We now show that these optimality conditions are equivalent to those previously given. Let x, π respectively be flows and node potentials satisfying the dual optimality conditions. Let W be any cycle in the network, along which we can augment flow.

If the direction of flow in any arc is opposite to the direction of flow augmentation in the cycle W, then for the purposes of this exposition, replace that arc (i,j) with the reverse arc (j,i). In this case $x_{ij} > 0$ and thus $\bar{c} \leq 0$. Also, replace $\bar{c}_{ij}$ with $c^*_{ij} = -\bar{c}_{ij}$.

If the flow direction in an arc is consistent with the direction of cycle flow augmentation, then $\bar{c}_{ij} \geq 0$ and let $c^*_{ij} = \bar{c}_{ij}$.

Now for all arcs, $(i,j) \in W$, $c^*_{ij} \geq 0$, so

$$\sum_{(i,j)\in W} c^*_{ij} \geq 0$$

$$\text{But} \sum_{(i,j)\in W} c^*_{ij} = \sum_{(i,j)\in W} c_{ij} + \sum_{(i,j)\in W} (-\pi(i) + \pi(j))$$
$$= \sum_{(i,j)\in W} c_{ij}$$

Hence the network satisfying the dual optimality conditons is free of negative cost augmenting cycles. To show the converse, we again transform the solution. Replace all arcs (i, j) with $x_{ij} = u_{ij}$, with arcs (j, i) where $\bar{c}_{ji} = -\bar{c}_{ij}$. The optimality conditions become

$$\begin{aligned} \bar{c}_{ij} &= 0 \qquad \text{if } 0 \le x_{ij} \le u_{ij} \\ \bar{c}_{ij} &> 0 \qquad \text{otherwise} \end{aligned}$$

A basis structure is defined by partitioning the set of arcs, $\mathcal{A}$, into three disjoint sets, as follows:

$\mathcal{B}$: the set of basic arcs, which form a spanning tree of the network,
$\mathcal{L}$: the set of non-basic arcs at their lower bounds,
$\mathcal{U}$: the set of non-basic arcs at their upper bounds,

We suppose the network contains no negative cost augmenting cycles. Given a basis structure $(\mathcal{B}, \mathcal{L}, \mathcal{U})$ we compute node potentials as follows, fulfilling the condition that $\bar{c}_{ij} = 0$ for all basic arcs.

(1) Set $\pi(root) = 0$.
(2) Find an arc $(i, j) \in \mathcal{B}$ for which one node potential is known and the other not. If $\pi(i)$ is known, then set $\pi(j) = \pi(i) - c_{ij}$. If $\pi(j)$ is known, then set $\pi(i) = \pi(j) + c_{ij}$.
(3) Repeat (2) until all π are known.

All π-values will be found since there are $n-1$ arcs in $\mathcal{B}$, and they form a spanning tree. Suppose there is some arc $(l, k) \in \mathcal{L}$ or $\mathcal{U}$ for which $\bar{c}_{ij} < 0$. We can add basic arcs to form a cycle W with arc (l, k) such that

$$\begin{aligned} \sum_{(i,j)\in W} c_{ij} &= \sum_{(i,j)\in W} \bar{c}_{ij} - \sum_{(i,j)\in W} (-\pi(i) + \pi(j)) \\ &= \sum_{(i,j)\in W} \bar{c}_{ij} \\ &= \bar{c}_{lk} + \sum_{(i,j)\in W\setminus(l,k)} \bar{c}_{ij} \\ &= \bar{c}_{lk} < 0 \end{aligned}$$

$$\text{Hence } \sum_{(i,j)\in W} \bar{c}_{ij} < 0$$

and thus W is a negative cost augmenting cycle. (It is an augmenting cycle by virtue of the definition of all arcs $(i, j) \in \mathcal{B}$, and the transformation of arc (l, k)). Hence by contradiction, if the network contains no negative cost augmenting cycles, then it fulfills the dual (reduced cost) optimality conditions, so the two sets of optimality conditons are equivalent.

3.4 Statement of the NSM Algorithm

The Network Simplex Method for solving the Minimum Cost Flow Problem is an implementation of the bounded variable simplex method, in which all operations are performed directly on the graph of the problem. The special structure of the MCF problem allows NSM to achieve great improvements in efficiency over the simplex method.

A primal Basic Feasible Solution (BFS) is maintained throughout the algorithm. This BFS is defined by partitioning the arcs of the network into three disjoint sets:

$\mathcal{B}$: the set of basic arcs, which form a spanning tree of the underlying graph of the problem

$\mathcal{L}$: the set of non-basic arcs at their lower bounds

$\mathcal{U}$: the set of non-basic arcs at their upper bounds

Thus if $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ describes the graph of the problem, then $\mathcal{A} = \mathcal{B} \cup \mathcal{L} \cup \mathcal{U}$, and $(\mathcal{B}, \mathcal{L}, \mathcal{U})$ describes a BFS. In a problem in which all lower flow bounds are zero, a basic solution will be feasible if the flow, x_{ij}, in each arc $(i, j) \in A$, is as follows:

$$x_{ij} = \begin{cases} 0 & \text{if } (i,j) \in \mathcal{L} \\ u_{ij} & \text{if } (i,j) \in \mathcal{U} \\ 0 < x_{ij} < u_{ij} & \text{defined in basis otherwise} \end{cases}$$

NSM iterates towards the optimal solution by exchanging basic with non-basic arcs and adjusting the flows accordingly. The algorithm terminates when a basic feasible solution is reached which satisfies the reduced cost optimality conditions outlined in section 3.3.3 above. The correspondence between NSM and the LP Simplex method can be seen by comparing simple algorithmic statements.

NSM Algorithm	LP Simplex Algorithm
0. Construct Initial Basis Tree	0. Generate Initial BFS
1. Choose Entering Arc	1. Choose Entering Variable
2. Determine Leaving Arc	2. Determine Leaving Variable
3. Construct New Basis Tree	3. Move to New Basic Solution
Steps 1 – 3 are repeated	until optimality is achieved.

Starting Data

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be the underlying directed graph of a network problem, such that $|\mathcal{N}| = n$ and $|\mathcal{A}| = m$. We assume the nodes are numbered $1, 2, ..., n$ in an arbitrary, but fixed order. The basic input data for solution by NSM are as follows. For each arc $(i, j) \in \mathcal{A}$, we must know

c_{ij}: the cost per unit flow in the arc

l_{ij}: the lower bound on flow in the arc

u_{ij}: the upper bound on flow in the arc

For each node $i \in \mathcal{N}$, we need the value $b(i)$ which represents the exogenous flow at that node, defined as:

$$b(i) \text{ is } \begin{cases} < 0 & \text{if } i \in \mathcal{N} \text{ is a demand node} \\ = 0 & \text{if } i \in \mathcal{N} \text{ is a transshipment node} \\ > 0 & \text{if } i \in \mathcal{N} \text{ is a supply node} \end{cases}$$

Step 0: Initialisation

Two transformations are performed on the problem. First, all non-zero lower bounds are removed so that $l_{ij} = 0$ for all $(i, j) \in \mathcal{A}$. Second, an artificial node, denoted r, and artificial

arcs joining this root to all nodes $i \in \mathcal{N}$ are added. Details of both these transformations were described earlier in section 3.2.

An all-artificial initial basic feasible solution is generated by assigning flows to artificial arcs as follows:

$$\text{if } b(i) < 0 \text{ set } x_{ri} \leftarrow -b(i)$$
$$b(i) \geq 0 \text{ set } x_{ir} \leftarrow b(i)$$

for all nodes $i \in \mathcal{N}$. The set $\mathcal{L}$ is composed of the remaining arcs, i.e. the arcs of the original problem, and the set $\mathcal{U}$ is empty.

The initial node potentials are calculated using the dual feasibility conditions outlined in section 3.3.3 above

As we have $n-1$ arcs in the basis tree, and n nodes for which we must calculate potentials, the potential of one node is set arbitrarily. Thus for definiteness, we set $\pi(r) = 0$.

Step 1: Select an Entering Arc

An arc may be admitted to the basis to improve the objective value of the solution if it violates the dual feasibility conditions. Thus an arc $(i, j) \in \mathcal{A}$, is said to be *admissible* if

$$\bar{c}_{ij} < 0 \text{ and } x_{ij} = 0$$
$$\text{or } \bar{c}_{ij} > 0 \text{ and } x_{ij} = u_{ij}$$

As previously shown, such arcs when form negative augmenting cycles with the arcs of the basis. A range of methods, called *pricing strategies*, may be used to select the entering arc at each iteration. The chosen arc is designated (i^*, j^*). If no admissible arc exists, then the current solution is optimal, and the algorithm proceeds to *Step 4*. Otherwise, we perform *Step 2*.

Step 2: Determine the Leaving Arc

The entering arc, (i^*, j^*), forms a unique cycle, $\mathcal{W}$, with the arcs of the basis. In order to eliminate this cycle, one of its arcs must leave the basis. By augmenting flow in a negative cost augmenting cycle, we improve the objective value of the solution. The cycle is eliminated when we have augmented flow by a sufficient amount to force the flow in one or more arcs of the cycle to their upper or lower bounds.

The first task in determining the leaving arc is the identification of all arcs of the cycle. We let *join* be the common predecessor of both endpoints of the entering arc, i^* and j^*. Then all arcs on the paths from i^* and j^* to join are in the cycle, $\mathcal{W}$. We then find the maximum allowable flow increase or decrease, δ_{ij} in each of these arcs, and define

$$\delta = \min\{\delta_{ij} \text{ for all } (i, j) \in \mathcal{W}\}$$

The leaving arc is selected from those arcs (i, j) for which $\delta_{ij} = \delta$, so that a *strongly feasible basis* is maintained (see section 3.5 below). Now that both entering and leaving arcs have been determined, we perform *Step 3*.

Step 3: Pivot

In this step, we exchange the entering arc for the leaving arc, and construct the new BFS, $(\mathcal{B}', \mathcal{L}', \mathcal{U}')$ adjusting flows and node potentials accordingly. Flow change is the first task. The flow in every arc of the cycle $\mathcal{W}$ is increased or decreased by the amount δ, depending on the orientation of the arc in relation to the orientation of the cycle. Generally, we must exchange a basic with a non-basic arc. For example, if (i^*, j^*) was previously at its lower bound, and the leaving arc, (p, q), is to become non-basic at its upper bound, then

$$\begin{aligned} B' &= B \backslash \{(p,q)\} \cup \{(i^*, j^*)\} \\ L' &= L \backslash \{(i^*, j^*)\} \\ U' &= U \cup \{(p,q)\} \end{aligned}$$

If the entering arc is also the leaving arc, then the flow in (i^*, j^*) switches from its lower bound to its upper bound, or vice versa. Thus

$$\begin{aligned} B' &= B \\ L' &= L \backslash \{(i^*, j^*)\} \\ U' &= U \cup \{(i^*, j^*)\} \end{aligned}$$

The removal of the leaving arc divides the original basis tree into two subtrees, T_1, T_2, such that T_1 contains the root. The node potentials of all nodes in T_2 must be updated. This completes the construction of the new BFS, and so we return to *Step 1.*

Step 4: Reverse Transformation

When the optimal solution is found, it is described by the current BFS, $(\mathcal{B}, \mathcal{L}, \mathcal{U})$, in terms of a problem with all zero lower bounds. The final step is to reverse the transformation which removed the original lower bounds. If the original problem is feasible and balanced (i.e. if total supply equals total demand), then all artificial arcs will be non-basic at their lower bounds. Thus for each arc of the original problem, we set

$$x_{ij} \leftarrow x_{ij} + l_{ij}, \qquad \text{for all } (i,j) \in \mathcal{A}$$

The algorithm now terminates having found the optimal solution.

3.5 Strongly Feasible Bases

A strongly feasible basis is one in which a positive amount of flow can be sent from any node to the root, r, without violating any of the flow bound constraints. The artificial starting solution described above generates a strongly feasible basis, for consider each node, $i \in \mathcal{N}$:

If $b(i) < 0$, then arc (r, i) exists with positive flow. Reducing the flow in arc (r, i) is equivalent to sending a positive amount of flow from i to the root.

If, on the other hand, $b(i) \geq 0$, then arc (i, r) exists with positive or zero flow. This flow may always be increased, since there is an infinte upper flow bound on this arc, i.e. $u_{ir} = \infty$.

At each iteration of the NSM algorithm, the entering arc, (i^*, j^*), forms a unique cycle, W, with the arcs of the basis tree, and flow is augmented in that cycle by the maximum

algorithm *Network Simplex Method*
begin
Transform lower bounds to zero
Generate Initial BFS, Tree $\mathcal{T}$
$(i^*, j*) \leftarrow$ *Entering Arc* $\in \mathcal{L} \cup \mathcal{U}$
while $(i^*, j^*) \neq$ *NULL* **do**
Find Cycle $\mathcal{W} \in \mathcal{T} \cup (i^*, j^*)$
$\delta \leftarrow$ *Flow Change*
$(p, q) \leftarrow$ *Leaving Arc* $\in \mathcal{W}$
Update Flow in W by δ
Update BFS, Tree T
Update node potentials
$(i^*, j^*) \leftarrow$ *Entering Arc* $\in \mathcal{L} \cup \mathcal{U}$
end while
Reverse Transformation
end alg

amount possible, δ. We let δ_{ij} be the maximum allowable flow increase or decrease in arc $(i, j) \in W$, and refer to any arc for which $\delta_{ij} = \delta$ as a *blocking arc.* If $(i^*, j^*) \in \mathcal{L}$, then the orientation of the cycle is from i^* to j^*. Conversely, if $(i^*, j^*) \in \mathcal{U}$, then the orientation of the cycle is from j^* to i^*. The cycle, W, consists of all arcs on the paths from i^* and j^* to *join*, their first common predecessor in the basis tree. In Figure 3, arc (i^*, j^*), which is currently at its lower bound, has been chosen to enter the basis. Thus if arcs (b, c) and (j^*, d) are both blocking arcs, then we would select (j^*, d) as the leaving arc, in order to maintain a strongly feasible basis.

The *strongly feasible basis technique* was proposed by Cunningham (1976) to reduce the number of degenerate pivots, i.e. pivots in which a flow change of zero occurs, and to prevent cycling. The technique specifies that when there is more than one blocking arc, the *last* blocking arc encountered in traversing the W, in the direction of its orientation, starting at *join*, should be selected as the leaving arc. By consistently applying this technique, a strongly feasible basis is maintained throughout the operation of the algorithm, and the number of iterations required to obtain the optimal solution is guaranteed to be finite.

Theorem:

A strongly feasible basis is preserved by applying the technique stated above.

Proof:

Let (p, q) be the leaving arc chosen by the technique as stated. Let W_1 be that portion of W from *join* to the first node of (p, q), in the direction of the orientation of W. Let W_2 be the remainder of the cycle, excluding arc (p, q). We define the orientation of the paths W_1 and W_2 consistent with the orientation of W.

Since (p, q) is the last blocking arc, no arc in W_2 is blocking, and therefore every node in W_2 can send positive flow to the root node via the arcs of W_2 in the direction of its orientation to *join*, and from there to the root.

We now consider W_1. If a positive amount of flow is augmented in W, i.e. if the pivot is non-degenerate, in the direction of its orientation, then every node in W_1 can send a positive

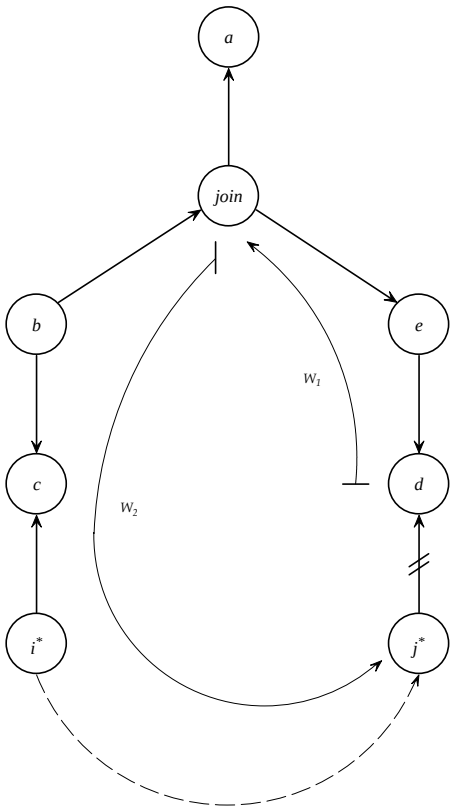

Figure 3. Leaving-Arc Selection.

flow to the root via W_1 in a direction opposite to its orientation.

If the pivot is degenerate, then zero flow is augmented. Then W_1 must be in the segment from i^* to $join$ if $(i^*, j^*) \in \mathcal{L}$, or in the segment j^* to $join$ if $(i^*, j^*) \in \mathcal{U}$, because, by the strongly feasible basis technique, all arcs in the opposite segments can augment positive flow to the root (the opposite segments in each case point to the root). Therefore, since every node in W_1 could send positive flow to the root *before* the pivot was performed, and the pivot does not change any flows, every node in W_1 can also send positive flow to the root *after* the pivot has been performed.

Hence, by application of the technique outlined above, a strongly feasible basis is maintained at each pivot. ■

Theorem:

The strongly feasible basis technique gaurantees that NSM will obtain the optimal solution in a finite number of pivots.

Proof:

The value of the objective function is bounded above and below by mCU and 0 respectively. In a non-degenerate pivot, the value of the objective function strictly decreases. Therefore, the number of non-degenerate pivots is guaranteed to be finite.

In a degenerate pivot, we must look at the change in node potentials.

Case 1:

$(i^*, j^*) \in \mathcal{L}$. By the strongly feasible basis property, the leaving arc (p, q) is on the path from i^* to $join$. Let T_2 be the subtree consisting of all nodes of W_2 on the path from i^* to the first node of (p, q), and the subtrees of these nodes. The potentials of nodes in T_2 only, are changed by the pivot.

$$\begin{aligned} \text{For basic arcs, } \bar{c}_{ij} &= c_{ij} - \pi(i) + \pi(j) = 0 \\ \text{therefore } \pi(i) - \pi(j) &= c_{ij} \\ \text{But } \bar{c}_{i^*j^*} &= c_{i^*j^*} - \pi(i^*) + \pi(j^*) < 0 \\ \text{and } \left(\pi(i^*) + \bar{c}_{i^*j^*}\right) - \pi(j^*) &= c_{i^*j^*} \end{aligned}$$

Hence as $\bar{c}_{ij} < 0$, $\pi(i^*)$ falls by the amount $\bar{c}_{i^*j^*}$. The relationship $\pi(i) - \pi(j) = c_{ij}$ for all basic arcs (i, j) implies that all potentials in T_2 fall by this constant amount.

Case 2:

$(i^*, j^*) \in U$. Thus the leaving arc (p, q) is on the path form j^* to $join$. Let T_2 be the subtree consisting of all nodes of W_2 on the path from j^* to the first node of (p, q), and the subtrees of these nodes. The potentials of nodes in T_2 only, are changed by the pivot. In this case

$$\begin{aligned} \bar{c}_{i^*j^*} &= c_{i^*j^*} - \pi(i^*) + \pi(j^*) > 0 \\ \text{and } \left(\pi(i^*) + \bar{c}_{i^*j^*}\right) - \pi(j^*) &= c_{i^*j^*} \end{aligned}$$

This implies that $\pi(j^*)$ falls by the positive amount $\bar{c}_{i^*j^*}$ and, as before, so do the potentials of all nodes in T_2.

Thus , during a degenerate pivot, the sum of all node potentials strictly decreases. As a lower bound of $-n^2C$ can be put on the sum of potentials (i.e. n nodes, each with a minimum potential of $-nC$), the NSM using the strongly feasible basis technique obtains the optimal solution in a finite number of iterations. ■

3.6 Complexity Analysis of NSM

An upper bound on the value of the objective function of a Minimum Cost Flow problem is given by mCU, i.e. the maximum flow, U, in each of the m arcs, at maximum cost, C.

Each non-degenerate pivot strictly decreases the value of the objective function. Therefore, mCU is a bound on the number of non-degenerate pivots.

Degenerate pivots may also occur. Since a bound on a node potential is nC, so n^2C represents a bound on the sum of node potentials. A degenerate pivot strictly decreases the sum of node potentials when the strongly feasible bases technique is applied. Therefore, n^2C is a bound on the number of sequential degenerate pivots. A series of degenerate pivots may occur between each pair of non-degenerate pivots, and thus a bound on the total number of iterations is:

$$mCUn^2C = mn^2C^2U$$

To find the entering arc is an $O(m)$ operation. Finding the cycle, amount of flow change, and leaving arc, as well as the updating of all indices are O(n) operations. Hence the complexity of each pivot is $O(m + n)$.

Therefore the complexity of the algorithm is

$$O\big((m+n)mn^2C^2U\big)$$

We now develop a tighter bound. Let there be k non-degenerate pivots. Let c_k be the improvement in the objective function in the k^{th} such pivot. Let a_k denote the number of nodes whose potentials have been changed during this pivot. Thus c_k is an upper bound on a_k.

Thus

$$a_1 + a_2 + \ldots + a_k \leq c_1 + c_2 + \ldots + c_k \leq mCU$$

$$\text{i.e } \sum_{i=1}^{k} a_i = k\bar{a} \leq mCU \text{ where } \bar{a} = \sum_{i=1}^{k} a_i/k$$

$$\text{Hence } k \leq \frac{mCU}{\bar{a}} \text{ for } k \text{ non} - \text{degenerate pivots}$$

Asymtotically, after the k^{th} non-degenerate pivot, we can have up to $a_k nC$ degenerate pivots. Therefore, the total number of pivots is bounded by

$$\begin{aligned} k + \sum_{i=1}^{k} a_i nC &= k + nC \sum_{i=1}^{k} a_i \\ &= k + nCk\bar{a} \\ &\leq \frac{mCU}{\bar{a}} + nmC^2U \end{aligned}$$

As a pivot is $O(m+n)$ this gives the complexity of the NSM algorithm as

$$O\big((m+n)mnC^2U\big)$$

Chapter 4

Implementation Details

The computer science literature offers a number of different storage schemes which might be considered in an implementation of the Network Simplex Method. The following discussion outlines a few of these, and explains the structures which we have chosen.

4.1 Input Format

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be the underlying directed graph of a network problem, such that $|\mathcal{N}| = n$ and $|\mathcal{A}| = m$. We assume the nodes are numbered $1, 2, ..., n$ in an arbitrary, but fixed order. For each node $i \in \mathcal{N}$, we read in the value $b(i)$ which represents the exogenous flow at that node, defined as:

$$b(i) \text{ is } \begin{cases} < 0 & \text{if } i \in \mathcal{N} \text{ is a demand node} \\ = 0 & \text{if } i \in \mathcal{N} \text{ is a transshipment node} \\ > 0 & \text{if } i \in \mathcal{N} \text{ is a supply node} \end{cases}$$

The set of arcs A is fully described by five lists, again arbitrarily ordered. For each arc $(i, j) \in \mathcal{A}$, we read in the $Inode$, $Jnode$, $Cost$ per unit flow, lower bound, L, and upper bound, U.

4.2 Internal Storage

Two main factors influence the design of data structures for efficient large scale problem solving. One is that the structures should facilitate the speedy execution of the steps of the algorithm, and that the data used can be efficiently updated. The second is to limit the amount of data which is explicitly stored, avoiding duplication of data. Both of these factors were taken into account in our implementation of NSM, though in cases of conflict, we generally favoured speed over space.

Two basic Abstract Data Types (ADT) are used in the implementation of NSM. The first we call ADT *Arc Set*, of which B, L and U are instances. The following are the operations associated with ADT Arc Set:

(i) $Add(k, \mathcal{A})$ – adds the arc with index k to the Arc Set $\mathcal{A}$.
(ii) $Delete(k, \mathcal{A})$ – deletes the arc with index k from the Arc Set $\mathcal{A}$.
(iii) $Access(k)$, $Change(k)$ – allow the attributes of an arc, k, to be examined and altered, respectively.

The second is ADT *Basis Tree*. This maintains the current basic feasible solution to the problem, in the form of a rooted tree, which is updated at each iteration of the algorithm. The following are the operations associated with ADT Basis Tree:

(i) $Access(u)$, $Change(u)$ – allow the attributes of a node, u, to be examined and altered, respectively.
(ii) $Mapping(u) : \mathcal{T} \rightarrow \mathcal{A}$ – returns index of the arc joining node $u \in \mathcal{T}$ to its parent.
(iv) $Traverse(u)$ – visits all nodes on the subtree rooted at u.
(v) $NCA(u, v)$ – returns the Nearest Common Ancestor of two nodes, u and v, in the Basis Tree.

(vi) *AddChild*(u, v) – adds node u to the subtree of v, such that v becomes the parent of u. *DeleteChild*(u, v) – deletes node u from the subtree of v, where the parent of u is v.

The application of the operations associated with each ADT will become clear in the discussion of our implementation.

4.2.1 Arc Set Storage Schemes

Beginning with the input format outlined above, two additional pieces of information about each arc are needed throughout the solution. The first is the *Bound* on the flow in the arc, as adjusted by the transformation of the original problem to one with all zero lower flow bounds. In addition, we store the current *State* of each arc k as follows:

$$State(k) = \begin{cases} 1 & \text{if arc } k \in \mathcal{L} \\ 0 & \text{if arc } k \in \mathcal{B} \\ -1 & \text{if arc } k \in \mathcal{U} \end{cases}$$

where $\mathcal{B}$ is the set of basic arcs, and $\mathcal{L}$,$\mathcal{U}$ are the sets of non-basic arcs at their lower and upper bounds respectively.

We now examine three storage schemes which might be used in implementing the ADT Arc Set. These are the *Dense* scheme, the *Forward Star* scheme, and our preferred *Arc-List* scheme.

Dense Arc Set Storage

A two-dimensional $n \times n$ array is used to store each piece of Arc Set information. The array is indexed with a row and a column for each node of the problem, so that for example, the cell in the i^{th} row and j^{th} column of the *Cost* matrix would store the cost per unit flow along arc (i, j). This scheme offers random access to the all arcs and is therefore easily formed from the input data.

The overwhelming disadvantage of the scheme, however, is its size. While the Assignment Problem generally forms a dense network, most other MCF formulations are based on very sparse networks. For example, a planar graph (such as a road network) may have at most $m = 3n - 6$ arcs, which for a network with 50 nodes, gives a sparsity factor of less than 6%. (The sparsity factor is defined as the number of actual arcs, m, expressed as a percentage of the number of possible arcs, $n(n - 1)$).

Forward Star Arc Set Storage

Bradley, Brown and Graves (1977) use a form of sparse storage which supports their particular choice of pricing strategy. Three arrays of length m store the *Cost*, *Bound* and *Jnode* of each arc. A sign-bit on the *Bound* marks arcs out of the basis at their upper bound. The arcs are sorted so that all arcs with the same *Jnode* are stored in contiguous space. An array of length m is then used whose i^{th} entry is the location of the first arc with *Inode* i. This scheme is far better suited to sparse networks than the Dense scheme.

One apparent limitation is that pure random access to a particular arc is no longer possible. Instead, one must start with the *Inode* and search until the correct *Jnode* is found. On the other hand, this scheme, like the Dense one, does allow the sequential examination of all arcs emanating from a particular *Inode*. This, in fact, is the cornerstone of the authors 'interesting node' pricing mechanism.

A more significant drawback is the requirement that the arcs be sorted before solution. The text is ambiguous as to whether or not this sorting routine is included in the running time of the algorithm. Even if excluded on theoretical grounds, sorting a large number of arcs will have significant implications for practical applications.

Arc-List Storage

The Arc Set structure which we have implemented is similar to the Forward Star scheme, but uses an array of length m to store *both* the $Inode$ and $Jnode$ of each arc. Since network problems invariably have more arcs than nodes, this results in a slighlty larger overall structure than Forward Star, though still taking advantage of the sparsity of the network. We also chose to explicitly store the *State* of each arc to distinguish between basic arcs and non-basic arcs at their lower bounds. In the Arc-List scheme used by Grigoriadis (1983), a negative sign-bit on the *Bound* marks arcs out of the basis at their upper bound. There is nothing to distinguish a basic arc from a non-basic arc at its lower bound. Our scheme, on the other hand, allows the pricing mechanism to make this distinction, and thus check the state of *every* arc before calculating its reduced cost.

Since each arc has a unique index, multiple arcs between a pair of nodes may be included. A great deal of flexibility in pricing strategy is also possible by pure random access to all of the arcs of the problem. The candidate list devised by Bradley, Brown and Graves (1977) can also be replicated by sorting the arcs into the required order, while most other pricing strategies are based on an arbitrarily-ordered Arc-List.

4.2.2 Basis Tree Storage Schemes

At each iteration of the NSM algorithm, the set of basic arcs, $\mathcal{B}$, is described by a rooted spanning tree of the underlying network. For each node of the problem, we must know its depth and potential. For each basic arc, meanwhile, we store the flow (and direction of flow) along that arc and an the index of the arc in the Arc-List. We define the parent of a node as its immediate predecessor in the tree, and store the above details as in the following arrays:
Depth – the number of arcs in the path from a node to the root, the root of the tree has depth zero, and every other node has depth one greater than its parent.
Flow – the amount of flow in the basic arc joining the node to its parent, the flow is always positive.
Pi – the current simplex multiplier, or potential of the node, the root always has potential zero.
Arc – the index into the Arc Set Structure of the basic arc joining the node to its parent.
Dir – the direction of flow is defined as:

$$Dir(i) = \begin{cases} forward & \text{if the arc is from node } i \text{ to its parent} \\ reverse & \text{if the arc is from parent of } i \text{ to node } i \end{cases}$$

There are essentially two main schemes for Basis Tree storage. These we refer to as the *Parent-Leftmost Child-Right Sibling (PLR)* scheme and the *Parent-Thread-Depth (PTD)* scheme. A number of variations on PTD also exist and are discussed briefly, but their operation is essentially the same as that described below. The depth index is also used in PLR, primarily to support an efficient cycle-finding routine, but has an additional, more vital role to play in the PTD scheme.

In order to update the potentials and depths at each iteration, it is necessary to visit all the nodes on a particular subtree. This is achieved by a *preorder traversal* of the subtree, which ensures that the root of any subtree is visited before its successors. Knuth (1973) defines a preorder traversal as follows:

(a) Visit the root of tree;

(b) traverse the subtrees of the tree in preorder.

Parent-Leftmost Child-Right Sibling Scheme

This three-list structure was first used by Johnson (1966). Each node has a unique parent (the parent of the root is NULL), a pointer to one of its immediate successors, called the *Leftmost Child*, and a pointer to another node with the same parent, called the *Right Sibling*. The scheme may be implemented in the form of three node-length arrays. O'Connor (1980) describes an equivalent implementation for a fast primal transportation code using linked lists, based on dynamic memory allocation in the Pascal language.

A feature of this scheme is the ease with which the Basis Tree can be reconstructed after each pivot. This is based on two primitive operations – one which deletes a given node (or subtree) from the tree, and the other which appends a node (or subtree) to the tree. The preorder traversal may be performed directly on the tree using this structure. Aho, Hopcroft and Ullman (1983) provide the following recursive procedure to achieve this:

```
procedure Preorder
Visit performs a desired operation at each node
begin
  Visit(root)
  i ← LeftMostChild(root)
  while i ≠ NULL do
    Preorder(i)
    i ← RightSibling(i)
  end while
end proc
```

Parent-Thread-Depth Scheme

In contemporary NSM implementations, the use of another Basis Tree scheme requiring three node-length arrays, known as *Parent-Thread-Depth*, has become popular. It draws from three sources. Johnson (1966) suggested the parent, while Glover, Klingman and Stutz (1974), developed the thread. The depth index is due to Bradley and Brown (1975). Helgason and Kennington (1977) recommend PTD saying:

> We believe that the combination of the parent, thread and depth provides one of the most powerful three label data structures available for implementing both primal and dual, single and multi-commodity network algorithms.

Bradley, Brown and Graves (1977), Chvátal (1983), Grigoriadis (1986) and Ahuja, Magnanti and Orlin (1989) all describe the PTD scheme. As in PLR, the *Parent* array contains the index of the immediate predecessor of each node in the tree. The *Thread* describes a preorder traversal of the tree by defining $Thread(i)$ as the node following node i in the traversal. Knuth (1973) credits A. J. Perlis and C. Thornton with devising the threaded tree representation for binary trees, of which this scheme is an extension.

The *Depth* index enables us to identify all the successors of a given node by following the thread until a node is found whose depth is less than or equal to the depth of the starting node. For example, we would visit the successors of node 1 in Figure 4 in the order $5-6-3$. The thread of node 3 points to node 2. At this point, $Depth(2) = Depth(1) = 1$, so we stop.

Node i	1	2	3	4	5	6	r
Parent(i)	r	r	6	2	1	1	r
Thread(i)	5	4	2	r	6	3	1
Depth(i)	1	1	3	2	2	2	0

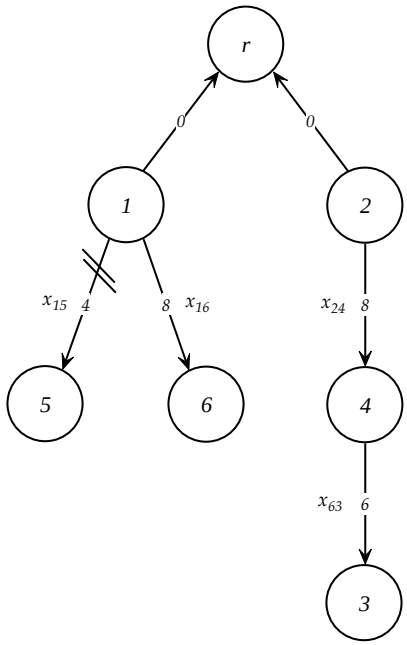

Figure 4. Parent-Thread-Depth Scheme for Basis Storage.

The usefulness of the PTD scheme would of course be defeated were it not possible to efficiently update the thread on reconstructing the tree. In fact, the design of a procedure to do this is far from straightforward, with a number of exceptional cases arising. The occurence of these cases must be checked at every iteration, and special routines performed as necessary. Despite this, the procedure to update the thread has order of complexity $O(n)$. Figure 4 shows a tree solution to the distribution problem, introduced in section 3.1.2, and the parent, thread and depth indices used to describe it.

Bradley, Brown and Graves (1977) also describe a variation on the PTD scheme above which replaces the *Depth* array with an array storing the number of successors of each node. Both the updating of the thread and efficient cycle finding are facilitated in an equivalent manner.

4.3 Initialisation

Once a network problem has been input, the first step is to find an Initial Basic Feasible Solution (BFS). In order to ensure the satisfaction of bound constraints, a problem with lower bounds greater than zero is transformed to one with all zero lower bounds. This is achieved for an individual arc by assigning a flow equal to the lower bound through the arc, adjusting its upper bound and the supply or demand at each of its adjacent nodes accordingly. The procedure *Transform* implements this operation over the entire arc-set.

```
procedure Transform
begin
  for k ← 1 to m do
    if L(k) < 0 then
      i ← Inode(k)
      j ← Jnode(k)
      Bound(k) ← U(k) − L(k)
      b(i) ← b(i) − L(k)
      b(j) ← b(j) + L(k)
    end if
    else Bound(k) ← U(k)
  end for
end proc
```

We now have an MCF formulation in which all lower bounds are zero. It is therefore assured that all flow bound constraints are satisfied by a zero flow through all of the arcs of this problem. In order to satisfy the supply and demand constraints, we append to the problem an *artificial node*, with index $n+1$ referred to as the *root*, and an *artificial arc* for each node of the original problem. Each artificial arc will have lower bound zero, upper bound infinity, and a high cost, C, per unit flow. For each node $i \in \mathcal{N}$, we add the arc (i, r) and assign flow $x_{ir} = b(i)$ if $b(i) > 0$, and arc (r, i) with flow $x_{ri} = -b(i)$ otherwise. This satisfies all supply and demand constraints. We mark the *state* of each artificial arc basic, and all others non-basic at their lower bounds (zero). This is the initial BFS for a problem with $n' = n + 1$ nodes and $m' = m + n$ arcs. The BFS has n $(= n' - 1)$ arcs and is fully connected, so it is clearly a spanning tree of the underlying graph, $\mathcal{G}' = (\mathcal{N}', \mathcal{A}')$.

Given the contrived structure of the initial, fully artificial BFS, the initialisation of *Parent, Depth, Thread, Dir* and *Pi* arrays is straightforward. The root is assigned the *NULL* parent and depth 0, while all other nodes have the root as their parent and depth 1. The direction of flow depends on $b(i)$, while the preorder traversal starts at the root and visits the nodes in numeric order (by setting $Thread(r) = 1$ and $Thread(i) = i + 1$ for $i = 1, 2, \ldots, n$). Finally, we determine the $Pi(i)$ values as C or $-C$ depending on the direction of the basic arc adjacent to each node i, since $Pi(r) = 0$.

4.4 Selecting the Entering Arc

Any non-basic arc may be admitted to the basis to improve the objective value of the solution if it forms a negative cost augmenting cycle with the arcs of the current basis. Given the node potentials, $\pi(i)$, for all $i \in \mathcal{N}$, the reduced cost, of arc $k = (i, j)$ is calculated using the formula

$$\bar{c}_{ij} = c_{ij} - \pi(i) + \pi(j)$$

The criterion for an admissible arc (i.e. one which will improve the objective value of the solution) may then be stated as follows:

$$(i,j) \text{ is admissible if } \begin{cases} \bar{c}_{ij} < 0 & \text{and } (i,j) \text{ is at its lower bound} \\ \bar{c}_{ij} > 0 & \text{and } (i,j) \text{ is at its upper bound} \end{cases}$$

```
procedure Artificial BFS
begin
  root ← n + 1
  for k ← 1 to m do
    State(k) ← 1
  Parent(root) ← NULL
  Thread(root) ← 1
  Flow(root) ← 0
  Supply(root) ← 0
  Depth(root) ← 0
  Pi(root) ← 0
  for i ← 1 to n do
    m ← m + 1
    if b(i) ≥ 0 then
      Inode(m) ← i
      Jnode(m) ← root
      Flow(i) ← b(i)
      Dir(i) ← forward
      Pi(i) ← C
    end if
    else
      Inode(m) ← i
      Jnode(m) ← root
      Flow(i) ← −b(i)
      Dir(i) ← reverse
      Pi(i) ← −C
    end else
    Cost(m) ← C
    Bound(m) ← ∞
    State(m) ← 0
    Parent(i) ← root
    Arc(i) ← m
    Thread(i) ← i + 1
    Depth(i) ← 1
  end for
end proc
```

By our convention of storing the *State* of each arc, an arc is admissible if the product of its *State* and reduced cost, which we refer to as the *price* of the arc, is negative. Thus for a given basic solution, $(\mathcal{B}, \mathcal{L}, \mathcal{U})$, we choose the entering arc $k^* = (i^*, j^*)$ from among those arcs which have a negative price. If no arc has a negative price then the current solution is optimal.

The method of selection of a specific arc from among the admissible arcs is called the *Pricing Strategy* and there are perhaps as many of these as there are NSM codes, ranging from the very simple to the very complex. The object of every pricing strategy is to minimise

the average total solution time over a particular class, or classes, of problems. Grigoriadis (1986) estimates that up to three-quarters of the total solution time is accounted for by the pricing mechanism.

We have chosen to implement a number of pricing strategies as a representative sample of the methods used, along with one which we ourselves have developed. While those implemented in commercial codes remain closely guarded secrets, a wide range of so-called 'rules' are documented in the literature. In some cases, the data stuctures on which these strategies are based are quite different to our own. In order to implement them, therefore, we have added extra indexing mechanisms to simulate those other structures as far as is required, without prejudicing the speed of the original rule.

4.4.1 Most Negative

In the simplex method for standard LP problems, the entering variable is selected as the "variable with the most negative objective co-efficient because computational experience has shown that such a selection is more likely to lead to the optimum solution rapidly", Taha (1987). The *Most Negative* pricing strategy for NSM may be given the same justification. It should be noted that the term 'rapidly' here, refers strictly to the number of iterations required to achieve optimality – it does *not* refer to the time taken, which is what we are most interested in.

At each iteration, *all* non-basic arcs are re-priced, and the arc with the most negative price is selected as the entering arc. With the arcs stored as an arbitrarily ordered list, the implementation of this strategy is straightforward. A single pass is made through the entire arc list, and an index to the arc with most negative price so far is maintained. On completion of the pass, this index will point to the entering arc.

```
procedure Most Negative
begin
  k* ← NULL
  min ← 0
  for k ← 1 to m do
    if State(k) ≠ 0 then
      i ← Inode(k)
      j ← Jnode(k)
      c̄_k ← State(k) × (c_k − π(i) + π(j))
      if c̄_k < min then
        min ← c̄_k
        k* ← k
      end if
    end if
  end for
  if k* = NULL then current solution is optimal
end proc
```

4.4.2 First Negative

By contrast to the exhaustive search required to find the arc with the most negative price, the *First Negative* strategy performs the minimum amount of searching at each iteration. The entering arc is simply the first arc encountered with a negative price. Each subsequent iteration begins its search at the point where the last entering arc was found. If a full cycle of the arc list occurs during a single iteration, without finding an admissible arc, then we have reached the optimum solution.

The First Negative strategy is likely to find the entering arc far quicker on average than almost any other strategy. In general, the choice of pricing strategy involves a trade-off between the time to find each entering arc, and the number of times an entering arc must be found. Thus the number of iterations required to reach optimality may outweigh the benefit of faster arc selection.

procedure *First Negative*
start is the index after the previous entering arc
begin
  $k^* \leftarrow$ *NULL*
  $k \leftarrow start$
  if $start = 1$ **then** $last \leftarrow m$
  else $last \leftarrow start - 1$
  $finished \leftarrow$ *FALSE*
  while $NOT finished$ **do**
    if $State(k) \neq 0$ **then**
      $i \leftarrow Inode(k)$
      $j \leftarrow Jnode(k)$
      $\bar{c}_k \leftarrow State(k) \times \big(c_k - \pi(i) + \pi(j)\big)$
      if $\bar{c}_k < 0$ **then**
        $k^* \leftarrow k$
        $finished \leftarrow$ *TRUE*
      end if
    end if
    if $k = last$ **then** $finished \leftarrow$ *TRUE*
    else if $k = m$ **then** $k = 1$
    else $k = k + 1$
  end while
  $start \leftarrow k$
  if $k^* =$ *NULL* **then** *current solution is optimal*
end proc

4.4.3 Arc Block and Arc Sample

As a compromise between the two strategies outlined above, Grigoriadis (1986) describes a very simple *Arc Block* pricing strategy based on dividing the arc set into a number of subsets of specified size. At each iteration, the entering arc is selected from a block as the arc of that block with most negative price. Only the arcs of one block are re-priced at any iteration,

with consecutive blocks examined in each subsequent iteration. If no admissible arc is found in the first block searched, then the next block is also re-priced and searched. If all blocks have been exhausted, then the optimal solution has been reached.

By varying the block size, we can exploit the trade-off between time spent in pricing at each iteration, and the 'goodness' of the selected arc in terms of reducing the number of iterations required to reach the optimal solution. A block size of between 1% and 8.5% of the size of the arc set is recommended by for Grigoriadis (1986), for large MCF problems.

```
procedure Arc Block
start is the index after the previous entering arc
begin
  k* ← NULL
  min ← 0
  if start = 1 then last ← m
  else last ← start − 1
  blockstart ← start
  finished ← FALSE
  while (NOTfinished) do
    blockend ← blockstart + blocksize − 1
    if Blockend < m then blockend ← m
    for k ← blockstart to blockend do
      if State(k) ≠ 0 then
        i ← Inode(k)
        j ← Jnode(k)
        c̄_k ← State(k) × (c_k − π(i) + π(j))
        if c̄_k < min then
          min ← c̄_k
          k* ← k
        end if
      end if
    if k* ≠ NULL then finished ← TRUE
    if k = last then finished ← TRUE
    else if blockend = m then blockstart = 1
    else blockstart = blockend + 1
  end while
  if blockend = m then start ← 1
  else start ← blockend + 1
  if k* = NULL then current solution is optimal
end proc
```

Under the assumption that the list of arcs is arbitrarily ordered, the blocks may be chosen by simply examining the required number of consecutive arcs. However, as the input data for many problems is generated is some order – often in order of $Inode$ – we feared that the performance of the method might be distorted. To overcome this, we implemented a variation on the Arc Block strategy, which we call *Arc Sample*. Instead of selecting the

```
procedure Arc Sample
firstarc is the index of the first node in the sample
skip = m/sampsize
begin
  k* ← NULL
  min ← 0
  sampstart ← firstarc
  finished ← FALSE
  while (NOTfinished) do
    k ← firstarc
    while k < m do
      if State(k) ≠ 0 then
        i ← Inode(k)
        j ← Jnode(k)
        c̄_k ← State(k) × (c_k − π(i) + π(j))
        if c̄_k < min then
          min ← c̄_k
          k* ← k
        end if
      end if
      k ← k + skip
    end while
    if firstarc = skip then firstarc = 1
    else firstarc = firstarc + 1
    if k* ≠ NULL then finished ← TRUE
    else if firstarc = sampstart then finished ← TRUE
  end while
  if k* = NULL then current solution is optimal
end proc
```

entering arc from among the required number of consecutive arcs, this method considers arcs at constant intervals, called the *skip* factor, from throughout the entire arc set. The first sample begins with the first arc in the list, while subsequent iterations begin with the second and third arcs, etc.. In this way, the same number of Arc Samples as Arc Blocks are required to scan the entire arc set, but the arcs considered are likely to be more widely distributed throughout the network, in spite of any ordering of the arc list.

4.4.4 BBG Queue

The most complicated pricing strategy which we have implemented attempts to replicate that used by Bradley, Brown and Graves (1977), and is thus called the *BBG Queue*. In order to simulate the authors' forward star storage of the arc set, we added an extra node-length array, called *Slice* to our own structures. For each node $i \in \mathcal{N}$, $Slice(i)$ contains a pointer to a list of indices to that node's incident arcs. This is constructed with complexity $O(m)$, (as compared to an $O(m \log m)$ sort routine) and facilitates sequential access to all of the arcs incident with a given node, just as forward star does.

```
procedure BBG Queue
start is the address in the queue where the last
entering arc was found
begin
  k* ← NULL
  count ← 0
  q ← start
  finished ← FALSE
  while NOTfinished do
    if q = Back of Q then
      RefreshPage
      q = Front of Q
    end if
    if Q = Empty then finished ← TRUE
    else
      i ← q(Node)
      if i = NULL then
        k ← q(Arc)
        count ← count + 1
      end if
      else
        Delete q(Arc) from Q
        k ← BestAdjacentArc(i)
      end else
      if (k ≠ NULL ) then
        i ← Inode(k)
        j ← Jnode(k)
        c̄_k ← State(k) × (c_k − π(i) + π(j))
        if c̄_k < min then
          min ← c̄_k
          k* ← k
          Advance q along Q
        end if
        else if c̄_k ≥ 0 then Delete q(Arc) from Q
      end if
      if count = blocksize then
        if k* = NULL then count = 0
        else finished ← TRUE
      end if
    end else
  end while
  start = q
  if k* = NULL then current solution is optimal
end proc
```

The BBG Queue pricing strategy operates using a dynamic queue containing the indices of so-called 'interesting' nodes and admissible arcs. An 'interesting' node is a node whose incident arcs have not been re-priced in recent iterations. At each iteration, the entering arc is selected from a specified number of consecutive queue arcs, called the *blocksize*. Each of these arcs is individually re-priced. Any which have positive or zero prices are deleted from the queue, while the arc with most negative price enters the basis. If an interesting node is encountered on the queue, then all of its incident arcs are re-priced and considered for selection. All incident arcs with negative prices are added to the end of the queue.

Initially, the queue is loaded with all demand nodes of the problem. While the number of iterations is less than a specified number, called the opening *gambit*, both the *Inode* and *Jnode* of each entering arc are added to the end of the queue. The idea is to form 'chains' or 'paths' emanating from the demand nodes to quickly obtain a feasible solution in the original network, i.e. by eliminating artificial arcs from the basis tree.

After each full circuit of the queue, or in the event of it being emptied, the queue is refreshed. This is done by adding to it all of the admissible arcs incident with a specified number of nodes, called the *pagesize*. Each 'page' begins where the last one stopped, so that all the nodes, and thus all of the arcs, are eventually examined.

The choice of *blocksize*, *pagesize* and length of opening *gambit* is determined purely by empirical testing. It is probable that different parameters will be better suited to different classes of problems, given that the average number of arcs incident to each node varies with the sparsity of the problem. Bradley, Brown and Graves (1977) recommend the following default settings:

$gambit = n/10 + 1;\ pagesize = 3n/4;\ blocksize = 32.$

4.4.5 Mulvey List

Mulvey (1978) describes a candidate list which consists solely of arcs. His pricing strategy is based on performing both *Major* and *Minor* iterations. A minor iteration chooses the entering arc as the most negatively priced of the arcs on the candidate list. Any arcs which do not have negative prices are deleted from the list. In a major iteration, the arc list is scanned, and negatively priced arcs are added to the list, until either all arcs have been examined, or the number of arcs added reaches a preset length, called *listlength*.

The solution begins with a major iteration, and major iterations are thereafter performed when the list is emptied, or when a preset number of minor iterations, *minorlimit*, has been performed. This strategy, like the BBG Queue, is based on a forward star arc storage scheme, which we again simulate using the *Slice* array. Mulvey recommends $listlength = 40$ and $minorlimit = 20$.

The Mulvey List attempts to make use of all favourably priced arcs incident within a given block before moving on to the next block. The setting of *minorlimit* prevents the algorithm from concentrating on one node for a long period of time. The list of arcs is repeatedly cycled, in a manner similar to the Arc Block strategy, until the optimal solution is reached.

```
procedure Mulvey List
minors is the number of iterations since the last major iteration
begin
  k* ← NULL
  min ← 0
  if minors < minorlimit then CL ← Empty
  finished ← FALSE
  while NOT finished do
    if CL = Empty then
      MajorIteration
      minors ← 0
    end if
    if CL = Empty then finished ← TRUE
    else
      c ← CL
      while c ≠ End of CL do
        k ← c(Arc)
        if State(k) ≠ 0 then
          i ← Inode(k)
          j ← Jnode(k)
          c̄_k ← State(k) × (c_k − π(i) + π(j))
          if c̄_k < 0 then
            min ← c̄_k
            k* ← k
          end if
        end if
        Advance c along CL
      end while
      if k* = NULL then CL ← Empty
      else finished ← TRUE
    end else
  end while
  minors ← minors + 1
  if k* = NULL then current solution is optimal
end proc
```

4.4.6 Two Phase

The final pricing strategy which we have implemented is one which we developed ourselves. It is closely based on the Arc Sample strategy described above, but includes a variation on the phased strategy used by Bradley, Brown and Graves (1977). We allow the sample size to be changed at a certain point in the solution. During the first phase, the entering arc is selected as the most negatively priced arc in a sample of size $size1$. At each iteration, the price of this entering arc, $\bar{c}$, is compared to a pre-specified limit price, based on the cost penalty assigned to artificial arcs. If $\bar{c}$ is less than the limit, then the entering arc in all subsequent iterations is selected from a sample of size $size2$. The limit on $\bar{c}$, which

triggers the second phase of solution, is intended to ensure that all artificial arcs have been eliminated from the basis. The default value for the second sample size is one and a half times the size of the first sample, but the sample sizes may also be set independently.

4.5 Determining the Leaving Arc

The entering arc $k^* = (i^*, j^*)$ forms a unique cycle, $\mathcal{W}$, with the arcs of the Basis Tree. In order to restore the cycle-free property of the solution, therefore, one of the arcs of this cycle must leave the basis. Generally speaking, the determination of the leaving arc is much more restricted than the selection of an entering arc. There are three steps to this process, namely cycle-finding, identification of candidates to leave, and the selection of the leaving arc from among those candidates. In our implementation, the latter two steps are combined, with the objective of maintaining a *strongly feasible basis* throughout the solution of the problem. (For a graphical representation of $\mathcal{W}$, see Figure 3 in section 3.5).

4.5.1 Cycle-Finding

The unique cycle, $\mathcal{W}$, associated with a given entering arc, $k^* = (i^*, j^*)$, is found using the following reasoning. There is a unique path from each node in the tree to the root. This we call the *backpath*. Let $join$ be the first node common to the backpaths of i^* and j^*. Then $\mathcal{W}$ consists of the entering arc, plus all arcs on the paths from i^* to $join$ and j^* to $join$.

A number of methods for finding $join$ are described in the literature. Bradley, Brown and Graves (1977) and O'Connor (1980) outline a simple method which involves traversing the entire backpath of i^*, labelling each node along it. The backpath is traversed using the $Parent$ pointer of each node. Then $join$ is the first labelled node encountered on the backpath of j^*. Once $join$ has been found, all labels are erased.

The inherent inefficiency of this procedure is due to the labelling of nodes common to the backpaths of both i^* and j^*, i.e. nodes on the backpath of $join$. Though the extent of any such inefficiency depends on the length of the path from $join$ to the root, and may not be significant in practice, it can be easily eliminated by a slightly more clever procedure. The version we have adopted makes use of the $Depth$ index, and is found in Bradley, Brown and Graves (1977), Chvátal (1983) and Ahuja, Magnanti and Orlin (1989).

Both $Depth(i^*)$ and $Depth(j^*)$ must be greater than or equal to $Depth(join)$. In order to find $join$, then, we iterate upwards (using the $Parent$ index) along the backpath of the deeper node, until we reach an equal depth on both backpaths. At this point, we test for equality of the current nodes. If they are equal, we have found $join$ and the procedure is terminated. Otherwise, we move up a depth on *both* backpaths until a common node is reached. This means that we traverse both backpaths only as far as $join$, without explicitly labelling any nodes.

```
procedure Find Join
(i*, j*) is the entering arc
begin
  i ← i*
  j ← j*
  while i ≠ j do
    if Depth(i) < Depth(j) then i ← Parent(i)
    else if Depth(i) < Depth(j) then j ← Parent(j)
    else
      i ← Parent(i)
      j ← Parent(j)
    end else
  end while
  join ← i
end
```

An equivalent cycle-finding method, also described by Bradley, Brown and Graves (1977), uses a 'Number of Successors' index instead of the depth index. In this case, one iterates upwards from the node with the lesser number of nodes in its subtree. If i has fewer successors than j, then i may be in the subtree of j, but not vice versa. Equality of i and j is checked whenever both nodes have the same number of successors.

4.5.2 Arc Selection

The cycle $\mathcal{W}$ represents the negative cost augmenting cycle associated with entering arc $k^* = (i^*, j^*)$. The cycle is eliminated by augmenting flow in all arcs of the cycle by the maximum amount possible. If the entering arc is currently at its lower bound, we attempt to increase the flow in that arc. Conversely, the flow in an entering arc which is currently at its upper bound would be decreased. For all other arcs in the cycle, we increase or decrease the flow as appropriate to their directions, and the direction of flow change in the entering arc.

We first define the orientation of the cycle, $\mathcal{W}$. If the entering arc, k^*, is currently at its lower bound, then the orientation of $\mathcal{W}$ is in the direction $Inode(k^*)$ to $Jnode(k^*)$. If, on the other hand, k^* is currently at its upper bound, then $\mathcal{W}$ is oriented in the direction $Jnode(k^*)$ to $Inode(k^*)$. Thus, flow will be increased in all arcs whose orientation is the same as the orientation of the cycle, and decreased in all arcs whose orientation is opposite to the cycle.

To identify the candidate leaving arcs, we will calculate for each basic arc in the cycle, $k = (i, j) \in \mathcal{W}$, the allowable flow increase or decrease, δ_k, in that arc. If k^* is at its lower bound, 0, then we may increase flow in it by the amount of its upper bound. If it is at its upper bound, we may decrease flow in the arc by the amount of that bound. Hence, the allowable flow increase or decrease in the entering arc, δ_{k^*} is always equal to $Bound(k^*)$. The overall flow change, δ, is then given by:

$$\delta = \min\{Bound(k), \delta_k \text{ for all } k \in \mathcal{W}\}$$

All arcs $k \in \mathcal{W}$, for which $\delta_k = \delta$ are candidate leaving arcs. In order to maintain a strongly feasible basis, we break any ties by choosing the *last* candidate arc encountered in

```
procedure Leaving Arc
k* = (i*, j*) is the entering arc
begin
  if State(k*) < 0 then
    first ← i*
    second ← j*
  end if
  else
    first ← j*
    second ← i*
  end else
  δ ← Bound(k*)
  k ← second
  while k ≠ join do
    if Dir(k) = forward then δ_k ← Flow(k)
    else δ_k ← Bound(Arc(k)) − Flow(k)
    if δ_k ≤ δ then
      δ ← δ_k
      i_out ← k
      i_new ← first
    end if
    k ← Parent(k)
  end while
  k ← second
  while k ≠ join do
    if Dir(k) = forward then δ_k ← Bound(Arc(k)) − Flow(k)
    else δ_k ← Flow(k)
    if δ_k < δ then
      δ ← δ_k
      i_out ← k
      i_new ← second
    end if
    k ← Parent(k)
  end while
end proc
```

a traversal of the cycle, in the direction of its orientation, starting at $join$. To this end, we define the ordering of the search for the leaving arc as follows:

$$first, second = \begin{cases} i^*, j^* & \text{if } k^* \in \mathcal{L} \text{ (i.e. at lower bound)} \\ j^*, i^* & \text{if } k^* \in \mathcal{U} \text{ (i.e. at upper bound)} \end{cases}$$

Thus, in terms of the orientation of $\mathcal{W}$, the path from $first$ to $join$ represents the first portion of the cycle, while the path from *second* to $join$ represents the latter portion. With δ initialized to δ_{k^*}, we traverse the path from *second* to $join$, comparing δ_k for each arc, k, on this path to the current minimum δ value. By using a '$\leq$' comparison, we are assured that k^* is the index of the *last* candidate arc encountered on this path. Along this path, we are traversing $\mathcal{W}$ in the direction of its orientation, so the allowable flow increase or decrease in arc k is defined:

$$\delta_k = \begin{cases} Bound(k) - Flow(k) & \text{if } Dir(k) = forward \\ Flow(k) & \text{otherwise} \end{cases}$$

In traversing the path from $first$ to $join$, we are traversing $\mathcal{W}$ in a direction opposite to its orientation. We would therefore select the *first* candidate encountered along this path to leave the basis, and so use a '$<$' comparison between $delta_k$ and $delta$. The definition of the allowable flow increase or decrease is also opposite to the one used in traversing the previous path. Thus on the path from $first$ to $join$,

$$\delta_k = \begin{cases} Flow(k) & \text{if } Dir(k) = forward \\ Bound(k) - Flow(k) & \text{otherwise} \end{cases}$$

As the *Arc* index of each node stores the arc between that node and its parent in the basis tree, we identify leaving arc by storing the index of the node whose associated *Arc* is to leave the basis. This node we call i_{out}. We also maintain an index, i_{new}, such that i_{out} is on the path from i_{new} to $join$ in the current basis. This enables us to determine whether, in the new basis tree, i^* should be the parent of j^*, or vice versa.

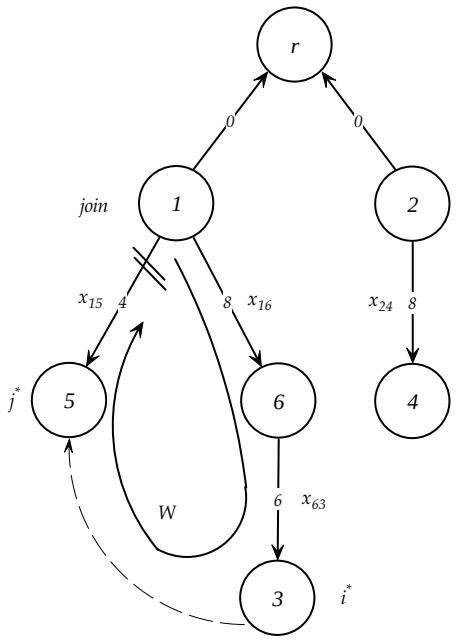

Figure 5. Leaving-Arc Determination in Ireland Network.

Returning again to the distribution network of Ireland, we suppose that arc $(3, 5)$ is to be the entering arc, and that it is currently at its lower bound, 0. The augmenting cycle $\mathcal{W}$ is as marked in Figure 5. We assume for simplicity that the problem is uncapacitated, thus flow in the entering arc may be increased by an infinite amount. The path from $i^* = 3$ to $join = 1$ is traversed first. Both arcs on this path are have directions consistent with the orientation of $\mathcal{W}$, and the flow in both may also be increased by an infinite amount. The single arc, $(1, 5)$, which forms the backpath from $j^* = 5$ to 1 is directed opposite to the orientation of the augmenting cycle, and so is determined to be the leaving arc. Thus i_{out} and i_{new} are both 5, since in the new tree, node 3 will be the parent of node 5.

4.6 Performing the Pivot

The group of procedures required to replace the leaving arc with the entering arc at each iteration are collectively referred to as the pivot. Essentially, this involves updating flows, node potentials, the *State* of the arcs exchanged, and the various tree indices – *Thread*, *Depth*, *Parent*, *Arc* and *Dir* – to describe the new BFS. In terms of the LP Simplex method, the flow change is equivalent to the updating of the right-hand side of the tableau. Updating the node potentials is an update of the dual variables. Finally, the reconstruction of the tree corresponds to calculating the new simplex tableau, in terms of basic variables. The order in which these procedures are performed is vital, and adheres strictly to the order outlined below. If the entering arc is also the leaving arc, then the tree indices need not be updated.

4.6.1 Changing Flows

The flows in arcs of the cycle W increase or decrease by the amount δ. It may also happen that δ is zero, in which case this procedure is not performed at all. We define the direction of flow increase, $DirInc$ as follows:

$$DirInc = \begin{cases} forward & \text{if } k^* \text{ is at its lower bound} \\ reverse & \text{if } k^* \text{ is at its upper bound} \end{cases}$$

We then traverse the paths from i^* to $join$ and j^* to $join$, increasing or decreasing the flow in each arc, $k \in W$, as appropriate. On the path from i^* to $join$, the new flow is:

$$Flow(k) \leftarrow \begin{cases} Flow(k) + \delta & \text{if } Dir(k) = DirInc \\ Flow(k) - \delta & \text{otherwise} \end{cases}$$

Conversely, on the path from j^* to $join$, since we are traversing the cycle in the opposite direction, the new flow is:

$$Flow(k) \leftarrow \begin{cases} Flow(k) - \delta & \text{if } Dir(k) = DirInc \\ Flow(k) + \delta & \text{otherwise} \end{cases}$$

4.6.2 Updating Parent and Thread Indices

In the discussion which follows, the leaving arc, (p, q), is referred to as the arc joining node i_{out} to its parent, j_{out}, in the current basis tree. The endpoints of the entering arc,

```
procedure Change Flows
begin
  if State(k*) < 0 then DirInc ← forward
  else DirInc ← reverse
  k ← i*
  while k ≠ join do
    if Dir(k) = DirInc then Flow(k) ← Flow(k) + δ
    else Flow(k) ← Flow(k) − δ
    k ← Parent(k)
  end while
  k ← j*
  while k ≠ join do
    if Dir(k) = DirInc then Flow(k) ← Flow(k) − δ
    else Flow(k) ← Flow(k) + δ
    k ← Parent(k)
  end while
end proc
```

$k^* = (i^*, j^*)$, are relabelled so that j_{new} will be the parent of i_{new} in the new basis tree. The nodes on the path from i_{new} to i_{out} are called *stem* nodes.

Simply stated, the parent relationships between all pairs of stem nodes must be reversed, as the nodes on the path from i_{new} to i_{out}, and their subtrees, will be successors of j_{new} in the new basis tree. We perform this operation in combination with the updating of the thread index. The outline for our procedure draws on the methods described by Bradley, Brown and Graves (1977), Chvátal (1983) and Ahuja et al (1989). However, the complete procedure is significantly more complex, as a number of special cases must be dealt with. We found little mention of these special cases in the literature, save an allusion by Ahuja, Magnanti and Orlin (1989) to the extent that "this step is rather involved".

In a preorder traversal, a node may only be visited *after* its parent, and all other predecessors, have been visited. Thus all nodes on the new subtree of i_{new} (i.e. all stem nodes and their subtrees) should be visited *after* node j_{new} in the new traversal. For each node on the stem, we refer to those of its successors which appear on the thread before another stem node as *left* successors, and those which appear on the thread after the next stem node (and its subtree) as *right* successors. This follows the convention adopted by Bradley, Brown and Graves (1977).

For each node on the stem, we find its *last* right successor, i, and let $right = Thread(i)$. Thus *right* is the first node on the thread after all successors of the stem node. $Thread(i)$ is reset to the parent of the stem node, so that the preorder is preserved when the parent relationship between these two nodes is reversed. Next, we find the last left successor of the parent, and set its thread to *right*. Finally, we reverse the relationship between the stem node and its parent. Thus the preorder will visit the stem node, followed by its original parent and the nodes of its subtree (which are now its successors), before continuing as before with the node we call *right*.

The index *last* is used to store the node which followed the last right successor of i_{out} on the original thread. Thus if i is the node which appeared on the original thread immediately

```
procedure Update Thread and Parent
begin
  parfirst ← FALSE
  if join = j_out then
    i ← Parent(join)
    while Thread(i) ≠ i_new AND Thread(i) ≠ j_new do
      i ← Thread(i)
    if Thread(i) = j_new then
      parfirst ← TRUE
      while Thread(i) ≠ i_out do
        i ← Thread(i)
      first ← i
    end if
  end if
  i ← i_new
  while Depth(Thread(i)) < Depth(i_new) do
    i ← Thread(i)
  right ← Thread(i)
  if Thread(j_new = i_out then
    last ← i
    while Depth(last) < Depth(i_out) do
      last ← Thread(last)
    if last = i_out then last ← Thread(last)
  end if
  else last ← Thread(j_new)
  Thread(j_new) ← i_new
  stem ← i_new
  parstem ← j_new
  while stem ≠ i_out do
    Thread(i) ← Parent(stem)
    i ← Parent(stem)
    while Thread(i) ≠ stem do
      i ← Thread(i)
    Thread(i) ← right
    newstem ← Parent(stem)
    Parent(stem) ← parstem
    parstem ← stem
    stem ← newstem
    ← istem
    while Depth(Thread(i)) < Depth(stem) do
      i ← Thread(i)
    right ← Thread(i)
  end while
  Thread(i) ← last
```

```
if join = j_out then
  if NOTparfirst then
    i ← j_out
    while Thread(i) ≠ i_out do
      i ← Thread(i)
    Thread(i) ← right
  end if
  else if first ≠ j_new then Thread(first) ← right
end if
else
  i ← j_out
  while Thread(i) ≠ i_out do
    i ← Thread(i)
  Thread(i) ← right
end else
Parent(i_out) ← parstem
end proc
```

before i_{out}, then by setting $Thread(i) = last$, we exclude i_{out} and its subtree from the thread following j_{out}.

Special cases arise particularly when the nodes j_{out} and $join$ coincide. In this case we must determine whether i_{new} or j_{new} appear first on the original thread. If i_{new} is first, then the procedure is not altered. If, however, j_{new} appears first, then we use an additional index, $first$, to store the node which appears on the thread immediately before i_{out}.

4.6.3 Updating Arc and Direction Indices

The three indices Arc, $Flow$, and Dir, describe the arc joining a node to its parent in the basis tree. As the parent relationships between stem nodes are reversed, so must the information in these indices be exchanged. These exchanges are performed along a traversal of the path from i_{out} ot i_{new}, using the new parent indices. The flow, and direction of flow, in the entering arc is also set at this point.

4.6.4 Updating Node Potentials and Depths

The updating of the final pair of indices, Pi and $Depth$, is performed by traversing the subtree of j_{new} using the reconstructed thread. These are the only nodes whose potentials and depths are affected by the pivot. The new potential of each node is calculated with respect to the potential of its parent and the cost of the arc between them. For each basic arc, $(i, j) \in \mathcal{B}$, we must satisfy the relation $\bar{c}_{ij} = c_{ij} - \pi(i) + \pi(j) = 0$. Thus $\pi(i) = \pi(j) + c_{ij}$.

Given that the depth of any node must be one greater than the depth of its parent, the preorder traversal of the subtree of j_{new} allows us to update both potentials and depths in a single visit to each node. Neither the depth nor the potential of j_{new} itself are changed by the pivot.

Figure 6 shows the spanning tree solution obtained by performing a pivot on the distribution network of Ireland. Arc $(3, 5)$ is now basic with flow of four units, having replaced arc $(1, 5)$. In this case, there was only one stem node, namely node 5. It is now a successor

```
procedure Update Arc Flow Dir Indices
begin
  i ← i_out
  while i ≠ i_new do
    j ← Parent(i)
    Arc(i) ← Arc(j)
    Flow(i) ← Flow(j)
    if Dir(j) ← forward then Dir(i) ← reverse
    else Dir(i) ← forward
    i ← j
  end while
  Arc(i_new) ← k*
  if State(k*) = 1 then Flow(i_new) ← δ
  else Flow(i_new ← Bound(k*) − δ
  if i_new ← Inode(k*) then Dir(i_new) ← forward
  else Dir(i_new) ← reverse
end proc
```

```
procedure Update Depths and Potentials
begin
  i ← Thread(j_new)
  finished ← FALSE
  while NOTfinished do
    j ← Parent(i)
    if j = NULL then finished ← TRUE
    else
      k ← Arc(i)
      Depth(i) ← Depth(j) + 1
      if Dir(i) = forward then Pi(i) ← Pi(j) + Cost(k)
      else Pi(i) ← Pi(j) − Cost(k)
      if Depth(i) ≤ Depth(j_new) then finished ← TRUE
      else i ← Thread(i)
    end else
  end while
end proc
```

of node 3. The flow in arcs $(1, 6)$ has been incremented by four units. The other arcs of the basis, and their flows, are unchanged. Also shown are the updated parent, thread and depth indices. Only the potentials of nodes 3, 5 and 6 would be updated, as they are nodes on the subtree of node 1, the parent of i_{out}.

Node i	1	2	3	4	5	6	r
Parent(i)	r	r	6	2	3	1	r
Thread(i)	6	4	5	r	2	3	1
Depth(i)	1	1	3	2	4	2	0

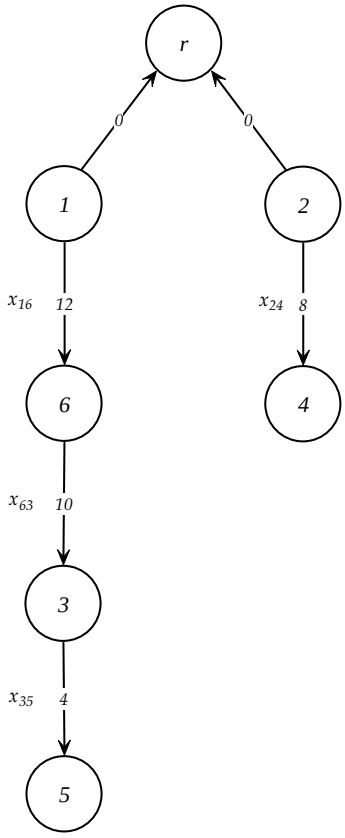

Figure 6. New Spanning Tree Solution for Ireland.

4.7 Step-Through Mode

From our discussion of NSM and its implementation, it is clear that the representation of the basis as a rooted tree is central. In order to aid understanding of this aspect of the method, we have developed a *Step-Through Mode* for the solution of small problems. The user is given control over the performance of each iteration, and the solution may be viewed on the screen. For aesthetic purposes, we have limited the application of this method to problems with 20 nodes or less, in order that the graphics screen does not become too cluttered. The number of arcs in a problem is not limited. A most negative pricing rule is used throughout the solution, since this generally yields a smaller number of iterations. For small problems, the exhaustive search for the entering arc is used as the solution time is not relevant.

4.7.1 Drawing the Basis Tree

The crucial consideration in drwaing the basis tree at each iteration is to ensure that tree-arcs do not cross one another. If this were to occur, then the cycle-free property of the tree would not be obvious, and the instructive purpose of the step-through mode would be defeated. Hence the need to adopt a strict convention for drawing the trees. We consistently place the root of the tree in the top, left-hand corner of the graphics screen. The successors of each node are then located below and to the right of that node. Thus the vertical position

```
procedure Accumulate Widths
S is a stack
begin
  i ← Thread(root)
  Width(root) ← 0
  while i ≠ root do
    Insert ionto S
    Width(i) ← 0
    i ← Thread(i)
  end while
  while NOTEmpty(S) do
    i ← DeleteTop(S)
    j ← Parent(i)
    if Width(i) = 0 then Width(i) ← 1
    Width(j) ← Width(j) + Width(i)
  end while
end proc
```

of each node is determined by its depth index, while its horizontal position is relative to the horizontal position of the parent and siblings of the node.

The first task faced by the drawing procedure is to determine the width of the subtree of each node. We can then ensure that sufficient space is allocated between nodes at a given depth, so that the successors of these nodes are not too cosely packed. In order to determine these widths, we need to visit all successors of a given node *before* its parent. This is achieved with the help of a *Stack*. A stack is described by Aho, Hopcroft and Ullman (1983) as "a special kind of list in which all insertions and deletions take place at one end, called the *top*". Thus by inserting the nodes of the tree onto the stack in preorder (i.e. the order in which they are encountered in following the thread index), and then deleting them from the stack, we can perform an "inverse-preorder" traversal of the tree, accumulating widths as we go.

The nodes of the basis tree are then drawn by following the thread (in its usual direction). The horizontal position of the *next* node to be drawn at each depth is stored in an array called $Offset$. The precise horizontal position of each node is stored in $Hpos$. If a node is the first successor of its parent encountered on the thread, it is placed directly below its parent, with all its siblings to the right. The tree arcs are represented as lines joining each node to its parent. If i and j are two nodes, such that j is the parent of i, then a basic arc (i, j) is indicated by a *positive* flow value, whereas the arc (j, i) would be indicated by a *negative* flow value. Each node is numbered as in the original problem formulation, with R representing the root.

```
procedure Draw Basis Tree
begin
  for i ← 1 to root do
    FirstChild(i) ← TRUE
    Offset(i) ← 1
  end for
  Draw root at (1, 1)
  Hpos(root) ← 1
  i ← Thread(root)
  while i ≠ root do
    j ← Parent(i)
    di ← Depth(i)
    dj ← Depth(j)
    if FirstChild(i) then
      Offset(di) ← Hpos(i)
      FirstChild(i) ← FALSE
    end if
    Hpos(i) ← Offset(di)
    Offset(di) ← Offset(di) + 1
    Draw i at (Hpos(i), di)
    Draw arc from (Hpos(i), di) to (Hpos(j), dj)
    i ← Thread(i)
  end while
end proc
```

Finally, arcs which are out of the basis at their upper bounds are appended to the graph, in a different colour, so that the graphics image fully describes the solution at each stage.

```
procedure Draw Arcs at Upper Bounds
begin
  for k ← 1 to arcs do
    if State(k) < 0 then
      i ← Inode(k)
      j ← Jnode(k)
      Draw arc from (Hpos(i), Depth(i)) to (Hpos(j), Depth(j))
    end if
  end for
end proc
```

Chapter 5

Testing and Applications

5.1 Test Problem Generation

In order to test the speed of our NSM code, and in particular, the relative solution speeds using the various pivot rules described in section 4.4, we needed to generate a significant number of test problems. Our original intention was to use the NETGEN code, due to Klingman, Napier, and Stutz (1974), specifically for this purpose. This would have given us the benefit of comparing the performance of our code and results to the results obtained by Grigoriadis (1986) and Bertsekas and Tseng (1988). (These studies were discussed in section 2.4 above.) More importantly, use of the NETGEN are generally accepted to represent a comprehensive set of artificial problems. Unfortunately, we were unable to obtain NETGEN, and proceeded therefore to develop our own test problem generator, which we have called OURGEN. Before presenting the results of our tests, we describe the methodology underlying OURGEN. Our empirical testing is dependent on the successful generation of realistic network problems by this code.

5.1.1 OURGEN Methodology

In designing OURGEN, we have closely followed the verbal description of NETGEN contained in Klingman, Napier, and Stutz (1974). The following are the input parameters for problem generation:

$nodes$	– total number of nodes in the problem
$psource, psink$	– number of pure supply and demand nodes respectively
$totarcs$	– total number of arcs in the problem
$cmax$	– maximum per unit flow cost (minimum is 1)
$totsupply$	– total supply to be distributed among supply nodes
$tsource, tsink$	– number of transshipment supply and demand nodes respectively
$pcdiff$	– percentage 'difficulty' (see below)
$pccap$	– percentage of arcs to be capacitated
$umin, umax$	– range of upper bounds for capacitated arcs
$seed$	– seed for random number generation

Nodes not accounted for by $psource$, $psink$, $tsource$ and $tsink$ parameters are included as pure transhipment nodes. As with NETGEN, each problem is generated in two parts. The first part of the program distributes the total supply and constructs a *skeleton* network, for which a feasible solution is guaranteed to exist. The second part completes the network by repeatedly generating random arcs until the number of arcs in the problem is equal to $totarcs$.

The Feasible Skeleton

Throughout the first part of the program, transshipment sources and sinks are treated as pure supply and demand nodes respectively. The total supply is distributed randomly among the source nodes. For each of these nodes, a 'chain' of arcs joining the supply node to a random selection of pure transshipment nodes is created. Arcs are then added which join the last node in the chain to a random selection of demand nodes. The supply at the

source node is distributed among these demand nodes. In the case of pure transportation and assignment problems, each supply node is connected directly to a number of demand nodes.

A percentage, *pccap*, of these skeleton arcs must be capacitated. In order to ensure feasibility, the upper bound on a skeleton arc is set to the larger of the supply at the source of the chain to which it belongs, and the minimum upper bound, *umin*. The lower bound of all arcs in the problem will be zero. The 'difficulty' of the problem is influenced by assigning a large per unit cost to a percentage, $pcdiff$, of the skeleton arcs. Klingman, Napier and Stutz (1974) reason that this makes the use of skeleton arcs in a solution unattractive.

Completing the Network

If the number of arcs in the skeleton is less than *totarcs*, then OURGEN proceeds to generate random arcs to rectify this shortfall. At this stage, the distinction between transshipment supply or demand nodes and pure supply and demand nodes is made. No arc may have a pure supply node as its *Jnode*, or a pure demand node as its *Inode*. The appropriate percentage of arcs are capacitated, and a random per unit flow cost in the range $1-cmax$ is assigned to all arcs, except to skeleton arcs with a large per unit flow cost already assigned to them.

procedure *Complete Network*
begin
  $allcaps \leftarrow (totarcs \times pccap/100) + arcs - skelcaps$
  while $arcs \leq totarcs$ **do**
    $Inode(arcs) \leftarrow$ *Random puresource or transhipment node*
    $Jnode(arcs) \leftarrow$ *Random puresink or transhipment node*
    if $arcs \leq allcaps$ **then** $Bound(arcs) \leftarrow$ *Random umin to umax*
    $arcs \leftarrow arcs + 1$
  end while
  for $k \leftarrow 1$ **to** *totarcs* **do**
    if $Cost(k) = 0$ **then** $Cost(k) \leftarrow$ *Random 1 to cmax*
  end for
end proc

5.1.2 Test Problem Specifications

The following table describes the characteristics of the problems generated using OURGEN and used to test our NSM code. We have based our testing on the specifications of the first twenty-seven NETGEN problems, which we have divided into the six classes shown in Table V. Classes A and B consist of pure transporation problems. The problems in Class B are larger than those in Class A. Class C problems are pure Assignments. Classes D,E and F, consist of Minimum Cost Flow problems, with 20%, 40%, and 80% of arcs capacitated respectively. All of the MCF problems have $n = 400$ nodes, but in Class F, there are very few sources and sinks. The details of test parameters and output are included in the Appendices. For each of the four or five problems in each class, we generated three examples, using the

```
procedure ConstructSkeleton
begin
  sources ← psource + tsource
  sinks ← psink + tsink
  ptrans ← nodes − (sources + sinks)
  for k ← 1 to totarcs do Cost(k) ← 0
  for i ← 1 to nodes do Supply(i) ← 0
  Initialise Random Seed
  for i ← 1 to sources do Assign Random Supply(i)
  for k ← 1 to totarcs do Bound(k) ← totsupply
  arcs ← 1
  for i ← 1 to sources do
    s ← i
    chain ← Random number of transshipment nodes
    for j ← 1 to chain do
      t ← Random transhipment node
      Inode(arcs) ← s
      Jnode(arcs) ← t
      Flow(arcs) ← Supply(i)
      arcs ← arcs + 1
    end for
    dests ← Random number of sink nodes
    for j ← 1 to dests do
      s ← Random sink node
      Inode(arcs) ← t
      Jnode(arcs) ← s
      Flow(arcs) ← Supply(i)
      arcs ← arcs + 1
      Assign Random Negative Supply(s) from Supply(i)
    end for
  end for
  skelcaps ← arcs × pccap/100
  for k ← 1 to skelcaps do
    p ← Random arc
    if Flow(p) < umin then Bound(p) ← Flow(p)
    else Bound(p) ← umin
  end for
  skeldiff ← arcs × pcdiff/100
  for k ← 1 to skeldiff do
    p ← Random arc
    Cost(p) ← cmax × 10
  end for
end proc
```

random seed numbers 1, 7000, and 14000.

Table V

OURGEN Problem Classes

Class	*No. of Problems*	*n*	*Range of m*	*Cost Range*	*Total Supply*
A	5	200	1398-2900	1-100	1000
B	5	300	3196-6370	1-100	1000
C	5	400	1619-4786	1-100	200
D	4	400	2961-3894	1-100	4000
E	4	400	3124-4806	1-100	4000
F	4	400	2046-3602	1-100	4000

5.2 Test Results and Interpretation

5.2.1 Hardware and Software Configuration

The NSM code tested here was written and compiled using Turbo Pascal version 5.5, developed in 1988 by Borland International Inc. The code was compiled for use with an Intel 8087/80287 numeric co-processor, with range-checking and stack-checking switched *off*. The tests were performed on am IBM PC-compatible machine, using DOS, and equipped with an Intel 80387 micro-processor, running at a clock-speed of 20 MHz.

5.2.2 Summary of Results

Tables VI and VII contain summaries of the performance of the seven pricing strategies which we have implemented, across the six classes of test problems outlined above. By examining the total solution times, and the number of iterations performed under each strategy, we gain an insight into the suitability, or otherwise, of different strategies in solving problems with varying characteristics. Table VI contains the *aggregate* solution times for the four or five problems in each class, for each of the seven pricing strategies. Aggregates are used in order to highlight the variations in speed of solution observed. Table VII contains the *average* number of iterations performed by each pricing strategy, in each of the six problem

classes. Graphical representations of this data are contained in Figures 7 and 8.

Table VI

Aggregate Solution Times (seconds) by Problem Class

Class	*Most Negative*	*First Negative*	*Arc Block*	*Arc Sample*	*Two Phase*	*Mulvey List*	*BBG Queue*
A	42.33	23.32	8.17	8.64	8.83	10.76	18.44
B	172.14	66.19	21.70	23.24	24.89	28.66	49.28
C	121.71	44.85	16.24	17.10	17.28	19.48	33.07
D	17.62	5.13	4.17	3.48	3.46	5.90	11.66
E	39.82	11.35	7.96	6.62	6.66	9.54	18.04
F	4.24	1.76	1.13	1.20	1.30	2.34	5.12
Total	397.86	152.60	59.37	60.27	62.43	76.68	135.62

Table VII

Average Number of Iterations by Problem Class

Class	*Most Negative*	*First Negative*	*Arc Block*	*Arc Sample*	*Two Phase*	*Mulvey List*	*BBG Queue*
A	311	1711	481	503	482	565	923
B	505	3464	747	766	754	962	1864
C	572	3181	780	806	815	933	1643
D	99	475	214	214	215	191	313
E	193	885	348	327	341	326	612
F	28	134	53	71	82	52	74

Figure 7: Aggregate Solution Times (seconds) by Problem Class

Figure 8: Average Number of Iterations by Problem Class

Most Negative and First Negative

The acid test of a pricing rule is the total solution time to optimality. The Most Negative rule is immediately eliminated from further consideration. It is three times slower than the sampling rules, i.e. the Arc Block, Arc Sample, and Two Phase methods, due to its performance of an exhaustive $O(m)$ search for the entering arc at each iteration. The First Negative rule performs badly for the transportation and assignment problems, giving times of more than double those of the sampling methods. For MCF problems however, the disparity between First Negative and sampling methods is not as pronounced. For these latter problems, First Negative solution times are between 50% and 70% in excess of the sampling method times. This may be explained by observing that for MCF problems there are far fewer source and sink nodes than transshipment nodes, and artificial arcs joining transshipment arcs to the root do not influence the objective function. Therefore the number of admissible arcs at each iteration is relatively low, so First Negative will generally choose a 'good' arc.

BBG Queue and Mulvey List

The pricing strategy due to Bradley, Brown, and Graves (1977), fails empirically. Its solution times are consistently twice to three times those of the sampling methods. The phased movement towards optimality is theoretically elegant, but the complexity of the operations involved leads to an very large computational overhead. The candidate list pricing strategy, due to Mulvey (1978), is similar in approach to the BBG Queue. The Mulvey List is however, far simpler than the BBG Queue, and this simplicity is manifest in the superior performance the method. For the transportation and assignment problems, the solution times for the Mulvey rule are only 20% to 30% in excess of those of the sampling methods, although this margin increases to between 45% and 100% for the MCF problems. It seems that as the problems in Classes D,E and F were solved in relatively few pivots across all methods, so the computational overhead involved in forming and maintaining the candidate list was higher on a 'per-pivot' basis.

Arc Block, Arc Sample and Two Phase

The unambiguous finding of our testing is that the three best pricing strategies from among those that we tested are the Arc Block, Arc Sample, and Two Phase methods. These methods are all based on the idea, suggested by Grigoriadis (1986), of selecting the incomming arc as the 'best' candidate from within a subset, or sample of the arcs at each iteration. Grigoriadis' (1986) own method takes the sample as a consecutive group of arcs, (i.e. an Arc Block) whereas Arc Sample takes constructs each subset from throughout the set of arcs, thus reducing the influence of any arc ordering due to the generation of the test data. The Two Phase method is based on the Arc Sample methodology, but the sample size increases at some stage during the solution of the problem. Our implementation differs from that of Grigoriadis (1986) as we have chosen to use the strongly feasible basis technique, which he omits, while we do not include his gradual penalty method for the elimination of artificial arcs from the basis.

The solution times for these methods are consistantly better than those for any of the other methods tested. The longest solution time recorded by any of these methods was 6.37 seconds – and that for a problem with 300 nodes and 6300 arcs. For transportation and assignment problems the Arc Block rule is the best of the three. This almost certainly

arises from the ordering of the arcs in these problems. Since all arcs of the original problem with the same *Inode* are listed in contiguous space, the arcs incident at each node would be considered in rotation. For lightly and moderately capacitated MCF problems the Arc Sample based methods are preferable. The Two Phase method proved faster for the lightly capacitated problems in Class D, while Arc Sample was better suited to those of of moderate capacitation (Class E). For heavily capacitated MCF problems, the Arc Block rule performed best, but this could also be due to the fact that these problems have fewer sources and sinks than the other MCF problems.

Total Solution Times

The total times for solving all of the problems in a given class are not directly comparable with those of Bertsekas (1985), Grigoriadis (1986), or Bertsekas and Tseng (1988), cited earlier, as we used a different problem generator. However, the fact that our times are no worse than double those of these researchers strongly suggests that an efficient PC-based implementation of the Network Simplex Method may offer a real alternative to codes designed for much larger machines, like the VAX 11/750 or IBM/370 family, for many applications.

5.2.3 Pricing Rule Parameter Settings

The parameters required by the pricing rules are as follows:

- For the sampling methods we specify the sample size. Two Phase uses a second sample size equal to $1\frac{1}{2}$ times the first sample size.
- The Mulvey List method requires two parameters, namely the maximum size of the list, and a bound on the number of minor iterations. For these we have used a ratio of 2:1.
- The required parameters for the BBG Queue method are the size of the opening gambit, the page size, and the block size. We have consistently used the authors recommended default values of 75% of nodes, $\frac{m}{10} + 1$, and 32 respectively.

The following recommendations are based on the results of our testing. In cases where we give a range for the optimal parameter, the range extends over the optimal settings for 80% of the problems in that class.

Transportation Problems

The optimal sample size for these problems lies in the range 4.0% to 5.6% of the total number of arcs. This compares with Grigoriadis' (1986) recommendation of 2.1% of m. The optimal range for the Mulvey List length was in the range 1.5% to 2% of m.

Assignment Problems

Our experience with assignment problems promotes a significantly different range of parameter settings to those used for the transportation problems. The optimal parameters, rather than settling around a certain percentage of the total number of arcs, decreased percentage-wise as the problem density increased. For example, the parameters for the sampling methods declined from 7.4% to 3.1%, and those for the Mulvey List decreased from 5.6% to 2.7%. There were indications that these parameters were close to stabilising, as the rate of decrease had slowed. Grigoriadis (1986) recommends a block size of 7.1% for assignment problems.

MCF Problems

For MCF problems, the range is from 3.3% to 6.1% of the number of arcs worked best for the sampling strategies, while 1.3% to 2% of m proved most suitable for the Mulvey List. These are similar to the ranges recommended for transportation problems, but the optimal settings here were more widely spread. This could be expected, given that a the broader range of MCF problems were generated. A block size of 4.3% of m is recommended by Grigoriadis (1986).

5.3 MCF Network Applications

Before presenting the industrial application which we have modeled and solved using the MCF problem framework, we briefly review a number of network applications extant in the literature. These applications constitute a major part of management science practice, with their number and diversity increasing. Glover and Klingman (1977) cite the following reasons for this development:

(i) significant advances in solution methodology,
(ii) improved computational software due to intensive code development,
(iii) rigorous empirical studies to determine the best solution and implementation procedures,
(iv) extensions of the network framework to broader classes of problems.

5.3.1 Documented Applications

The classical transportation problem is perhaps the best known network model. A notable application of this model by T.C. Koopmans in 1947 involved the reduction of cargo shipping times. Use of the model is not restricted to distribution problems, however, and it can be applied to a range of situations. Roberts (1978) applies the method to street-sweeping, and to Ribonucleic Acid (RNA) mapping. Glover and Klingman (1977) solve a file aggregation problem which arose in the United States Department of the Treasury, and a crop efficiency problem. In Chachra, Ghare and Moore (1979), the problem of moving soil from sites to fill dams or other such destinations is treated. The authors claim that this is the oldest recorded application of Graph Theory, having been used by a French engineer named Monge, in 1784.

The transshipment and general MCF formulations have also been extensively applied. Glover and Klingman (1977) describe the routing of motor vehicle transportation across the United States for a major automobile manufacturer. Kantorovich (1939) treats freight transportation, while Chachra, Ghare and Moore (1979) formulate a highway route planning problem.

The assignment problem, as a subclass of the above problem, is very widely applicable. A variety of allocation problems, involving contracts, employees, workgroups, machines, work-locations and work-crews are discussed by Chachra, Ghare and Moore (1979).

The maximum flow problem can also be solved using the MCF framework, although it is usually solved by a more specialised algorithm. Chachra, Ghare and Moore (1979) describe a formulation of this type applied to the solution of a petroleum piping problem for the Universal Oil Company.

Vehicular traffic problems on road networks are a class of problems whose network structure is readily apparent. For example, Roberts (1978) treats the problem of implementing a

one-way system to increase the efficiency of a city street-plan. Chachra, Ghare and Moore (1979) expand on the 'traffic' class of problems to include situations involving the movement of people, mail, factory parts and even court cases.

Network flow formulations also provide a very convenient method of modelling production planning and scheduling problems. Wilson and Beineke (1978) examine how production to meet varying demand can be scheduled, and also examine the similar problem of scheduling a number of related activities required to complete a large task. Kantorovich (1939) considers the distribution of the processing of various items among a number of machines in order to maximumise total output. Chachra, Ghare and Moore (1979) treat airline and railroad scheduling, as well as a rather idiosyncratic model called the 'Fisherman's Problem'. The object here is to transport three items across a river, but certain combinations of items may not be left together on the same bank. The 'items' in question were a wolf, a goat, and a cabbage.

The 'Warehousing Problem' is that faced by an entrepreneur who seeks to maximise the profit from storing a variety of goods, prior to their sale, in a warehouse of fixed capacity. This is treated as a network problem by Ford and Fulkerson (1962). The authors also describe the 'Caterer Problem', which is to be found in a number of Linear Programming texts, including Murty(1976) and Taha(1987). In this example, the owner of a restaurant seeks to minimize the cost of meeting his daily requirement of fresh napkins, with the option of purchasing new ones, or by availing of a one-day or two-day laundering service, all at varying costs.

The field of chemistry is a fertile one for the application of graph theory and network models. Applications in this field are described by Wilson and Beineke (1979) and in Balaban (1976). Electrical networks and their analysis are the subjects of Minty's (1960) paper. The author includes an independently devised Out-of-Kilter algorithm, which aided him in his work. Rockafellar (1984) deals with a range of hydraulic and mechanical applications of network flows, including a city water supply system involving pump-stations and aquaducts.

5.3.2 An Irish Application

In the course of our research, we have had the opportunity to contribute to the solution of a distribution problem for an large Irish company in the agri-business sector. The company currently supplies approximately two hundred towns from four large depots. With a view to reducing the cost associated with the distribution of its products, the company is considering the re-organization or re-location of its depots.

As a preliminary step in the formulation of the new distribution policy, we have modelled the existing network in terms of a transportation problem. Thus there are four supply nodes, and approximately two hundred demand nodes in the problem, with arcs representing the shortest road distance linking each depot to each town. At this early stage in the development of the model, the per unit cost of distribution is assumed to be linearly related to the road distance travelled. The preparation of the input data proved straightforward, and we obtained the optimal solution (i.e. the minimisation of the total distribution cost) in 0.55 seconds, under the same test conditions as those described in section 5.2.1 above. As with the randomly generated test set, the performance of Arc Block, Arc Sample and Two Phase pricing strategies far outstripped the performance of the other methods. A block/sample of

size 40 was used by all methods, with the Two Phase method increasing this to 60 in the second phase of solution.

We hope that our implementation of NSM will contribute further to the development of an improved distribution policy for the company in question, as more accurate data is obtained, and the network model is refined.

Chapter 6

Conclusions

From the foregoing discussion, we draw the following conclusions.

1. The Minimum Cost Flow Problem has a rich history, and it forms a framework upon which a wide range of theoretical and practical problems may be formulated and solved.
2. There exist, at present, three dominant solution paradigms, namely the Network Simplex Method, Relaxation algorithms and Scaling algorithms. The superiority of any one method is a highly contentious issue.
3. The area of PC-based implementations of NSM is under-researched, but we believe that such implementations may offer a real alternative to the large scale codes now prevalent
4. Simple pricing strategies, in general, out-perform their more complicated counterparts, mainly due to the elimination of expensive computational overhead.
5. The entire field of network flow theory and practice offers a plethora of fertile research opportunities, exemplified by the dedication of the First DIMACS Implementation Challenge to the development of more efficient methods for the solution of the Minimum Cost Flow Problem.

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1983) *Data Structures and Algorithms* (Addison-Wesley Publishing Company).
2. Ahuja, R.K., Magnanti, T.L., and Orlin, J.B. (1989) 'Network Flows' in *Handbooks in Operations Research and Management Science* Volume 1 edited by G.L. Nemhauser (Elsevier Science Publications B.V., North Holland).
3. Armstrong, R.D., Klingman, D., and Whitman, D. (1980) 'Implementation and Analysis of a Variant of the Dual Method for the Capacitated Transshipment Problem' *European Journal of Operational Research* Volume 4 p403-420.
4. Balaban, A.T. (1976) 'Chemical Applications of Graph Theory' (Academic Press).
5. Barr, R.S., Glover, F., and Klingman, D. (1977) 'The Alternating Basis Algorithm for Assignment Problems' *Mathematical Programming* Volume 13 p1-13.
6. Bazarra, M.S., Jarvis, J.J., and Sherali, H.D. (1977) *Linear Programming and Network Flows* (John Wiley & Sons).
7. Bertsekas, D.P. (1985) 'A Unified Framework for Primal Dual Methods in Minimum Cost Network Flow Problems' *Mathematical Programming* Volume 32 p125-145.
8. Bertsekas, D.P., and Eckstein, J. (1988) 'Dual Coordinate Step Methods for Linear Network Flow Problems' *Mathematical Programming* Series B Volume 42 p203-243.
9. Bertsekas, D.P., and Tseng, P. (1988) 'Relaxation Methods for Minimum Cost Ordinary and Generalised Network Flow Problems' *Operations Research* Volume 36 p93-114.
10. Bradley, G., Brown, G., and Graves, G. (1977) 'Design and Implementation of Large Scale Primal Transshipment Algorithms' *Management Science* Volume 24 p1-38.
11. Chachra, V., Ghare, P.M., and Moore, J.M. (1979) *Applications of Graph Theory Algorithms* (North Holland Inc., New York).
12. Chvátal, V. (1983) *Linear Programming* (W.H. Freeman and Company, New York, San Francisco).
13. Cunningham, W.H. (1976) 'A Network Simplex Method' *Mathematical Programming* Volume 11 p105-116.
14. Cunningham, W.H. (1979) 'Theoretical Properties of the Network Simplex Method' *Mathematics of Operations Research* Volume 4 p196-208.
15. Dantzig, G.B. (1963) *Linear Programming and Extensions* (Princeton University Press, Princeton, New Jersey).
16. Edmonds, J. and Karp, R.M. (1972) 'Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems' *Association for Computing Machinery Journal* Volume 19 p248-264.
17. Ford, L.R., and Fulkerson, D.R. (1962) *Flows in Networks* (Princeton University Press, Princeton, New Jersey).
18. Gavish, B., Schweitzer, P., and Shlifer, E. (1977) 'The Zero-Pivot Phenomenon in Transportation and Assignment Problems and its Computational Implications' *Mathematical Programming* Volume 12 p226-240.
19. Glover, F., Karney, D., and Napier, A. (1974) 'A Computational Study on Start Procedures, Basis Change Criteria, and Solution Algorithms for Transportation Problems' *Management Science* Volume 20 p793-813.

20. Glover, F., and Klingman, D. (1977) 'Network Applications in Industry and Government' *AIEE Transactions* Volume 9 p363-376.
21. Goldfarb, D., Reid, J.K. (1977) 'A Practicable Steepest Edge Simplex Algorithm' *Mathematical Programming* Volume 12 p361-371.
22. Grigoriadis, M.D. (1986) 'An Efficient Implementation of the Network Simplex Method' *Mathematical Programming Study* Volume 26 p83-111.
23. Helgason, R.V., and Kennington, J.C. (1977) 'An Efficient Procedure for Implementing a Dual Simplex Network Flow Algorithm' *AIEE Transactions* Volume 9 p63-68.
24. Hitchcock, F.C. (1941) 'The Distribution of a Product from Several Sources to Numerous Localities' *Journal of Maths Physics* Volume 20 p224-230.
25. Kantorovich, L.V. (1939) 'Mathematical Methods in the Organization and Planning of Production' translated in *Management Science* (1960) Volume 6 p336-422.
26. Klein, M. (1967) 'A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems' *Management Science* Volume 14, p205-220.
27. Klingman, D., Napier, A., and Stutz, J. (1974) 'NETGEN: A Program for Generating Large Scale Capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems' *Management Science* Volume 20 p814-821.
28. Knuth, D.E. (1973) *The Art of Computer Programming* Volume 1 (Addison Wesley Publishing Company).
29. Minty, G.J. (1960) 'Monotone Networks' *Royal Society Proceedings – London* Series A Volume 257 p194-212.
30. Mulvey, J.M. (1978) 'Pivot Strategies for Primal Simplex Network Codes' *Association for Computing Machinery Journal* Volume 25 p266-270.
31. Murty, K.G. (1976) *Linear and Combinatorial Programming* (John Wiley & Sons).
32. O'Connor, D.R. (1980) 'Fast Primal Transportation Algorithms' *Working Papers.*
33. Orden, A. (1956) 'The Transhipment Problem' *Management Science* Volume 2 p276-285.
34. Roberts, F.S. (1978) *Graph Theory and its Applications to Problems of Society* (Society for Industrial & Applied Mathematics, Philadelphia, Pennsylvania).
35. Rockafellar, R.T. (1984) *Network Flows and Monotropic Optimization* (John Wiley & Sons).
36. Srinivasan, V., and Thompson, G.L. (1973) 'Benefit-Cost Analysis of Coding Techniques for the Primal Transportation Algorithm' *Association for Computing Machinery Journal* Volume 20 p194-213.
37. Taha, H.A. (1987) *Operations Research* (MacMillan Publishing Co., New York, Collier MacMillan Publishers, London).
38. Tarjan, R.E. (1983) *Data Structures and Network Algorithms* (Society for Industrial & Applied Mathematics, Philadelphia, Pennsylvania).
39. Weintraub, A. (1974) 'A Primal Algorithm to Solve Network Flow Problems with Convex Costs' *Management Science* Volume 21 p87-97.
40. Wilson, R.J., and Beineke, L.W. (1979) *Applications of Graph Theory* (Academic Press).

Appendix A

Test Problem Specifications

Appendix B
Pricing Rule Parameters

Appendix C
Summarised Test Results

Appendix D
Detailed Test Output