4

Interior point algorithms for network flow problems

Mauricio G.C. Resende

AT&T Bell Laboratories, Murray Hill, NJ 07974-2070 USA

Panos M. Pardalos

The University of Florida, Gainesville, FL 32611-6595 USA

1 Introduction

A large number of problems in transportation, communications and manufacturing can be modelled as network flow problems. In these problems one seeks to find the most efficient, or optimal, way to move flow (e.g. materials, information, buses, electrical currents) on a network (e.g. postal network, computer network, transportation grid, power grid). Many of these optimization problems are special classes of linear programming problems, with combinatorial properties that allow the development of efficient solution techniques. In this chapter, we limit our discussion to linear network flow problems. For a treatment of non-linear network flow problems, the reader is referred to [17, 28, 29, 48].

Given a directed graph $G = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N}$ is a set of m nodes and $\mathcal{A}$ a set of n arcs, let (i, j) denote a directed arc from node i to node j. Every node is classified in one of the following three categories. *Source* nodes produce more flow than they consume. *Sink* nodes consume more flow than they produce. *Transshipment* nodes produce as much flow as they consume. Without loss of generality, one can assume that the total flow produced in the network equals the total flow consumed. Each arc has associated with it an origin node and a destination node, implying a direction for flow to follow. Arcs have limitations (often called capacities or bounds) on how much flow can move through them. The flow on arc (i, j) must be no less than l_{ij} and can be no greater than u_{ij}. To set up the problem in the framework of an optimization problem, a unit flow cost c_{ij}, incurred by each unit of flow moving through arc (i, j), must be defined. Besides being restricted by lower and upper bounds at each arc, flows must satisfy another important condition, known as Kirchhoff's Law (conservation of flow), which states that for every node in the network, the sum of all incoming flow plus the flow produced at the node must equal the sum of all outgoing flow plus the flow consumed at the node. The objective of the *minimum cost network flow problem* is to determine the flow on each arc of the network, such that all of the flow produced in the network is moved from the source nodes to the sink nodes in the most cost-effective way, whilst not violating Kirchhoff's Law and the flow limitations on the arcs. The minimum cost network flow problem can be formulated as the following linear program:

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}, \tag{1.1}$$

$$\text{subject to} \quad \sum_{(j,k)\in\mathcal{A}} x_{jk} - \sum_{(k,j)\in\mathcal{A}} x_{kj} = b_j, \;\; j \in \mathcal{N}, \tag{1.2}$$

$$l_{ij} \le x_{ij} \le u_{ij}, \;\; (i,j) \in \mathcal{A}. \tag{1.3}$$

In this formulation, x_{ij} denotes the flow on arc (i,j) and c_{ij} is the cost of transporting one unit of flow on arc (i,j). For each node $j \in \mathcal{N}$, let b_j denote a quantity associated with node j that indicates how much flow is produced or consumed at the node. If $b_j > 0$, node j is a source. If $b_j < 0$, node j is a sink. Otherwise ($b_j = 0$), node j is a transshipment node. For each arc $(i,j) \in \mathcal{A}$, as before, let l_{ij} and u_{ij} denote, respectively, the lower and upper bounds on flow on arc (i,j). The case where $u_{ij} = \infty$, for all $(i,j) \in \mathcal{A}$, gives rise to the *uncapacitated* network flow problem. Without loss of generality, l_{ij} can be set to zero. Most often, the problem data (i.e. c_{ij}, u_{ij}, l_{ij}, for $(i,j) \in \mathcal{A}$ and b_j, for $j \in \mathcal{N}$) are assumed to be integer, and many software codes adopt this assumption. However, there can exist applications where the data are real numbers, and algorithms should be capable of handling problems with real data.

Constraints of type (1.2) are referred to as the flow conservation equations, while constraints of type (1.3) are called the flow capacity constraints. In matrix notation, the above network flow problem can be formulated as a linear program of the special form

$$\min \{c^\top x \mid Ax = b, \; l \le x \le u\},$$

where A is the $m \times n$ *node-arc incidence matrix* of the graph $G = (\mathcal{N}, \mathcal{A})$, i.e. for each arc (i,j) in $\mathcal{A}$ there is an associated column in matrix A with exactly two non-zero entries: an entry 1 in row i and an entry -1 in row j. Note that of the mn entries of A, only $2n$ are non-zero and because of this the node-arc incidence matrix is not a space-efficient representation of the network. There are many other ways to represent a network. A popular representation is the *node-node adjacency* matrix B. This is an $m \times m$ matrix with an entry 1 in position (i,j) if arc $(i,j) \in \mathcal{A}$ and 0 otherwise. Such a representation is efficient for dense networks, but is inefficient for sparse networks. A more efficient representation for sparse networks is the *adjacency list*, where for each node $i \in \mathcal{N}$ there exists a list of arcs emanating from node i, i.e. a list of nodes j such that $(i,j) \in \mathcal{A}$. The *forward star* representation is a multi-array implementation of the adjacency list data structure. The adjacency list allows easy access to the arcs emanating from a given node, but no easy access to the incoming arcs. The *reverse star* representation allows easy access to the list of arcs incoming into i. Another representation that is much used in interior point network flow implementations is a simple *arc list*, where the arcs are stored in a linear array. The complexity of an algorithm for solving network flow problems depends greatly on the network representation and the data structures used for maintaining and updating intermediate computations.

We denote the i-th column of A by A_i, the i-th row of A by $A_{.i}$ and a submatrix of A formed by columns with indices in set S by A_S. If graph G is disconnected and has p connected components, there are exactly p redundant flow conservation constraints, which are sometimes removed from the problem formulation. We rule out a trivially infeasible problem by assuming

$$\sum_{j \in \mathcal{N}^k} b_j = 0, \quad k = 1, \ldots, p, \tag{1.4}$$

where $\mathcal{N}^k$ is the set of nodes for the k-th component of G.

Often it is further required that the flow x_{ij} be integer, i.e. we replace (1.3) by

$$l_{ij} \le x_{ij} \le u_{ij}, \ x_{ij} \text{ integer}, \ (i,j) \in \mathcal{A}. \tag{1.5}$$

Since the node-arc incidence matrix A is totally unimodular, when the data is integer all vertex solutions of the linear program are integer. An algorithm that finds a vertex solution, such as the simplex method, will necessarily produce an integer optimal flow. In certain types of network flow problems, such as the assignment problem, one may only be interested in solutions having integer flows, since fractional flows do not have a logical interpretation.

In the remainder of this chapter we assume, without loss of generality, that $l_{ij} = 0$ for all $(i,j) \in \mathcal{A}$ and that $c \neq 0$. A simple change of variables can transform the original problem into an equivalent one with $l_{ij} = 0$ for all $(i,j) \in \mathcal{A}$. The case where $c = 0$ is a simple feasibility problem, and can be handled by solving a maximum flow problem [3].

Many important combinatorial optimization problems are special cases of the minimum cost network flow problem. Such problems include the linear assignment and transportation problems, and the maximum flow and shortest path problems. In the transportation problem, the underlying graph is bipartite, i.e. there exist two sets $\mathcal{S}$ and $\mathcal{T}$ such that $\mathcal{S} \cup \mathcal{T} = \mathcal{N}$ and $\mathcal{S} \cap \mathcal{T} = \emptyset$ and arcs occur only from nodes of $\mathcal{S}$ to nodes of $\mathcal{T}$. The set $\mathcal{S}$ is usually called the set of source nodes and the set $\mathcal{T}$ the set of sink nodes. For the transportation problem, the right-hand side vector in (1.2) is given by

$$b_j = \begin{cases} s_j, & \text{if } j \in \mathcal{S}, \\ -t_j, & \text{if } j \in \mathcal{T}, \end{cases}$$

where s_j is the supply at node $j \in \mathcal{S}$ and t_j is the demand at node $j \in \mathcal{T}$. The assignment problem is a special case of the transportation problem, in which $s_j = 1$ for all $j \in \mathcal{S}$ and $t_j = 1$ for all $j \in \mathcal{T}$.

The computation of the maximum flow from node s to node t in $G = (\mathcal{N}, \mathcal{A})$ can be done by computing a minimum cost flow in $G' = (\mathcal{N}', \mathcal{A}')$, where $\mathcal{N}' = \mathcal{N}$

and $\mathcal{A}' = \mathcal{A} \cup (t,s)$, where

$$c_{ij} = \begin{cases} 0, & \text{if } (i,j) \in \mathcal{A}, \\ -1, & \text{if } (i,j) = (t,s), \end{cases}$$

and

$$u_{ij} = \begin{cases} \mathrm{cap}(i,j), & \text{if } (i,j) \in \mathcal{A}, \\ \infty, & \text{if } (i,j) = (t,s), \end{cases}$$

where $\mathrm{cap}(i,j)$ is the capacity of arc (i,j) in the maximum flow problem.

The shortest paths from node s to all nodes in $\mathcal{N} \setminus \{s\}$ can be computed by solving an uncapacitated minimum cost network flow problem in which c_{ij} is the length of arc (i,j) and the right-hand side vector in (1.2) is given by

$$b_j = \begin{cases} m-1, & \text{if } j = s, \\ -1, & \text{if } j \in \mathcal{N} \setminus \{s\}. \end{cases}$$

Although all of the above combinatorial optimization problems are formulated as minimum cost network flow problems, several specialized algorithms have been devised for solving them efficiently.

In many practical applications, flows in networks with more than one commodity need to be optimized. In the multicommodity network flow problem, k commodities are to be moved in the network. The set of commodities is denoted by $\mathcal{K}$. Let x_{ij}^k denote the flow of commodity k in arc (i,j). The multicommodity network flow problem can be formulated as the following linear program:

$$\begin{aligned}
\text{minimize} \quad & \textstyle\sum_{k\in\mathcal{K}} \sum_{(i,j)\in\mathcal{A}} c_{ij}^k x_{ij}^k, && (1.6)\\
\text{subject to} \quad & \textstyle\sum_{(j,l)\in\mathcal{A}} x_{jl}^k - \sum_{(l,j)\in\mathcal{A}} x_{lj}^k = b_j^k, \quad j \in \mathcal{N}, \quad k \in \mathcal{K}, && (1.7)\\
& \textstyle\sum_{k\in\mathcal{K}} x_{ij}^k \le u_{ij}, \quad (i,j) \in \mathcal{A}, && (1.8)\\
& x_{ij}^k \ge 0, \quad (i,j) \in \mathcal{A}, \quad k \in \mathcal{K}. && (1.9)
\end{aligned}$$

The minimum cost network flow problem is a special case of the multicommodity network flow problem, in which there is only one commodity.

In the 1940s, Hitchcock [30] proposed an algorithm for solving the transportation problem and later Dantzig [14] developed the simplex method for linear programming problems. In the 1950s, Kruskal [41] developed a minimum spanning tree algorithm and Prim [51] devised an algorithm for the shortest path problem. During that decade, commercial digital computers were introduced widely. The first book on network flows was published by Ford and Fulkerson [19] in 1962. Since then, active research has produced a variety of algorithms, data structures and software for solving network flow problems. For an introduction to network flow problems and applications, see the books [3, 7, 17, 19, 39, 42, 58, 62].

The focus of this chapter is on recent computational approaches, based on interior point methods, for solving large-scale network flow problems. In the last two decades, many other approaches have been developed. A history of computational approaches up to 1977 is summarized in [11]. In addition, several computational studies had established the fact that, for network flow problems, specialized network simplex algorithms were orders of magnitude faster than simply applying a general simplex linear programming code. (See, e.g. the studies in [23, 27, 39, 43].) A collection of FORTRAN codes of efficient algorithms of that period can be found in [59]. Another important class of network optimization algorithms and codes are the relaxation methods described in [8]. More recent computational research is included in [25, 31].

In 1984, Karmarkar introduced a new polynomial time algorithm for solving linear programming problems [37]. This algorithm and many of its variants, known as interior point methods, have been used to efficiently solve network flow problems. This is the main topic of the remainder of this chapter.

The chapter is organized as follows. In Section 2 we provide a brief survey of the literature in the field of interior point network flow methods. A discussion of complexity issues is made. In Section 3 we focus on several important components of interior point network flow implementations, including a discussion on iterative methods for solving the large linear systems that occur in interior point methods, preconditioners, identification of the optimal partition, and recovery of the optimal flow. These points are illustrated on a particular interior point method, the dual affine scaling algorithm. In Section 4 we present some computational results, comparing an interior point network flow method with an efficient, commercially available, network simplex code. Concluding remarks are made in Section 5.

2 Implementation of interior point network flow methods

In this section we present issues related to efficient implementation of interior point methods for solving network flow problems. We present a brief review of the research literature relating to interior point network flow methods and discuss the computational complexity of interior point network flow methods.

2.1 Literature review

After the introduction of Karmarkar's algorithm in 1984, many groups of researchers hurried to implement the method. One of the first implementations is described in Adler *et al.* [1, 2]. They implemented what is now known as the dual affine scaling algorithm using direct factorization for solving, at each iteration, a direction-finding system of linear equations, of the form

$$ADA^{\top} u = t, \tag{2.1}$$

where A is an $m \times n$ constraint matrix, D is a diagonal $n \times n$ scaling matrix, and u and t are m-vectors. Though that implementation was shown to compare favourably with an efficient implementation of the simplex method [44] on general linear programming problems from the test problem set NETLIB [20], the code was orders of magnitude slower than the network simplex implementation NETFLO [39] on small assignment problems.