Linear Network Optimization: Algorithms and Codes

Linear Network Optimization: Algorithms and Codes

DIMITRI P. BERTSEKAS

The MIT Press
Cambridge, Massachusetts
London, England

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Bertsekas, Dimitri P.
Linear network optimization : algorithms and codes / Dimitri P. Bertsekas.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-02334-2
1. Network analysis (Planning). 2. Mathematical optimization. 3. Algorithms. I. Title.
T57.85.B43 1991
658.4'032–dc20 91-27588
CIP

Preface

Linear network optimization problems, such as shortest path, assignment, max-flow, transportation, and transhipment, are undoubtedly the most common optimization problems in practice. Extremely large problems of this type, involving thousands and even millions of variables, can now be solved routinely, thanks to recent algorithmic and technological advances. On the theoretical side, despite their relative simplicity, linear network problems embody a rich structure with both a continuous and a combinatorial character. Significantly, network ideas have been the starting point for important developments in linear and nonlinear programming, as well as combinatorial optimization.

Up to the late seventies, there were basically two types of algorithms for linear network optimization: the *simplex* method and its variations, and the *primal-dual* method and its close relative, the *out-of-kilter* method. There was some controversy regarding the relative merit of these methods, but thanks to the development of efficient implementation ideas, the simplex method emerged as the fastest of the two for most types of network problems.

A number of algorithmic developments in the eighties have changed significantly the situation. New methods were invented that challenged the old ones, both in terms of practical efficiency and theoretical worst-case performance. Two of these methods, originally proposed by the author, called *relaxation* and *auction*, will receive a lot of attention in this book. The relaxation method is a dual ascent method resembling the coordinate ascent method of unconstrained nonlinear optimization that significantly outperforms in practice both the simplex and the primal-dual methods for many types of problems. Auction is a form of dual coordinate ascent method, based on the notion of ϵ-complementary slackness and scaling ideas. This algorithm, together with its extensions, has excellent computational complexity, which is superior to that of the classical methods for many types of problems. Some auction algorithms have also proved to be very effective in practice, particularly for assignment and max-flow problems.

One of the purposes of the book is to provide a modern and up-to-date synthesis of old and new algorithms for linear network flow problems. The coverage is focused and selective, concentrating on the algorithms that have proved most successful in practice or otherwise embody important methodological ideas. Two fundamental ideas of mathematical programming are emphasized: *duality* and *iterative cost improvement*. Algorithms are grouped in three categories: (a) *primal cost improvement methods, including simplex methods*, which iteratively improve the primal cost by moving flow around simple cycles, (b) *dual ascent methods*, which iteratively improve the dual cost by changing the prices of a subset of nodes by equal amounts, and (c) *auction algorithms*, which try to improve the dual cost approximately along coordinate directions.

The first two classes of methods are dual to each other when viewed in the context of Rockafellar's monotropic programming theory [Roc84]; they are based on cost improvement along elementary directions of the circulation space (in the primal case) or the differential space (in the dual case). Auction algorithms are fundamentally different; they have their origin in nondifferentiable optimization and the ϵ-subgradient method in particular [BeM73].

A separate chapter is devoted to each of the above types of methods. The introductory chapter establishes some basic material and treats a few simple problems such as max-flow and shortest path. A final chapter discusses some of the practical performance aspects of the various methods.

A second purpose of the book is to supply state-of-the-art FORTRAN codes based on some of the algorithms presented. These codes illustrate implementation techniques commonly used in network optimization and should be helpful to practitioners. The listings of the codes appear in appendixes at the end of the book, and are also available on diskette from the author. I am thankful to Giorgio Gallo and Stefano Pallotino who gave me permission to include two of their shortest path codes.

The book can be used for a course on network optimization or for part of a course on introductory optimization; such courses have flourished in engineering, operations research, and applied mathematics curricula. The book contains a large number of examples and exercises, which should enhance its suitability for classroom instruction.

I was fortunate to have several outstanding collaborators in my linear network optimization research, and I would like to mention those with whom I have worked extensively. Eli Gafni programmed for the first time the auction algorithm and the relaxation method for assignment problems in 1979 and assisted with the computational experimentation. The idea of ϵ-scaling arose during my interactions with Eli at that time. Paul Tseng worked with me on network optimization starting in 1982. Together we developed the RELAX codes, we developed several extensions to the basic relaxation method and we collaborated closely and extensively on a broad variety of other subjects. Paul also read a substantial part of the book, and offered several helpful suggestions. Jon Eckstein worked with me on auction and other types of network optimization algorithms starting in 1986. Jon made several contributions to the theory of the ϵ-relaxation method, and coded its first implementation. Jon also proofread parts of the book, and his comments resulted in several substantive improvements. David Castañon has been working with me on auction algorithms for assignment, transportation, and minimum cost flow problems since 1987. Much of our joint work on these subjects appears in Chapter 4, particularly in Sections 4.2 and 4.4. David and I have also collaborated extensively on the implementation of various network flow algorithms. Our interactions have resulted in several improvements in the codes of the appendixes.

Funding for the research relating to this book was provided by the National Science Foundation and by the Army Research Office through the Center for Intelligent Control Systems at MIT. The staff of MIT Press worked with professionalism to produce the book quickly and efficiently.

Contents

1

Introduction

1.1 PROBLEM FORMULATION

This book deals with a single type of network optimization problem with linear cost, known as the *transshipment* or *minimum cost flow* problem. In this section, we formulate this problem together with several special cases. One of the most important special cases is the *assignment problem*, which we will discuss in detail because it is simple and yet captures most of the important algorithmic aspects of the general problem.

Example 1.1. The Assignment Problem

Suppose that there are n persons and n objects that we have to match on a one-to-one basis. There is a benefit or value a_{ij} for matching person i with object j, and we want to assign persons to objects so as to maximize the total benefit. There is also a restriction that person i can be assigned to object j only if (i, j) belongs to a set of given pairs $\mathcal{A}$. Mathematically, we want to find a set of person-object pairs $(1, j_1), \ldots, (n, j_n)$ from $\mathcal{A}$ such that the objects $j_1, \ldots, j_n$ are all distinct, and the total benefit $\sum_{i=1}^{n} a_{ij_i}$ is maximized.

The assignment problem is important in many practical contexts. The most obvious ones are resource allocation problems, such as assigning employees to jobs, machines to tasks, etc. There are also situations where the assignment problem appears as a subproblem in various methods for solving more complex problems.

We may associate any assignment with the set of variables $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$, where $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise. We may then formulate the assignment problem as the linear program

$$\begin{aligned} &\text{maximize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\ &\text{subject to} \\ &\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\, i = 1,\ldots,n, \\ &\sum_{\{i\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1, \qquad \forall\, j = 1,\ldots,n, \\ &0 \le x_{ij} \le 1, \qquad \forall\, (i,j) \in \mathcal{A}. \end{aligned} \tag{1.1}$$

Actually we should further restrict x_{ij} to be either 0 or 1; however, as we will show in the next chapter, the above linear program has a remarkable property: if it has a feasible solution at all, then it has an optimal solution where all x_{ij} are either 0 or 1. In fact, the set of its optimal solutions includes all the optimal assignments.

Another important property of the assignment problem is that it can be represented by a graph as shown in Fig. 1.1. Here, there are $2n$ nodes divided into two groups: n corresponding to persons and n corresponding to objects. Also, for every $(i,j) \in \mathcal{A}$, there is an arc connecting person i with object j. In the terminology of network problems, the variable x_{ij} is referred to as the *flow* of arc (i,j). The constraint $\sum_{\{j\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1$ indicates that the total outgoing flow from node i should be equal to 1, which may be viewed as the (exogenous) *supply* of the node. Similarly, the constraint $\sum_{\{i\mid(i,j)\in\mathcal{A}\}} x_{ij} = 1$ indicates that the total incoming flow to node j should be equal to 1, which may be viewed as the (exogenous) *demand* of the node.

Before we can proceed with a formulation of more general network flow problems we must introduce some notation and terminology.

1.1.1 Graphs and Flows

We define a *directed graph*, $\mathcal{G} = (\mathcal{N},\mathcal{A})$, to be a set $\mathcal{N}$ of *nodes* and a set $\mathcal{A}$ of pairs of distinct nodes from $\mathcal{N}$ called *arcs*. The numbers of nodes and arcs of $\mathcal{G}$ are denoted by N and A, respectively, and we assume throughout that $1 \le N < \infty$ and $0 \le A < \infty$. An arc (i,j) is viewed as an ordered pair, and is to be distinguished from the pair (j,i). If (i,j) is an arc, we say that (i,j) is *outgoing* from node i and *incoming* to node j; we also say that j is an *outward neighbor* of i and that i is an *inward neighbor* of j. We say that arc (i,j) is *incident* to i and to j, and that i is the *start* node and j is the

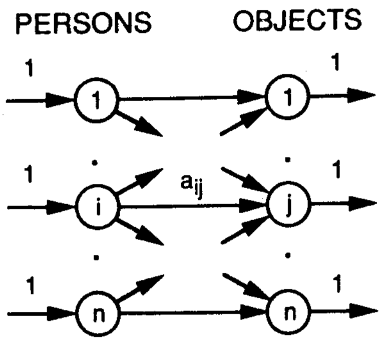


Figure 1.1 The graph representation of an assignment problem.

end node of the arc. The *degree* of a node i is the number of arcs that are incident to i.

A graph is said to be *bipartite* if its nodes can be partitioned into two sets $\mathcal{S}$ and $\mathcal{T}$ such that every arc has its start in $\mathcal{S}$ and its end in $\mathcal{T}$. The assignment graph of Fig. 1.1 is an example of a bipartite graph, with $\mathcal{S}$ and $\mathcal{T}$ being the sets of persons and objects, respectively.

We do not exclude the possibility that there is a separate arc connecting a pair of nodes in each of the two directions. However, we do not allow more than one arc between a pair of nodes in the same direction, so that we can refer unambiguously to the arc with start i and end j as arc (i,j). This was done for notational convenience. Our analysis can be simply extended to handle multiple arcs with start i and end j; the extension is based on modifying the graph by introducing for each such arc, an additional node, call it n, together with the two arcs (i,n) and (n,j). The codes in the appendixes can handle graphs that have multiple arcs between any pair of nodes in the same direction, without the above modification.

Paths and Cycles

A *path* P in a directed graph is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ with $k \geq 2$ and a corresponding sequence of $k-1$ arcs such that the ith arc in the sequence is either (n_i, n_{i+1}) (in which case it is called a *forward* arc of the path) or (n_{i+1}, n_i) (in which case it is called a *backward* arc of the path). A path is said to be *forward* (or *backward*) if all of its arcs are forward (respectively, backward) arcs. We denote by P^+ and P^- the sets of forward and backward arcs of P, respectively. Nodes n_1 and n_k are called the *start node* (or *origin*) and the *end node* (or *destination*) of P, respectively.

A *cycle* is a path for which the start and end nodes are the same. A path is said to be *simple* if it contains no repeated arcs and no repeated nodes, except that the start and end nodes could be the same (in which case the path is called a *simple cycle*). These definitions are illustrated in Fig. 1.2.

Note that the sequence of nodes $(n_1, n_2, \ldots, n_k)$ is not sufficient to specify a path; the sequence of arcs is also important, as Fig. 1.2(c) shows. The difficulty arises when for two successive nodes n_i and n_{i+1} of the path, both (n_i, n_{i+1}) and (n_{i+1}, n_i) are arcs, so there is ambiguity as to which of the two is the corresponding arc of the path. However, when the path is known to be forward or is known to be backward, it is uniquely specified by the sequence of its nodes. Throughout the book, we will make sure that the intended sequence of arcs is explicitly defined in ambiguous situations.

A graph that contains no simple cycles is said to be *acyclic*. A graph is said to be *connected* if for each pair of nodes i and j, there is a path starting at i and ending at j; it is said to be *strongly connected* if for each pair of nodes i and j, there is a forward path starting at i and ending at j. For example, the assignment graph of Fig. 1.1 may be connected but cannot be strongly connected.

We say that $\mathcal{G}' = (\mathcal{N}', \mathcal{A}')$ is a *subgraph* of $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ if $\mathcal{G}'$ is a graph, $\mathcal{N}' \subset \mathcal{N}$, and $\mathcal{A}' \subset \mathcal{A}$. A *tree* is a connected acyclic graph. A *spanning tree* of a graph $\mathcal{G}$ is a subgraph of $\mathcal{G}$ that is a tree and that includes all the nodes of $\mathcal{G}$.

Flow and Divergence

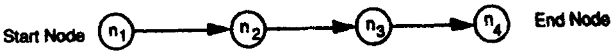
A *flow vector* x in a graph $(\mathcal{N}, \mathcal{A})$ is a set of scalars $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$. We refer to x_{ij} as the flow of the arc (i,j), and we place no restriction (such as nonnegativity) on its value. The *divergence vector* y associated with a flow vector x is the N-dimensional vector with coordinates

$$y_i = \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij} - \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji}, \qquad \forall\, i \in \mathcal{N}. \tag{1.2}$$

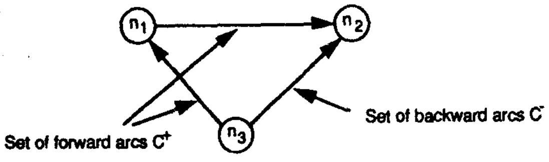
Thus, y_i is the total flow departing from node i less the total flow arriving at i; it is referred to as the divergence of i. For example, an assignment corresponds to a flow vector x with $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise (see Fig. 1.1); the assigned pairs involve each person exactly once and each object exactly once, if the divergence of each person node i is $y_i = 1$, and the divergence of each object node j is $y_j = -1$.

We say that node i is a *source* (respectively, *sink*) for the flow vector x if $y_i > 0$ (respectively, $y_i < 0$). If $y_i = 0$ for all $i \in \mathcal{N}$, then x is called a *circulation*. These definitions are illustrated in Fig. 1.3. Note that by adding Eq. (1.2) over all $i \in \mathcal{N}$, we obtain

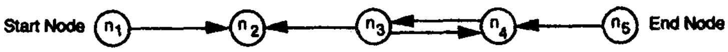
$$\sum_{i \in \mathcal{N}} y_i = 0$$



(a) A simple forward path P = (n_1, n_2, n_3, n_4).
The path P = (n_1, n_2, n_3, n_4 , n_3, n_2, n_3) is also legitimate;
it is not simple, and it is neither forward nor backward.



(b) A simple cycle C = (n_1, n_2, n_3, n_1) which is neither forward nor backward.



(c) Path P = (n_1, n_2, n_3, n_4, n_5) with corresponding sequence of arcs
{ (n_1, n_2), (n_3, n_2), (n_3, n_4), (n_5, n_4) }.

Figure 1.2 Illustration of various types of paths. Note that for the path (c) it is necessary to specify the sequence of arcs of the path (rather than just the sequence of nodes) because both (n_3, n_4) and (n_4, n_3) are arcs. For a somewhat degenerate example that illustrates the fine points of the definitions, note that for the graph of (c), the node sequence

$$C = (n_3, n_4, n_3)$$

is associated with four cycles:

(1) The simple forward cycle with

$$C^+ = \{(n_3, n_4), (n_4, n_3)\}, \qquad C^- : \text{ empty}.$$

(2) The simple backward cycle with

$$C^- = \{(n_4, n_3), (n_3, n_4)\}, \qquad C^+ : \text{ empty}.$$

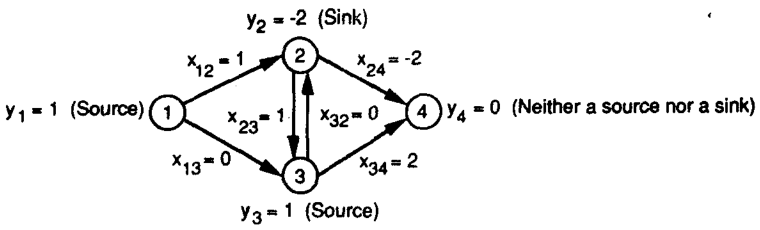
(3) The (nonsimple) cycle with

$$C^+ = \{(n_3, n_4)\}, \qquad C^- = \{(n_3, n_4)\}.$$

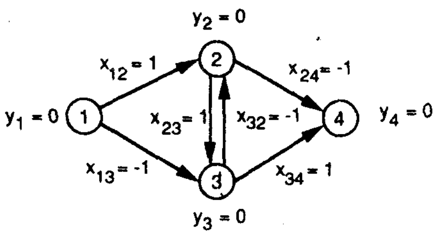
(4) The (nonsimple) cycle with

$$C^+ = \{(n_4, n_3)\}, \qquad C^- = \{(n_4, n_3)\}.$$

Note that the node sequence (n_3, n_4, n_3) determines the cycle uniquely if it is specified that the cycle is either forward or is backward.



(a)



(b) A circulation

Figure 1.3 Illustration of various types of flows. The flow in (b) is a circulation because $y_i = 0$ for all i.

for any divergence vector y.

In applications, a negative arc flow indicates that whatever flow represents (material, electric current, etc.), moves in a direction opposite to the direction of the arc. We can always change the sign of the arc flow to positive as long as we change the arc direction, so in many situations we can assume without loss of generality that all arc flows are nonnegative. For the development of a general methodology, however, this device is often cumbersome, which is why we prefer to simply accept the possibility of negative arc flows.

Conformal Decomposition

It is often convenient to break down a flow vector into the sum of simpler components. A particularly useful decomposition arises when the components involve simple paths and cycles with orientation which is consistent to that of the original flow vector. This leads to the notion of a conformal realization, which we proceed to discuss.

We say that a path P *conforms* to a flow vector x if $x_{ij} > 0$ for all forward arcs (i,j) of P and $x_{ij} < 0$ for all backward arcs (i,j) of P, and furthermore either P is a cycle or else the start and end nodes of P are a source and a sink of x, respectively. Roughly, a path conforms to a flow vector if it "carries flow in the forward direction" – that is, in the direction from the start node to the end node. In particular, for a forward cycle to conform to a flow vector, all its arcs must have positive flow; for a forward path which is not a cycle to conform to a flow vector, its arcs must have positive flow, and in addition the start and end nodes must be a source and a sink, respectively.

A *simple path flow* is a flow vector that corresponds to sending a positive amount of flow along a simple path; more precisely, it is a flow vector x of the form

$$x_{ij} = \begin{cases} a & \text{if } (i,j) \in P^+ \\ -a & \text{if } (i,j) \in P^- \\ 0 & \text{otherwise,} \end{cases} \tag{1.3}$$

where a is a positive scalar, and P^+ and P^- are the sets of forward and backward arcs, respectively, of some simple path P.

We say that a simple path flow x^s *conforms* to a flow vector x if the path P corresponding to x^s via Eq. (1.3) conforms to x. This is equivalent to requiring that

$$0 < x_{ij} \qquad \text{for all arcs } (i,j) \text{ with } 0 < x^s_{ij},$$

$$x_{ij} < 0 \qquad \text{for all arcs } (i,j) \text{ with } x^s_{ij} < 0,$$

and that either P is a cycle or else the divergence (with respect to x) of the start node of P is positive and the divergence (with respect to x) of the end node of P is negative.

We now show that any flow vector can be decomposed into a set of conforming simple path flows. This result, illustrated in Fig. 1.4, turns out to be fundamental for our purposes. The proof is based on an algorithm that can be used to construct the constituent conforming components one by one. Such constructive proofs are often used in network optimization.

Proposition 1.1: (*Conformal Realization Theorem*) A nonzero flow vector x can be decomposed into the sum of t simple path flow vectors $x^1, x^2, \ldots, x^t$ that conform to x, with t being at most equal to the sum of the numbers of arcs and nodes $A + N$. If x is integer, then $x^1, x^2, \ldots, x^t$ can also be chosen to be integer. If x is a circulation, then $x^1, x^2, \ldots, x^t$ can be chosen to be simple circulations, and $t \leq A$.

Proof: We first assume that x is a circulation. Our proof consists of showing how to obtain from x a simple circulation x' conforming to x and such that

$$0 \leq x'_{ij} \leq x_{ij} \qquad \text{for all arcs } (i,j) \text{ with } 0 \leq x_{ij}, \tag{1.4a}$$

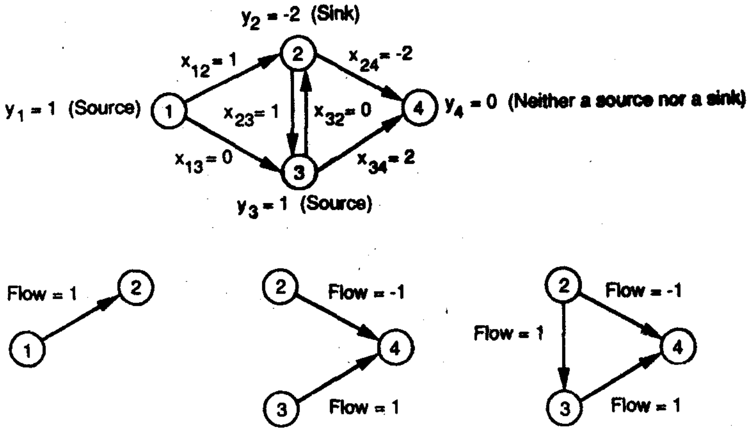


Figure 1.4 Decomposition of a flow vector x into three simple path flows conforming to x. The corresponding simple paths are $(1,2)$, $(3,4,2)$, and $(2,3,4,2)$. The first two are not cycles; they start at a source and end at a sink. Consistent with the definition of conformance of a path flow, each arc (i,j) of these paths carries positive (or negative) flow only if $x_{ij} > 0$ (or $x_{ij} < 0$, respectively). Arcs $(1,3)$ and $(3,2)$ do not belong to any of these paths because they carry zero flow. In this example, the decomposition is unique, but in general this need not be the case.

$$x_{ij} \le x'_{ij} \le 0 \qquad \text{for all arcs } (i,j) \text{ with } x_{ij} \le 0, \tag{1.4b}$$

$$x_{ij} = x'_{ij} \qquad \text{for at least one arc } (i,j) \text{ with } x_{ij} \ne 0. \tag{1.4c}$$

Once this is done, we subtract x' from x. We have $x_{ij} - x'_{ij} > 0$ only for arcs (i,j) with $x_{ij} > 0$, $x_{ij} - x'_{ij} < 0$ only for arcs (i,j) with $x_{ij} < 0$, and $x_{ij} - x'_{ij} = 0$ for at least one arc (i,j) with $x_{ij} \ne 0$. If x is integer, then x' and $x - x'$ will also be integer. We then repeat the process (for at most A times) with the circulation x replaced by the circulation $x - x'$ and so on, until the zero flow is obtained. This is guaranteed to happen eventually because $x - x'$ has at least one more arc with zero flow than x.

We now describe the procedure by which x' with the properties (1.4) is obtained; see Fig. 1.5. Choose an arc (i,j) with $x_{ij} \ne 0$. Assume that $x_{ij} > 0$. (A similar procedure can be used when $x_{ij} < 0$.) Construct a sequence of node subsets $T_0, T_1, \ldots$, as follows: Take $T_0 = \{j\}$. For $k = 0, 1, \ldots$, given T_k, let

$$T_{k+1} = \big\{ n \notin \cup_{p=0}^{k} T_p \mid \text{there is a node } m \in T_k \text{, and either an arc } (m,n) \\ \text{such that } x_{mn} > 0 \text{ or an arc } (n,m) \text{ such that } x_{nm} < 0 \big\},$$

and mark each node $n \in T_{k+1}$ with the label "(m,n)" or "(n,m)," where m is a node of T_k such that $x_{mn} > 0$ or $x_{nm} < 0$, respectively. The procedure terminates when T_{k+1} is empty. We may view T_k as the set of nodes n that can be reached from j with a path of k arcs carrying "positive flow" in the direction from j to n.

We claim that one of the sets T_k contains node i. To see this, consider the set $\cup_k T_k$ of all nodes that belong to one of the sets T_k. By construction, there is no outgoing arc from $\cup_k T_k$ with positive flow and no incoming arc into $\cup_k T_k$ with negative flow. If i did not belong to $\cup_k T_k$, there would exist at least one incoming arc into $\cup_k T_k$ with positive flow, namely the arc (i,j). Thus, the total flow of arcs incoming to $\cup_k T_k$ must be positive, while the total flow of arcs outgoing from $\cup_k T_k$ is negative or zero. On the other hand, these two flows must be equal, since x is a circulation; this can be seen by adding the equation

$$\sum_{\{n|(m,n)\in\mathcal{A}\}} x_{mn} = \sum_{\{n|(n,m)\in\mathcal{A}\}} x_{nm}$$

over all nodes $m \in \cup_k T_k$. Therefore, we obtain a contradiction, and it follows that one of the sets T_k contains node i.

We now trace labels backward from i until node j is reached. [This will happen eventually because if "(m,n)" or "(n,m)" is the label of node n and $n \in T_{k+1}$, then $m \in T_k$, so a "cycle" of labels cannot be formed before reaching j.] In particular, let "(i_1,i)" or "(i,i_1)" be the label of i, let "(i_2,i_1)" or "(i_1,i_2)" be the label of i_1, etc., until a node i_k with label "(i_k,j)" or "(j,i_k)" is found. The cycle $C = (j, i_k, i_{k-1}, \ldots, i_1, i, j)$ is simple, it contains (i,j) as a forward arc, and is such that all its forward arcs have positive flow and all its backward arcs have negative flow (see Fig. 1.2). Let $a = \min_{(m,n)\in C} |x_{mn}| > 0$. Then the circulation x', where

$$x'_{ij} = \begin{cases} a & \text{if } (i,j) \in C^+ \\ -a, & \text{if } (i,j) \in C^- \\ 0 & \text{otherwise,} \end{cases}$$

has the required properties (1.4).

Consider now the case where x is not a circulation. We form an enlarged graph by introducing a new node s and by introducing for each node $i \in \mathcal{N}$ an arc (s,i) with flow x_{si} equal to the divergence y_i of Eq. (1.2). Then (by using also the fact $\sum_{i\in\mathcal{N}} y_i = 0$) the resulting flow vector is seen to be a circulation in the enlarged graph. This circulation, by the result just shown, can be decomposed into at most $A + N$ simple circulations of the enlarged graph, conforming to the flow vector. Out of these circulations, we consider those containing node s, and we remove s and its two incident arcs while leaving the other circulations unchanged. As a result we obtain a set of at most $A + N$ path flows of the original graph, which add up to x. These path flows also conform to x, as is required in order to prove the proposition. **Q.E.D.**