The minimum cost maximum flow problem

Addis Ababa University

Faculty of Computer and Mathematical Science

Department of Mathematics

A project submitted to Department of Mathematics in partial fulfillment of the requirements for the degree of Master of Science in Mathematics.

Prepared by: Dawit Solomon

Advisor: Dr. Berhanu Guta

January 2012

Acknowledgment

Affection, passion and envisage towards the subject Mathematics dissolved in me due to the support of my instructors Dr. **Berhanu Guta**, Dr. **Semu Mitku** and all the instructors at Addis Ababa University. It is, therefore, a great honor to acknowledge my instructors for everything they did for me and yet to be done.

My special gratitude goes to my adviser Dr. **Berhanu Guta**, without whom none of such achievements would have been made. His insightful advices and friendly care made me a better man.

Finally, I am so proud to recognize my family and friends for their contribution and support in one way or another.

In my stay at Addis Ababa University, I have acquired a great deal of dependability, zeal, and knowledge. If it was not for the unimpeachable support of my instructors Dr. **Berhanu Guta**, Dr. **Semu Mitku** and all the instructors at Addis Ababa University, of the these achievements could have been a mere dream. I, therefore, would like to acknowledge my instructors for their support.

I specially want to thank Dr. **Berhanu Guta** for his faithful and friendly approach in juxtapose to the inevitable injection of cognizance.

Finally I am so proud to recognize my family and friends for their contribution and support in one way or another.

Preface

The main objective of this project is to present different approaches to solve minimum cost maximum flow problem, which is one of the problems we face in "network flow problems." The first chapter of this paper gives us basic definition of different terms that we use in the chapters that follows and introduction to different network problems, and the second chapter shows how to solve maximum flow problems and since we also need the knowledge to solve minimum cost flow problem we present it in chapter three. After we presented the pre-requisite in three chapters our main problem of interest comes in chapter four which is the minimum cost maximum flow problem. It deals with finding the minimum possible cost of the maximum flow in the given network.

Contents

1.Introduction 5

Networks 5

Network Flows 6

Basic Definitions 7

Network Flow Problems 13

2.Maximum Flow Problem 20

Notation and Assumptions 22

APPLICATIONS 23

FLOWS AND CUTS 28

GENERIC AUGMENTING PATH ALGORITHM 32

Algorithm augmenting path 32

LABELING ALGORITHM AND THE MAX-FLOW MIN-CUT THEOREM 33

Maximal Flow Algorithm 35

3.Minimum Cost Flow problems 36

Residual Network 38

Node potential 39

Reduced Cost Optimality Conditions 39

CYCLE-CANCELING ALGORITHM AND THE 41

INTEGRALITY PROPERTY 41

OUT-OF-KILTER ALGORITHM 42

Primal Simplex Algorithm for the Minimal Cost Flow Problem 48

4.The minimum cost maximum flow problem 52

The min-cost circulation 51

Chapter 1

Introduction

Networks

A directed network or directed linear graph G = [N, A] consists of a collection N of elements $i, j, \ldots$ together with a subset A of the ordered pairs (i, j) of elements taken from N. It is assumed throughout that N is a finite set, since our interest lies mainly in the construction of computational procedures. The elements of N are variously called nodes, vertices, junction points, or points; members of A are referred to as arcs, links, branches, or edges. We shall use the node-arc terminology throughout. A network may be pictured by selecting a point corresponding to each node x of N and directing an arrow from i to j if the ordered pair (i, j) is in A. Such a network is said to be directed, since each arc carries a specific orientation or direction. Occasionally we shall also consider undirected networks, for which the set A consists of unordered pairs of nodes, or mixed networks, in which some arcs are directed, others are not. We can of course picture these in the same way, omitting arrowheads on arcs having no orientation. Until something is said to the contrary, however, each arc of the network will be assumed to have an orientation. We have not as yet ruled out the possibility of arcs (i, i) leading from a node i to itself, but for our purposes we may as well do so. Thus, all arcs will be supposed to be of the form (i, j) with i≠j. Also, while the existence of at most one arc (i, j) has been postulated, the notion of a network frequently allows multiple arcs joining i to j. Again, for most of the problems we shall consider, this added generality gains nothing, and so we shall continue to think of at most one arc leading from any node to another, unless an explicit statement is made to the contrary.

Let i_1, $i_{2,}$...,i_n (n≥ 2) be a sequence of distinct nodes of a network such that (i_m, i_{m+1}) is an arc, for each m=1,…,n-1, Then the sequence of nodes and arcs i_1, (i_1, i_2), i_2, . . ., (i_n, i_{n-1}), i_n is called a chain; it leads from i_1 to i_n. Sometimes, for emphasis, we call it a directed chain. If the definition of a chain is altered by stipulating that i_1=i_n, then the displayed sequence is a directed cycle.

Let i_1,$i_{2,}$ . . ., i_n be a sequence of distinct nodes having the property that either (i_m,i_{m+1}) or (i_{m+1},i_m) is an arc, for each m=1,..., n-1. Singling out, for each m, one of these two possibilities, we call the resulting sequence of nodes and arcs a path from i_1 to i_n. Thus a path differs from a chain by allowing the possibility of traversing an arc in a direction opposite to its orientation in going from i_1 to i_n. (For undirected networks, the two notions coincide.) Arcs (i_m,i_{m+1}) that belong to the path are forward arcs of the path; the others are reverse arcs.

Network Flows

We encounter many different types of networks in our everyday lives, including electrical, telephone, cable, highway, rail, manufacturing, and, computer networks. Networks consist of special points called nodes and links connecting pairs of nodes called arcs. Some examples of networks are listed in Figure 1.1. In all of these networks, we wish to send some commodity, which we generically call flow from one node to another, and do so as efficiently as possible, subject to certain constraints. Network flow theory is the study of designing computationally efficient algorithms to solve such problems.

Some examples of networks

Networks	Nodes	Arcs	Flows
Communication	Telephone exchanges, Computers, Satellites	Cables, fiber optics, microwave relay links	Voice message, video, data.
Hydraulic	Reservoirs, lakes, pumping Stations	Pipe lines	Hydraulic fluids, water, gas, oil.
Financial	Currencies, stokes	Transactions	Money
Transportation	Airports, rail yards, intersections	Highways, rail beds, airline routes.	Freight, vehicles, passengers.

Fig 1.1

Basic Definitions

All of the problems we consider are defined on a directed graph (N, E) where N is an *n*-set of nodes and E is an *m*-set of directed arcs. For notational convenience, we assume that the graph has no parallel arcs; this allows us to uniquely specify an arc by its endpoints. Our algorithms easily extend to allow for parallel arcs, and the complexity bounds we present remain valid. We consider only simple directed paths and cycles. In this section we give several basic definitions from graph

theory and present some basic notation. We also state some elementary properties of graphs. We begin by defining directed and undirected graphs.

Directed Graphs and Networks: A directed graph G = (N, A) consists of a set N of nodes and a set A of arcs whose elements are ordered pairs of distinct nodes. Figure 1. A below gives an example of a directed graph. For this graph, N= {1, 2, 3, 4, 5, 6, 7} and A = {(1, 2), (1, 3), (2, 3), (2, 4), (3, 6), (4, 5), (4, 7), (5, 2), (5, 3), (5, 7), (6, 7)}. A directed network is a directed graph whose nodes and/or arcs have associated numerical values (typically, costs, capacities, and/or supplies and demands). In this paper we often make no distinction between graphs and networks, so we use the terms "graph" and "network" synonymously.

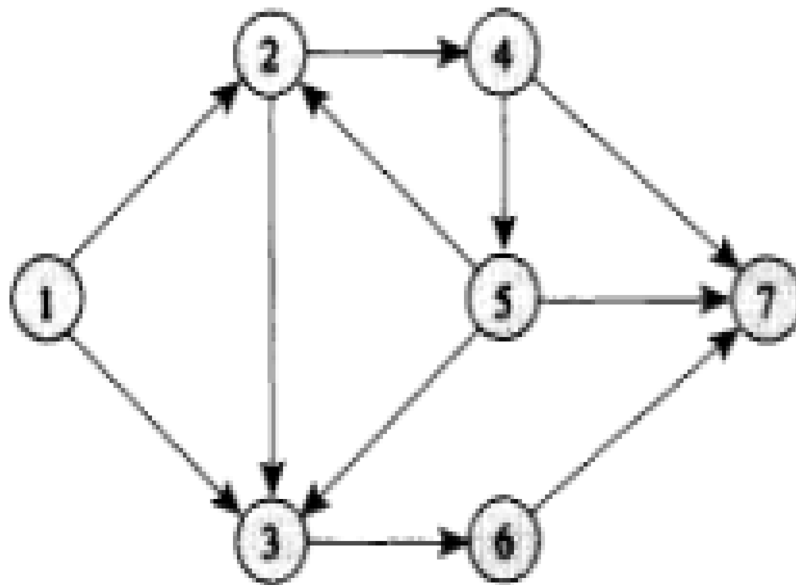


Figure 1.A

Undirected Graphs and Networks: We define an undirected graph in the same manner as we define a directed graph except that arcs are unordered pairs of distinct nodes. Figure 1.B gives an example of an undirected graph. In an undirected graph, we can refer to an arc joining the node pair i and j as either (i, j) or (j, i). An undirected arc (i,j) can be regarded as a two-way street with flow

permitted in both directions: either from node i to node j or from node j to node i. On the other hand, a directed arc (i,j) behaves like a one-way street and permits flow only from node i to node j.

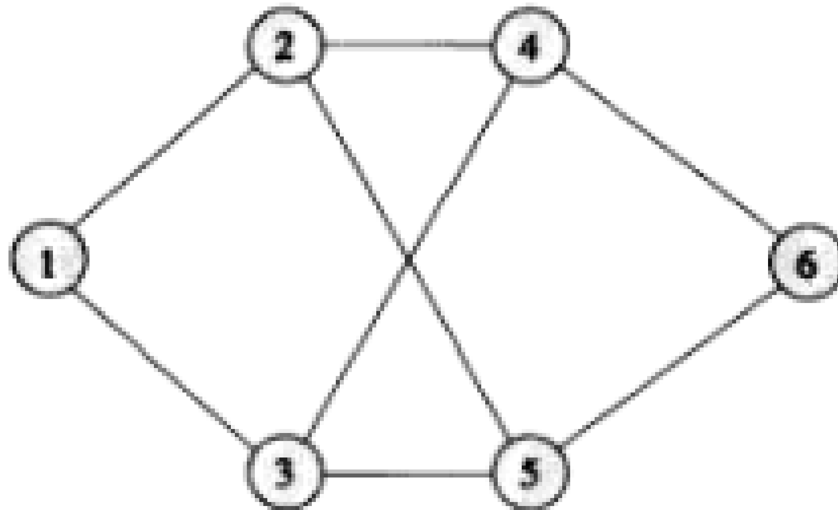


Figure 1.B.

Tails and Heads: A directed arc (i, j) has two end points i *and* j . We refer to node i as the tail of arc (i, j) and node j as its head. We say that the arc (i, j) emanates from node i and terminates at node j. An arc (i, j) is incident to nodes i and j. The arc (i, j) is an outgoing arc of node i and an incoming arc of node j. whenever an arc$(i, j) \in A$, we say that node j is adjacent to node i.

Sub-graph: A graph G' = (N', A') is a sub-graph of G = (N, A) if N' C N and A'CA. We say that G' = (N', A') is the sub-graph of G induced by N' if A' contains each arc of A with both endpoints in N'. A graph G' = (N', A') is a spanning sub-graph of G = (N, A) if N' = N and A' C A.

Walk: A walk in a directed graph G = (N, A) is a sub-graph of G consisting of a sequence of nodes and arcs $i_1 - a_1 - i_2 - a_2 - \dots - i_{r-1} - a_{r-1} - i_r$ satisfying the property that for all $1 \leq k \leq$ r - 1, either $a_k = (i_k, i_{k+1}) \in$ A or $a_k = (i_{k+1}, i_k) \in$

A. Alternatively, we shall sometimes refer to a walk as a set of (sequence of) arcs (or of nodes) without any explicit mention of the nodes (without explicit mention of arcs).

Directed Walk: A directed walk is an "oriented" version of a walk in the sense that for any two consecutive nodes $i_k and\ i_{k+1}$ on the walk if (i_k, i_{k+1}) ∈ A then (i_{k+1}, i_k) is not an element of A.

Path: A path is a walk without any repetition of nodes. We can partition the arcs of a path into two groups: forward arcs and backward arcs. An arc (i, j) in the path is a forward arc if the path visits node *i* prior to visiting node j, and is a backward arc otherwise.

Directed Path: A directed path is a directed walk without any repetition of nodes. In other words, a directed path has no backward arcs. We can store a path (or a directed path) easily within a computer by defining a predecessor index $pred(j)$ for every node j in the path. If i and j are two consecutive nodes on the path (along its orientation), $pred(j) = i.$ (Frequently, we shall use the convention of setting the predecessor index of the initial node of a path equal to zero to indicate the beginning of the path.) Notice that we cannot use predecessor indices to store a walk since a walk may visit a node more than once and a single predecessor index of a node cannot store the multiple predecessors of any node that a walk visits more than once.

Cycle: A cycle is a path $i_1 - i_2 - \cdots - i_r$ together with the arc$(i_r, i_1)\ or\ (i_1, i_r)$. we shall often refer to a cycle using the notation$i_1 - i_2 - \cdots - i_r - i_1$. Just as we did for paths, we can define forward and backward arcs in a cycle.

Directed Cycle: A directed cycle is a directed path $i_1 - i_2 - \cdots - i_r$ together with the arc (i_r, i_1) . The graph shown in Figure 1.C is a cycle, but not a directed cycle; the graph shown in Figure 2.D is a directed cycle.

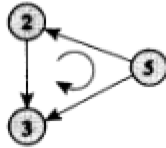


Fig 1.C

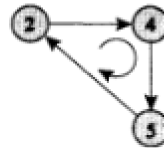


fig 1.D

Acyclic Graph: A graph is an acyclic if it contains no directed cycle. **Connectivity**: We will say that two nodes $i \; and \; j$ are connected if the graph contains at least one path from node i to node j. A graph is connected if every pair of its nodes is connected; otherwise, the graph is disconnected.

Tree: A tree is a connected graph that contains no cycle. Trees have the following property.

1) A tree on n nodes contains exactly n - 1 arcs.
2) A tree has at least two leaf nodes (i.e., nodes with degree 1).
3) Every two nodes of a tree are connected by a unique path.

Lengths: The shortest path and minimum mean cycle problems use a length function.

l: $E \longrightarrow R$ The length $l(i,j)$ is the distance from node i to node j. We denote the length of a cycle *(path)* Γ by

$$l(\Gamma) = \sum_{e \in \Gamma} l(e)$$

Costs: The minimum cost flow problem uses a cost function: E⟶ R The cost $C(i,j)$ is the unit shipping cost for arc (i,j) . We denote the cost of a cycle

$$C(\Gamma) = \sum_{e \in \Gamma} C(e)$$

Capacities: The maximum flow, minimum cost flow, and generalized maximum flow problem use a capacity function U: E⟶ $R \geq 0$. The capacity $u(i,j)$ limits the amount of flow we are permitted to send into arc (i,j).

Symmetry: For the maximum flow and minimum cost flow problems, we assume the input network is symmetric, i.e., if $(i,j) \in$ E then $(j,i) \in$ E also. This is without loss of generality, since we could always add the opposite arc and assign it zero capacity. Without loss of generality, we also assume the costs are anti-symmetric, i.e., $C(i,j) = -C(j,i)$ for every arc $(i,j) \in$ E. The reason for this assumption will become clear in the next paragraph.

Flows: A pseudo flow f: E⟶ R is a function that satisfies the capacity constraints: for all $(i,j) \in$ E: $f(i,j) \leq U(i,j)$ and the anti-symmetry constraints:

For all $(i,j) \in$ E: $f(i,j) = -f(j,i)$

To gain intuition, it is useful to think of only the nonnegative components of a pseudo flow. The negative flows are introduced for notational convenience. Note that we do not need to distinguish between upper and lower capacity limits. For example, a flow of 17 units sent along arc (i,j) is also viewed as a flow of -17 units sent along the reverse arc(j,i). The cost of sending one unit of flow along (i,j) is c_{ij} ; sending one unit of flow along the opposite arc(j,i) has cost $-c_{ij}$, and is equivalent to decreasing the flow on arc (i,j) by one unit. Now, to see how lower bounds are implicitly modeled, suppose arc(i,j) has zero capacity. This implies that variable $f(i,j)$ is nonnegative: the capacity constraint

for arc (i,j) is $f(j,i) \leq 0$, so then the anti-symmetry constraint implies $f(i,j) = -f(j,i) \geq 0$.

Residual Networks: With respect to a pseudo flow f in network G, the residual capacity functions u_f: E$\longrightarrow R$ is defined by $u_f(i,j) = u(i,j) - f(i,j)$. The residual network is $G_f = (V, E, u_f)$. Note that the residual network may include arcs with zero residual capacity, and still satisfies the symmetry assumption. For example, if $U(i,j) = 20$; $U(j,i) = 0$, and $(i,j) = -f(j,i) = 17$, then arc (i,j) has 20 -17 = 3 units of residual capacity, and arc (j,i) has 0-(-17) = 17 units of residual capacity.

We define $E_f = \{f(i,j) \in E: u_f(i,j) > 0\}$ to be the set of all arcs in G_f with positive residual capacity. A *residual arc* is an arc with positive capacity. A residual path (cycle) is a path (cycle) consisting entirely of residual arcs.

Network Flow Problems

In this section we formally define the shortest path, minimum mean cycle, maximum flow, minimum cut, and minimum cost flow problems. We also state the best known complexity bounds. We will use these as subroutines in our generalized flow algorithms

Minimum Cost Flow Problem

The minimum cost flow model is the most fundamental of all network flow problem. The problem is easy to state: We wish to determine a least cost shipment of a commodity through a network in order to satisfy demands at certain nodes from available supplies at other nodes. This model has a number of familiar applications: the distribution of a product from manufacturing plants to

warehouses, or from warehouses to retailers; the flow of raw material and intermediate goods through the various machining stations in a production line; the routing of automobiles through an urban street network; and the routing of calls through the telephone system. As we will see later the minimum cost flow model also has many less transparent applications. In this section we present a mathematical programming formulation of the minimum cost flow problem and then describe several of its specializations and variants as well as other basic models that we consider later.

Let G = (N, A) be a directed network defined by a set N of n nodes and a set A of m directed arcs. Each arc (i,j) ∈ A has an associated cost c_{ij} that denotes the cost per unit flow on that arc. We assume that the flow cost varies linearly with the amount of flow. We also associate with each arc (i, j) ∈ A a capacity U_{ij} that denotes the maximum amount that can flow on the arc and a lower bound l_{ij} that denotes the minimum amount that must flow on the arc. We associate with each node n ∈ N an integer number b(i) representing its supply/demand. If b(i) > 0, node i is a supply node; if b(i) < 0, node i is a demand node with a demand of –b(i) and if b(i) = 0, node i is a transshipment node. The decision variables in the minimum cost flow problem are arc flows and we represent the flow on an arc (i, j) ∈ A by x_{ij}. The minimum cost flow problem is an optimization model formulated as follows:

$$Minimize \sum_{(i,j)\in A} c_{ij}\, x_{ij}$$

$$subject\ to \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i), \qquad For\ all\ i \in N \ \ldots\ldots\ldots\ldots\ldots (*)$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \quad \text{for all (i,j)} \in A \text{ ....(**)}$$

$$WHERE \qquad \sum_{i=1}^{n} b(i) = 0.$$

We refer to the constraints in (*) as mass balance constraints. The first term in this constraint for a node represents the total outflow of the node (i.e., the flow emanating from the node) and the second term represents the total inflow of the node (i.e., the flow entering the node). The mass balance constraint states that the outflow minus inflow must equal the supply/demand of the node. If the node is a supply node, its outflow exceeds its inflow; if the node is a demand node, its inflow exceeds its outflow; and if the node is a transshipment node, its outflow equals its inflow. The flow must also satisfy the lower bound and capacity constraints (**), which we refer to as flow bound constraints. The flow bounds typically model physical capacities or restrictions imposed on the flows' operating ranges. In most applications, the lower bounds on arc flows are zero; therefore, if we do not state lower bounds for any problem, we assume that they have value zero.

In most parts of this paper we assume that the data are integral (i.e., all arc capacities, arc costs, and supplies/demands of nodes are integral). We refer to this assumption as the integrality assumption. The integrality assumption is not restrictive for most applications because we can always transform rational data to integer data by multiplying them by a suitably large number. Moreover, we necessarily need to convert irrational numbers to rational numbers to represent them on a computer. The following special versions of the minimum cost flow problem play a central role in the theory and applications of network flows.

Shortest path problem

The shortest path problem is perhaps the simplest of all network flow problems. For this problem we wish to find a path of minimum cost (or length) from a specified source node s to another specified sink node t, assuming that each arc (i,j) $\in$ A has an associated cost (or length) c_{ij}. Some of the simplest applications of the shortest path problem are to determine a path between two specified nodes of a network that has minimum length, or a path that takes least time to traverse, or a path that has the maximum reliability. This basic model has applications in many different problem domains, such as equipment replacement, project scheduling, cash flow management, message routing in communication systems, and traffic flow through congested cities. If we set b(s) = 1, $b(t)$= -1, and b(i) = 0 for all other nodes in the minimum cost flow problem, the solution to the problem will send 1 unit of flow from node s to node t along the shortest path. The shortest path problem also models situations in which we wish to send flow from a single-source node to a single-sink node in an uncapacitated network. That is, if we wish to send v units of flow from node s to node t and the capacity of each arc of the network is at least v, we would send the flow along a shortest path from node s to node t. If we want to determine shortest paths from the source node s to every other node in the network, then in the minimum cost flow problem we set b(s) = (n - 1) and $b(i)$= - 1 for all other nodes. [We can set each arc capacity U_{ij} to any number larger than (n - 1).] The minimum cost flow solution would then send unit flow from node s to every other node i along a shortest path.

Maximum flow problem

The maximum flow problem is in a sense a complementary model to the shortest path problem. The shortest path problem models situations in which flow incurs a cost but is not restricted by any capacities; in contrast, in the maximum flow problem flow incurs no costs but is restricted by flow bounds. The maximum flow problem seeks a feasible solution that sends the maximum amount of flow from a specified source node s to another specified sink node t. If we interpret U_{ij} as the maximum flow rate of arc (i, j), the maximum flow problem identifies the maximum steady-state flow that the network can send from node s to node t per unit time. Examples of the maximum flow problem include determining the maximum steady-state flow of petroleum products in a pipeline network, cars in a road network, messages in a telecommunication network, and electricity in an electrical network. We can formulate this problem as a minimum cost flow problem in the following manner. We set $b(i) = 0$ for all i ∈ N, $c_{ij} = 0$ for all (i, j) ∈ A, and introduce an additional arc (t, s) with cost c_{ts} = - 1 and flow bound $u_{ts} = \infty$. Then the minimum cost flow solution maximizes the flow on arc (t, s); but since any flow on arc (t, s) must travel from node s to node t through the arcs in A [since each $b(i) = 0$], the solution to the minimum cost flow problem will maximize the flow from node s to node t in the original network.

Minimum Cut Problem

The *s-t* minimum cut problem is intimately related to the maximum flow problem. The input is the same as for the maximum flow problem. The goal is to find a partition of the nodes that separates the source and sink, so that the total capacity of arcs going from the source side to the sink side is minimum. Formally, we define an *s-t cut* [*S; T*] to be a partition of the nodes

$V = S \cup T$ so that $s \in S$ and $t \in T$. The *capacity of a cut* is defined to be the sum of the capacities of "forward" arcs in the cut.

$$U[S,T] = \sum_{v \in S, w \in T} U(v,w)$$

(Cut capacity)

The goal is to find an s-t cut of minimum capacity. It is easy to see that the value of any flow is less than or equal to the capacity of any s-t cut. Any flow sent from s to t must pass through every s-t cut, since the cut disconnects s from t. Since flow is conserved, the value of the flow is limited by the capacity of the cut. A cornerstone result of network flows is the much celebrated max-flow min-cut theorem of Ford and Fulkerson. It captures the fundamental duality between the maximum flow and minimum cut problems.

Circulation problem

The circulation problem is a minimum cost flow problem with only transshipment nodes; that is, $b(i) = 0$ for all $i \in$ N. In this instance we wish to find a feasible flow that honors the lower and upper bounds l_{ij} and u_{ij} imposed on the arc flows x_{ij}. Since we never introduce any exogenous flow into the network or extract any flow from it, all the flow circulates around the network. We wish to find the circulation that has the minimum cost. The design of a routing schedule of a commercial airline provides one example of a circulation problem. In this setting, any airplane circulates among the airports of various cities; the lower bound l_{ij} imposed on an arc (i,j) is 1 if the airline needs to provide service between cities i and j, and so must dispatch an airplane on this arc (actually, the nodes will represent a combination of both a physical location and a time of day so that an arc connects two cities). We also study the following generalizations of the minimum cost flow problem.

Minimum spanning tree problem

A spanning tree is a tree (i.e., a connected acyclic graph) that spans (touches) all the nodes of an undirected network. The cost of a spanning tree is the sum of the costs (or lengths) of its arcs. In the minimum spanning tree problem, we wish to identify a spanning tree of minimum cost (or length). The applications of the minimum spanning tree problem are varied and include (1) constructing highways or railroads spanning several cities; (2) laying pipelines connecting offshore drilling sites, refineries, and consumer markets; (3) designing local access networks; and (4) making electric wire connections on a control panel.

A minimum spanning tree-problem is not a special type of minimum cost flow problem. (It is not even a special type of linear programming problem.) . That is the bad news. The good news is that you can solve it very easily by using Greedy algorithm without even using a computer.

Chapter 2

Maximum Flow Problem

The maximum flow problem and the shortest path problem are complementary. They are similar because they are both pervasive in practice and because they both arise as sub problems in algorithms for the minimum cost flow problem. The two problems differ, however, because they capture different aspects of the minimum cost flow problem: Shortest path problems model arc costs but not arc capacities; maximum flow problems model capacities but not costs. Taken together, the shortest path problem and the maximum flow problem combine all the basic ingredients of network flows. As such, they have become the nuclei of network optimizations. Our discussion o1f maximum flows, which we begin in this chapter, introduces several key ideas that reoccur often in the study of network flows. The maximum flow problem is very easy to state: In a capacitated network, we wish to send as much flow as possible between two special nodes, a source node s and a sink node t, without exceeding the capacity of any arc. In this chapter, we discuss a number of algorithms for solving the maximum flow problem. These algorithms are of two types:

1. Augmenting path algorithms that maintain mass balance constraints at every node of the network other than the source and sink nodes. These algorithms incrementally augment flow along paths from the source node to the sink node.

2. Preflow-push algorithms that flood the network so that some nodes have excesses (or buildup of flow). These algorithms incrementally relieve flow from

nodes with excesses by sending flow from the node forward toward the sink node or backward toward the source node.

We discuss the importance of the maximum flow problem; we begin by describing several applications. This discussion shows how maximum flow problems arise in settings as diverse as manufacturing, communication systems, distribution planning, matrix rounding, and scheduling. We begin our algorithmic discussion by considering a generic augmenting path algorithm for solving the maximum flow problem and describing an important special implementation of the generic approach, known as the labeling algorithm. The labeling algorithm is a pseudopolynomial-time algorithm. In this Chapter we develop improved versions of this generic approach with better theoretical behavior. The correctness of these algorithms rests on the renowned max-flow min-cut theorem of network flows (recall from the introduction part that a cut is a set of arcs whose deletion disconnects the network into two parts). This central theorem in the study of network flows (indeed, perhaps the most significant theorem in this problem domain) not only provides us with an instrument for analyzing algorithms, but also permits us to model a variety of applications in machine and vehicle scheduling, communication systems planning, and several other settings, as maximum flow problems. The max-flow min-cut theorem establishes an important correspondence between flows and cuts in networks. Indeed, as we will see, by solving a maximum flow problem, we also solve a complementary minimum cut problem: From among all cuts in the network that separate the source and sink nodes, find the cut with the minimum capacity. The relationship between maximum flows and minimum cuts is important for several reasons. First, it embodies a fundamental duality result that arises in many problem settings in discrete mathematics and that underlies linear programming as well as mathematical optimization in general. In fact, the max-

flow min-cut theorem, which shows the equivalence between the maximum flow and minimum cut problems, is a special case of the well-known strong duality theorem of linear programming. The fact that maximum flow problems and minimum cut problems are equivalent has practical implications as well. It means that the theory and algorithms that we develop for the maximum flow problem are also applicable to many practical problems that are naturally cast as minimum cut problems.

Notation and Assumptions

We consider a capacitated network G = (N, A) with a nonnegative capacity u_{ij} associated with each arc $(i,j) \in A$. Let $u = \max\{u_{ij} : (i,j) \in A\}$. To define the maximum flow problem, we distinguish two special nodes in the network G: a source node s and a sink node t. We wish to find the maximum flow from the source node s to the sink node t that satisfies the arc capacities and mass balance constraints at all nodes. We can state the problem formally as follows.

Maximize *v*

Subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v & for\ i = s \\ 0 & for\ all\ \ i \in N - \{s\ and\ t\} \ldots\ldots\ldots (a) \\ -v & for\ i = t \end{cases}$$

$$0 \le x_{ij} \le u_{ij} \quad for\ each\ (i,j)\in A \ldots\ldots\ldots\ldots\ldots\ldots\ldots (b)$$

We refer to a vector $x = \{x_{ij}\}$ satisfying (a) and (b) as a flow and the corresponding value of the scalar variable *v* as the value of the flow. We consider the maximum flow problem subject to the following assumptions.

Assumption 1: The network is directed.

We can always fulfill this assumption by transforming any undirected network into a directed network.

Assumption 2: All capacities are nonnegative integers.

Although it is possible to relax the integrality assumption on arc capacities for some algorithms, this assumption is necessary for others. Algorithms whose complexity bounds involve U assume integrality of the data. In reality, the integrality assumption is not a restrictive assumption because all modern computers store capacities as rational numbers and we can always transform rational numbers to integer numbers by multiplying them by a suitably large number.

Assumption 3: The network does not contain a directed path from node s to node t composed only of infinite capacity arcs.

Whenever every arc on a directed path P from s to t has infinite capacity, we can send an infinite amount of flow along this path, and therefore the maximum flow value is unbounded.

Assumption 4: Whenever an arc(i, j) belongs to A, arc(j, i) also belongs to A. This assumption is nonrestrictive because we allow arcs with zero capacity.

Assumption 5: The network does not contain parallel arcs.

Before considering the theory underlying the maximum flow problem and algorithms for solving it, and to provide some background and motivation for studying the problem, we first describe some applications.

APPLICATIONS

The maximum flow problem, and the minimum cut problem, arises in a wide variety of situations and in several forms. For example, sometimes the maximum flow problem occurs as a sub problem in the solution of more difficult network problems, such as the minimum cost flow problem or the generalized flow problem. As we will see later, the maximum flow problem also arises in a number of combinatorial optimization applications that might not appear to be maximum flow problems at all. The problem also arises directly in problems as far reaching as machine scheduling, the assignment of computer modules to computer processors. In this section we describe a few such applications.

Application 1 Feasible Flow Problem

The feasible flow problem requires that we identify a flow x in a network

G = (N, A) satisfying the following constraints:

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

$$0 \le x_{ij} \le u_{ij} \quad for\ each\ (i,j)\in A \ldots\ldots\ldots\ldots(b)$$

As before, we assume that $\sum_{i\in N} b(i) = 0$. The following distribution scenario illustrates how the feasible flow problem arises in practice. Suppose that merchandise is available at some seaports and is desired by other ports. We know the stock of merchandise available at the ports, the amount required at the other ports, and the maximum quantity of merchandise that can be shipped on a particular sea route. We wish to know whether we can satisfy all of the demands by using the available supplies. We can solve the feasible flow problem by solving

a maximum flow problem defined on an augmented network as follows. We introduce two new nodes, a source node s and a sink node t. For each node i with $b(i) > 0$, we add an arc(s, i) with capacity $b(i)$, and for each node i with $b(i) < 0$, we add an arc (i, t) with capacity $-b(i)$ we refer to the new network as the transformed network. Then we solve a maximum flow problem from node s to node t in the transformed network. If the maximum flow saturates all the source and sink arcs, the problem has a feasible solution; otherwise, it is infeasible.

Application 2 Scheduling on Uniform Parallel Machines

In this application we consider the problem of scheduling of a set J of jobs on M uniform parallel machines. Each job j ∈ J has a processing requirement p_j (denoting the number of machine days required to complete the job), a release date r_j (representing the beginning of the day when job j becomes available for processing), and a due date $d_j \geq r_j + p_j$ (representing the beginning of the day by which the job must be completed). We assume that a machine can work on only one job at a time and that each job can be processed by at most one machine at a time. However, we allow preemptions (i.e., we can interrupt a job and process it on different machines on different days). The scheduling problem is to determine a feasible schedule that completes all jobs before their due dates or to show that no such schedule exists. Scheduling problems like this arise in batch processing systems involving batches with a large number of units. The feasible scheduling problem, described in the preceding paragraph, is a fundamental problem in this situation and can be used as a subroutine for more general scheduling problems, such as the maximum lateness problem, the (weighted) minimum completion time

problem, and the (weighted) maximum utilization problem. Let us formulate the feasible scheduling problem as a maximum flow problem.

We illustrate the formulation using the scheduling problem described in Figure below with M = 3 machines. First, we rank all the release and due dates, $r_j\, and d_j$ for all j, in ascending order and determine $p \leq 2|J| - 1$ mutually disjoint intervals of dates between consecutive milestones. Let $T_{k,l}$ denote the interval that starts at the beginning of date k and ends at the beginning of date $l + 1$. For our example, this order of release and due dates is 1, 3, 4, 5, 7, and 9. We have five intervals, represented by $T_{1,2}, T_{3,3}, T_{4,4}, T_{5,6}, and T_{7,8}$. Notice that within each interval $T_{k,l}$, the set of available jobs (i.e., those released but not yet due) does not change: we can process all jobs j with $r_j \leq k\ and\ d_j \geq l + 1$, in the interval.

Job(j)	1	2	3	4
Processing time p_j	1.5	1.25	2.1	3.6
Release time r_j	3	1	3	5
Due date d_j	5	4	7	9

Figure c. Scheduling problem.

We formulate the scheduling problem as a maximum flow problem on a bipartite network G as follows. We introduce a source node s, a sink node t, a node corresponding to each job j, and a node corresponding to each interval $T_{k,l}$ as

shown in Figure d. We connect the source node to every job node j with an arc with capacity p_j, indicating that we need to assign p_j days of machine time to job j. We connect each interval node $T_{k,l}$ to the sink node t by an arc with capacity

$(l - k + 1)M$, representing the total number of machine days available on the days from $k\ to\ l$.

Finally, we connect a job node j to every interval node $T_{k,l}$ if $r_j \leq k\ and\ d_j \geq l + 1$ by an arc with capacity $(l - k + 1)$ which represents the maximum number of machines days we can allot to job j on the days from $k\ to\ l$. We next solve a maximum flow problem on this network: The scheduling problem has a feasible schedule if and only if the maximum flow value equals $\sum_{j \in J} p_j$[alternatively, the flow on every arc $(s, j)\ is\ p_j$. The validity of this formulation is easy to establish by showing a one-to-one correspondence between feasible schedules and flows of value $\sum_{j \in J} p_j$ from the source to the sink.

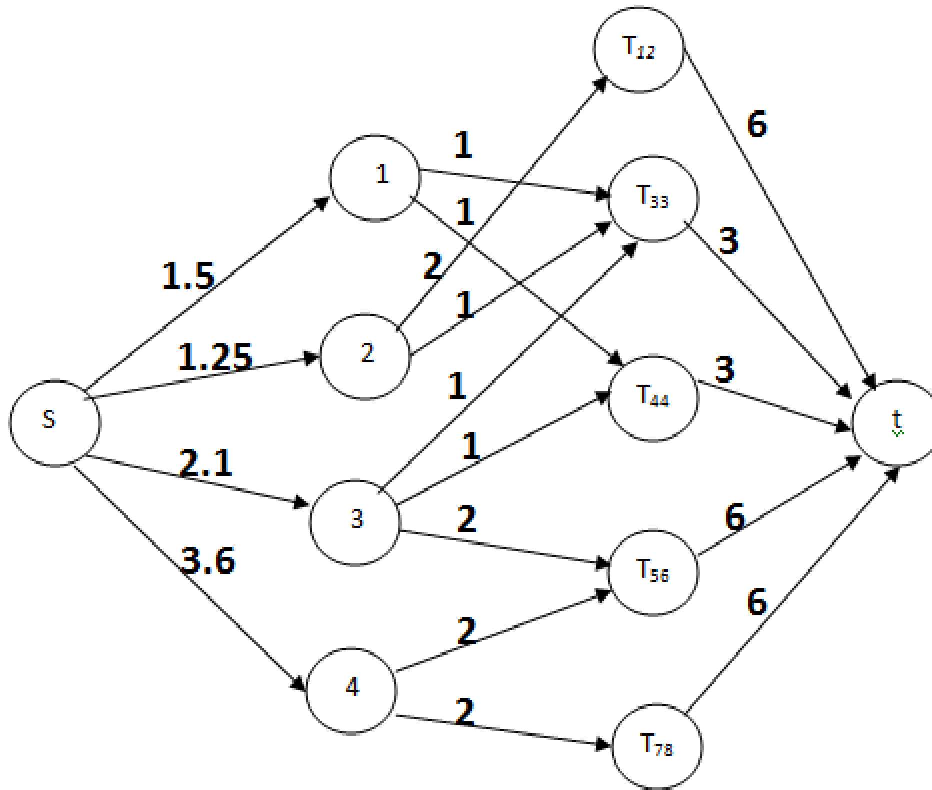


Figure d. Network for scheduling uniform parallel machines

FLOWS AND CUTS

In this section we discuss some elementary properties of flows and cuts. We use these properties to prove the max-flow min-cut theorem to establish the correctness of the generic augmenting path algorithm. We first review some of our previous notation and introduce a few new ideas.

Residual network: The concept of residual network plays a central role in the development of all the maximum flow algorithms we consider. Earlier we defined residual networks and discussed its properties. Given a flow x, the residual capacity r_{ij} of any arc $(i,j) \in A$ is the maximum additional flow that can be sent from node i to node j using the arcs $(i,j) and (j,i)$. [Recall our assumption from

that whenever the network contains arc (i,j), it also contains arc(j,i).] The residual capacity r_{ij} has two components: (1) $u_{ij} - x_{ij}$, the unused capacity of arc (i,j), and (2) the current flow x_{ij} on arc (j, i), which we can cancel to increase the flow from node i to node j. Consequently, $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. We refer to the network G(x) consisting of the arcs with positive residual capacities as the residual network (with respect to the flow x). Figure e gives an example of a residual network.

S-t cut: We now review notation about cuts. Recall that a cut is a partition of the node set N into two subsets $S\ and \bar{S}$ $=N - S$; we represent this cut using the notation $[S,\bar{S}]$. Alternatively, we can define a cut as the set of arcs whose endpoints belong to the different subsets S $and \bar{S}$. We refer to a cut as an s-t cut if $\in S\ and\ t \in \bar{S}$. We also refer to an arc (i,j) with $i \in S\ and\ j \in \bar{S}$ as a forward arc of the cut, and an arc $(i,j)\ with\ i \in \bar{S}$ and $j \in S$ as a backward arc of the cut $[S,\bar{S}]$. Let $(S,\bar{S})$ denote the set of forward arcs in the cut, and let $(\bar{S},S)$ denote the set of backward arcs. For example, in Figure f, the dashed arcs constitute an s-t cut. For this cut, $(S,\bar{S})$ = {(1, 2), (3, 4), (5, 6)}, and $(\bar{S},S)$ = {(2, 3), (4, 5)}.

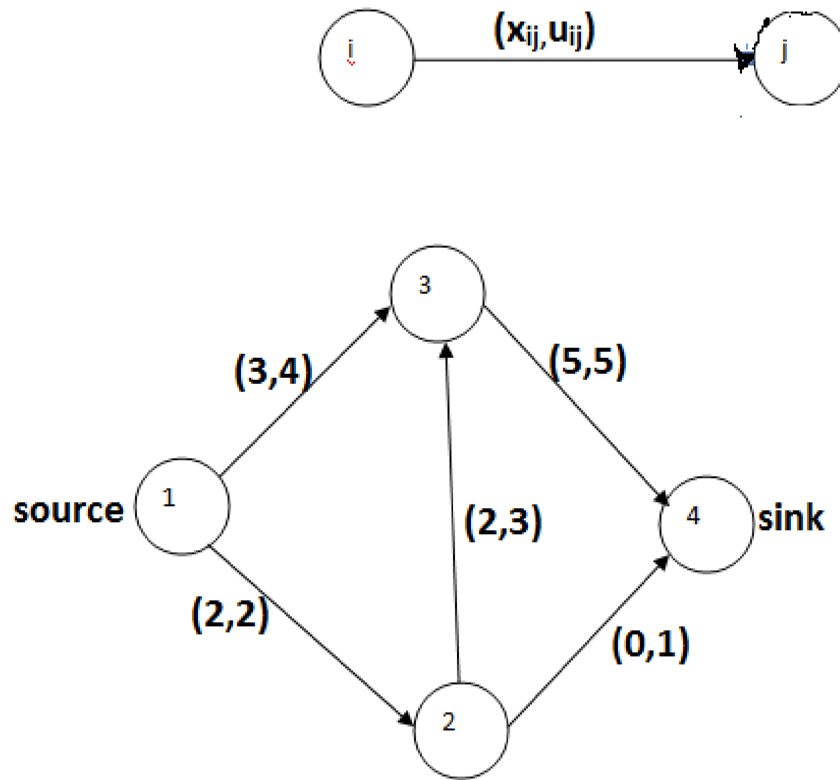


Figure e.1 an **orginal network with flow X and upper bound U.**

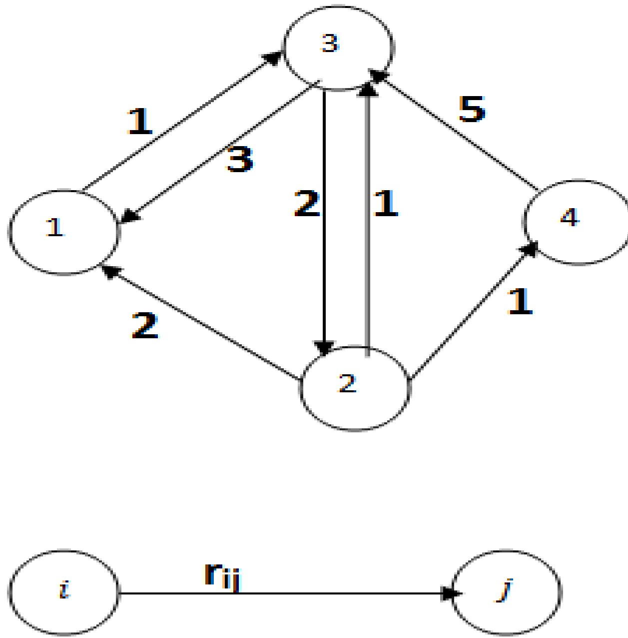


Figure e.2 the **residual network of the graph above**.

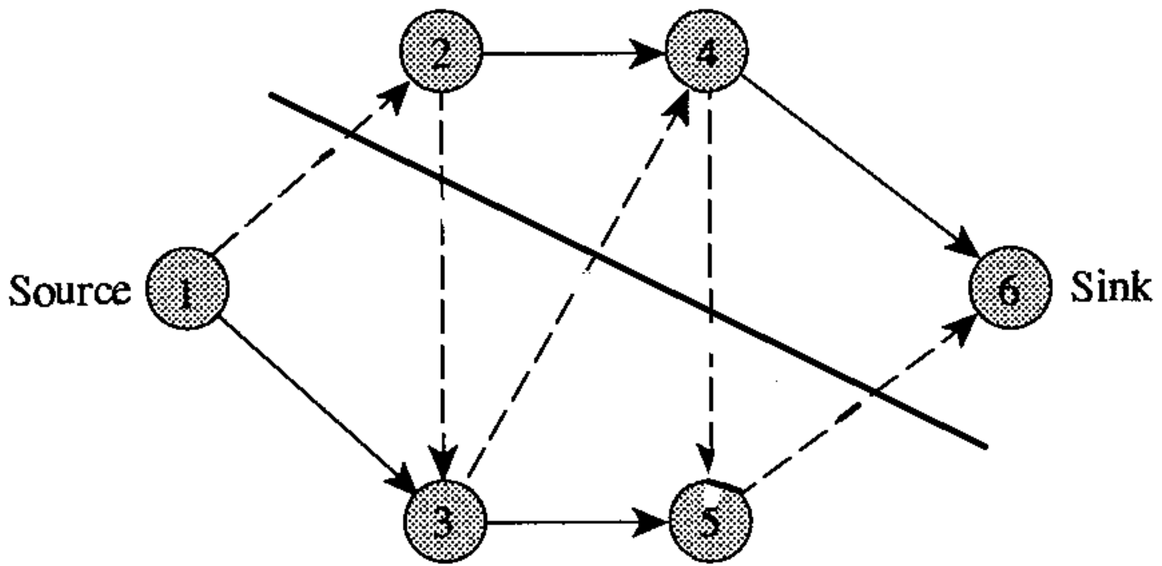


Figure f. example of s-t cut.

GENERIC AUGMENTING PATH ALGORITHM

We describe one of the simplest and most intuitive algorithms for solving the maximum flow problem. This algorithm is known as the augmenting path algorithm. We refer to a directed path from the source to the sink in the residual network as an augmenting path. We define the residual capacity of an augmenting path as the minimum residual capacity of any arc in the path. For example, the residual network in Figure e.2 above, contains exactly one augmenting path 1-3-2-4, and the residual capacity of this path is min {l, 2, 1} = 1. Observe that, by definition, the capacity of an augmenting path is always positive. Consequently, whenever the network contains an augmenting path, we can send additional flow from the source to the sink. The generic augmenting path algorithm is essentially based on this simple observation. The algorithm proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path. The generic augmenting path algorithm works as follows.

Algorithm augmenting path;

Begin

X: = 0;

While G(x) contains a directed path from node s to node t do

Begin

Identify an augmenting path P from node s to node t;

$\boldsymbol{\delta} = \min \{r_{ij} : (i, j) \in p\}$;

Augment **δ** units of flow along P and update G(x);

End;

End;

LABELING ALGORITHM AND THE MAX-FLOW MIN-CUT THEOREM

Here we discuss the augmenting path algorithm in more detail. In our discussion of this algorithm in the previous part, we did not discuss some important details, such as (1) how to identify an augmenting path or show that the network contains no such path, and (2) whether the algorithm terminates in finite number of iterations, and when it terminates, whether it has obtained a maximum flow. In this section we consider these issues for a specific implementation of the generic augmenting path algorithm known as the labeling algorithm. The labeling algorithm is not a polynomial-time algorithm. The labeling algorithm uses a search technique to identify a directed path in G(x) from the source to the sink. The algorithm fans out from the source node to find all nodes that are reachable from the source along a directed path in the residual network. At any step the algorithm has partitioned the nodes in the network into two groups: labeled and unlabeled. Labeled nodes are those nodes that the algorithm has reached in the fanning out process and so the algorithm has determined a directed path from the source to these nodes in the residual network; the unlabeled nodes are those nodes that the algorithm has not reached as yet by the fanning-out process. The algorithm iteratively selects a labeled node and scans its arc adjacency list (in the residual network) to reach and label additional nodes. Eventually, the sink becomes labeled and the algorithm sends the maximum possible flow on the path from node s to node t. It then erases the labels and repeats this process. The algorithm terminates when it has scanned all the labeled nodes and the sink remains unlabeled, implying that the source node is not connected to the sink node in the residual network.

Theorem 2.1 (Augmenting Path Theorem) A Flow is maximal if and only if it admits no augmenting path from s to t.

PROOF Clearly, if an augmenting path exists the flow is not maximal. Suppose x is a flow that does not admit an augmenting path. Let S be the set of all nodes j

(including s) for which there is an augmenting path from s to j, and let T be the complementary set. From the definition of augmenting path and from the definition of S and T: it follows that for all $i \in$ S and $j \in$ T,

$x_{ij} = c_{ij}$ and $x_{ij} = 0$. It follows from that

$$v = \sum_{i \in S} \sum_{j \in T} c_{ij}$$

The capacity of the cut set (S, *T)* is maximal.

Theorem 2.2 *(Integral Flow Theorem)* if all arc capacities are integers there is a maximal flow which is integral.

PROOF Suppose all capacities are integers and let $x_{ij}^0 = 0$, for all i and j. If the flow $x^0 = (x_{ij}^0)$ is not maximal it admits an augmenting path and hence there is an integral flow x^1 whose value exceeds that of x^0. If x^1 is not maximal it admits an augmenting path, and so on. As each flow obtained in this way exceeds the value of its predecessor by at least one, we arrive eventually at an integral flow that admits no augmenting path and hence is maximal.

Theorem *2.3 (Max-Flow Min-Cut Theorem)* the maximum value of an (s, *t)-flow* is equal to the minimum capacity of an (s, t)-cut-set.

PROOF The proofs of the previous two theorems are sufficient to establish the max-flow min-cut result for networks in which all capacities are integers and hence for those in which all capacities are commensurate (i.e., there exists some $\mathcal{E} > 0$ such that every c_{ij} is an integral multiple of c).

To complete the proof of the max-flow min-cut result, we must show that every network actually admits a maximal flow. (Note that the existence of a minimum capacity cut-set is not open to question,. There are only a finite number of (s, t)-cut-sets, and at least one of them must be minimal.)

We shall present now an algorithm for computing maximal flows.

Maximal Flow Algorithm

Step 0 (Start) let $x = x_{ij}$ be any integral feasible flow, possibly the zero flow. Give node s the permanent label $(-, \infty)$.

Step 1 *(Labeling and Scanning)*

(1.1). If all labeled nodes have been scanned, go to Step 3.

(1.2). find a labeled but unscanned node i and scan it as follows: For each arc *(i, j),* if $x_{ij} < c_{ij}$ and j is unlabeled, give j the label (i^{-}, δ_j), where

$$\delta_j = \min\{c_{ij} - x_{ij}, \delta_i\}$$

For each arc (j, *i),* if $x_{ji} > 0$, and j is unlabeled, give j the label (i^{-}, δ_j), where

$$\delta_j = \min\{x_{ji}, \delta_i\}$$

(1.3). If node t has been labeled, go to Step 2; otherwise go to Step 1.1.

Step 2 *(Augmentation)* Starting at node t, use the index labels to construct an augmenting path. (The label on node I indicates the second-to-last node in the path, the label on that node indicates the third-to-last node. and so on.) Augment the flow by increasing and decreasing the arc flows by δ_i as indicated by the superscripts on the index labels. Erase all labels, except the label on node s. *GO* to Step 1.

Step 3 *(Construction of Minimal cut)* the existing flow is maximal. A cut-set of minimum capacity is obtained by placing all labeled nodes in S and all unlabeled nodes in T. The computation is completed.

Chapter 3

Minimum Cost Flow problems

Networks are especially convenient for modeling because of their simple nonmathematical structure that can be easily depicted with a graph. This simplicity also reaps benefits with regard to algorithmic efficiency. Although the simplex method of linear programming is one of the primary solution techniques, we can see that its implementation for network problems allows many procedural simplifications. Small instances can be solved by hand and computer program. This section provides a solution algorithm for the pure network flow programming problem on a directed graph and the procedure for solving a generalized mode.

Problem Statement

A pure network flow minimum cost flow problem is defined by a given set of arcs and a given set of nodes, where each arc has a known capacity and unit cost and each node has a fixed external flow. The optimization problem is to determine the minimum cost plan for sending flow through the network to satisfy supply and demand requirements. The arc flows must be nonnegative and be no greater than the arc capacities, and they must satisfy conservation of flow at the nodes. For this section, it is assumed that all arc lower bounds on flow are zero.

Formulation

We now formulate the problem as a linear program for a directed, connected graph with m nodes and n arcs. It is assumed that there is at least one supply node and one demand node. Regarding notation, the arc connecting nodes i and j was denoted by (i, j) and the decision variable associated with that arc was x_{ij} .

x_{ij} = flow through arc (i, j) from node i to node j

And the given data are

c_{ij} = unit cost of flow through arc (i, j)

$b(i) =$ Net supply (arc flow out – arc flow in) at node i

u_{ij} = Capacity of arc (i, j)

The value of $b(i)$ is determined by the nature of node i. In particular,

$b(i) > 0$ if i is a supply node;

$b(i) < 0$ if i is a demand node;

$b(i) = 0$ if i is a transshipment node;

The mathematical model is

$$Minimize \sum_{(i,j)\in A} c_{ij}\, x_{ij} \; \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots (a)$$

$$subject\ to \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i), \qquad For\ all\ i \in N \; \dots\dots\dots\dots\dots (b)$$

$$0 \le x_{ij} \le u_{ij} \quad \text{for all } (i,j) \in A \; \dots\dots\dots\dots\dots\dots\dots\dots (c)$$

$$WHERE \qquad \sum_{i=1}^{n} b(i) = 0.$$

The objective function (a) sums the arc costs over all arcs in the network. Equation (b) describes the flow balance or conservation of flow constraints. The first summation represents the flow out of node i, and the second is the flow in. The difference between the two represents the net flow generated at node i. The right side values $b(i)$ are positive if node i supplies flow, negative if it consumes flow and zero otherwise. In some applications, the lower bound in (c) is not zero but some arbitrary value l_{ij}. But it is always possible to transform such a lower bound to zero by introducing the variable

$$\hat{x}_{ij} = x_{ij} - l_{ij} \ and\ substituting\ \hat{x}_{ij} + l_{ij}\ for\ x_{ij}$$

$$\hat{u}_{ij} = u_{ij} - l_{ij}$$

Feasibility property: A necessary condition for a pure minimum cost flow problem to have a feasible solution is

$$\sum_{i=1}^{n} b(i) = 0$$

In other words, the total flow being supplied at the nodes in the network must equal the total demand being absorbed by the nodes in the network.

Residual Network

Our algorithms rely on the concept of residual networks. The residual network G(x) corresponding to a flow x is defined as follows. We replace each arc $(i, j) \in A$ by two arcs (i, j) and (j, i). The arc (i, j) has cost c_{ij} and residual capacity $r_{ij} = u_{ij} - x_{ij}$ and the arc (j, i) has cost $c_{ji} = -c_{ij}$ and residual capacity $r_{ij} = x_{ij}$. The residual network consists only of arcs with positive residual capacity.

Theorem: (Negative Cycle Optimality Conditions).

A feasible solution x* is an optimal solution of the minimum cost flow problem if and only if it satisfies the negative cycle optimality conditions: namely, the residual network G(x*) contains no negative cost (directed) cycle.

Proof: Suppose that x is a feasible flow and that G(x) contains a negative cycle. Then x cannot be an optimal flow, since by augmenting positive flow along the cycle we can improve the objective function value. Therefore, if x* is an optimal flow, then G(x*) cannot contain a negative cycle. Now suppose that x* is a feasible flow and that G(x*) contains no negative cycle. Let x° be an optimal flow and

x*≠ x°. The augmenting cycle property shows that we can decompose the difference vector x° - x* into at most m augmenting cycles with respect to the flow x* and the sum of the costs of flows on these cycles equals $cx^0 - cx^*$. Since the lengths of all the cycles in G(x*) are nonnegative, $cx^0 - cx^* \geq 0$, or $cx^0 \geq cx^*$. Moreover, since x° is an optimal flow, $cx^0 \leq cx^*$. Thus $cx^0 = cx^*$, and x* is also an optimal flow. This argument shows that if G(x*) contains no negative cycle, then x* must be optimal, and this conclusion completes the proof of the theorem.

Node potential

We can analyze the optimality of a circulation using a node potential which we call price functions. Think of the flow units as widgets that are given away at the source and they are paid for at the source. There is a market for widgets at intermediate vertices. We can define then a price function π for the vertices of the network. At the source, π(s) = 0.Consider an edge (i, j) which has residual capacity. The price $\pi(j)$ is feasible if $\pi(j) \leq \pi(j) + C(i, j)$

Definition: The reduced cost of an edge (i, j) is $c_{ij}^{\pi} = c_{ij} + \pi(i) - \pi(j)$. We can think of the reduced cost as the cost of buying a widget at i, shipping it to j and selling it there. Note that if c_{ij}^{π} is positive, we would therefore not ship the item from i to j. using this definition, we can say that a price function is feasible for a residual graph if no residual edge has a negative reduced cost.

Reduced Cost Optimality Conditions

Suppose that we associate a real number $\pi(i)$, unrestricted in sign, with each node $i \in N$. We refer to $\pi(i)$as the potential of node i. $\pi(i)$ is the linear programming dual variable corresponding to the mass balance constraint of node i. For a given set of node potentials $\pi(i)$, we define the reduced cost of an arc (i, j) as $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$. These reduced costs are applicable to the residual network as well as the original network. We define the reduced costs in the

residual network just as we did the costs, but now using c_{ij}^{π} $in\ place\ of\ c_{ij}$. The following properties will prove to be useful in our subsequent developments in this chapter.

Property

(a) For any directed path P from node k to node l,

$$\sum_{(i,j)\in p} c_{ij}^{\pi} = \sum_{(i,j)\in p} c_{ij} - \pi(k) + \pi(l).$$

(b) For any directed cycle W,

$$\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij}\,.$$

Notice that this property implies that the node potentials do not change the shortest path between any pair of nodes k and l, since the potentials increase the length of every path by a constant amount $\pi(l) - \pi(k)$. This property also implies that if W is a negative cycle with respect to c_j as arc costs, it is also a negative cycle with respect to c_{ij}^{π} as arc costs. We can now provide an alternative form of the negative cycle optimality conditions, stated in terms of the reduced costs of the arcs.

Reduced Cost Optimality Conditions: A feasible solution x* is an optimal solution of the minimum cost flow problem if and only if some set of node potentials π satisfy the following reduced cost optimality conditions:

$$if\ c_{ij}^{\pi} > 0\ for\ every\ arc\ (i,j) in\ G(x^*)$$

Theorem: (Complementary Slackness Optimality Conditions). A feasible solution x* is an optimal solution of the minimum cost flow problem if and only if for some set of node potentials π, the reduced costs and flow values satisfy the following complementary slackness optimality conditions for every arc $(i,j) \in A$:

$$if\ c_{ij}^{\pi} > 0\ then\ x_{ij}^{*} = 0$$

$$if\ 0 < x_{ij} < u_{ij}\ then\ c_{ij}^{\pi} = 0$$

$$if\ c_{ij}^{\pi} < 0\ then x_{ij}^{*} = u_{ij}$$

Proof: We show that the reduced cost optimality conditions are equivalent to our complementary slackness optimality condition. To establish this result, we first prove that if the node potentials π and the flow vector x satisfy the reduced cost optimality conditions, and then they must satisfy the theorem. Consider three possibilities for any arc $(i, j) \in A$.

Case 1:If $c_{ij}^{\pi} > 0$, the residual network cannot contain the arc (j, i) because $c_{ji}^{\pi} = -c_{ij}^{\pi} < 0$ for that arc, contradicting reduced cost optimality condition. Therefore, x* = 0.

Case 2: If $0 < x_{ij}^{*} < u_{ij}$, the residual network contains both the arcs (i, j) and (j, i). The reduced cost optimality conditions imply that $c_{ij}^{\pi} \geq 0$ and $c_{ji}^{\pi} \geq 0$. But since $-c_{ij}^{\pi} = c_{ji}^{\pi}$, these inequalities imply that $c_{ij}^{\pi} = c_{ji}^{\pi} = 0$.

Case 3: If $c_{ij}^{\pi} < 0$, the residual network cannot contain the arc (i, j) because $c_{ij}^{\pi} < 0$ for that arc, contradicting the reduced cost optimality condition. Therefore, $x_{ij}^{*} = u_{ij}$. We have thus shown that if the node potentials π and the flow vector x satisfy the reduced cost optimality conditions, they also satisfy the complementary slackness optimality conditions.

CYCLE-CANCELING ALGORITHM AND THE INTEGRALITY PROPERTY

The negative cycle optimality conditions suggest one simple algorithmic approach for solving the minimum cost flow problem, which we call the cycle-canceling algorithm. This algorithm maintains a feasible solution and at every iteration attempts to improve its objective function value. The algorithm first establishes a feasible flow x in the network by solving a maximum flow problem. Then it iteratively finds negative cost-directed cycles in the residual network and augments flows on these cycles. The algorithm terminates when the residual network

contains no negative cost-directed cycle. Which implies that the algorithm terminates, and has found a minimum cost flow. Below we specify this generic version of the cycle-canceling algorithm.

Algorithm: cycle-canceling;

Begin

Establish a feasible flow x in the network;

While G(x) contains a negative cycle do

Begin

Identify a negative cycle W;

$\delta := \min \{r_{ij}, (i, j) \in W\}$;

Augment δ units of flow in the cycle W and update G(x);

End;

End;

Integrality Property: If all arc capacities and supplies/demands of nodes are integer, the minimum cost flow problem always has an integer minimum cost flow.

OUT-OF-KILTER ALGORITHM

The out-of-kilter algorithm, like the simplex algorithm, is one of the« classical algorithms of operational research. It was proposed independently by Minty in 1960 and by Fulkerson in 1961. From time to time, alternative network algorithms are reported with faster running times. However, the speed of the out-of-kilter algorithm depends critically on its mode of implementation. With a well-structured, fast implementation, the out-of-kilter algorithm is probably still strongly competitive with the other linear network algorithms. The out-of-kilter algorithm offers the following advantages over many other network algorithms:

(1) It uses the basic structure of nodes and arcs, and so can be readily implemented using a computer program involving linked-list structures;

(2) It does not require an initial feasible solution (i.e. a solution satisfying the capacity constraints) to be found before starting the algorithm;

(3) If the original network is modified, the algorithm can be re-started where it finished on the original network, thereby reducing considerably the extra computation required.

The out-of-kilter algorithm requires that the total flow into any node should always be equal to the total flow out of that node. If the network contains source and sink nodes that, respectively create and destroy flow, the network must be modeled by joining these nodes by dummy arcs (entering and leaving, if necessary, a dummy node) with whatever capacity is required and having zero cost. These arcs maintain conservation of flow at the source and sink nodes without affecting the optimal solution. An initial solution (not necessarily feasible) that conserves flow can be simply one with all flows in all arcs set to zero. If the complementary slackness condition is satisfied for any arc$(i,j) \in A$ we say the arc is 'in-kilter' (meaning, effectively, 'in balance'). If an arc is initially out-of-kilter, the algorithm will attempt to bring it into kilter by changing the flow through that arc and through a cycle (closed path) of arcs including that arc, in such a way that no arc that is already in-kilter is put out-of-kilter. The flow change must be made through a cycle of arcs in order to maintain conservation of flow. If this is possible, then *breakthrough* is said to have been achieved, and the flow is changed in the way indicated, thus bringing at least one more arc into kilter. If breakthrough is not possible with the existing node prices, then some of these prices are modified so as to bring, or to approach more closely, breakthrough. These steps are repeated until all arcs are in-kilter. If this is found not to be possible, then there is no solution to the original network problem; it is an infeasible problem. If all arcs are brought in-kilter then the flows are optimal, and it can be proved that the out-of-kilter algorithm will find this optimal solution in a finite number of steps.

demand/supply
(upper bound, cost)

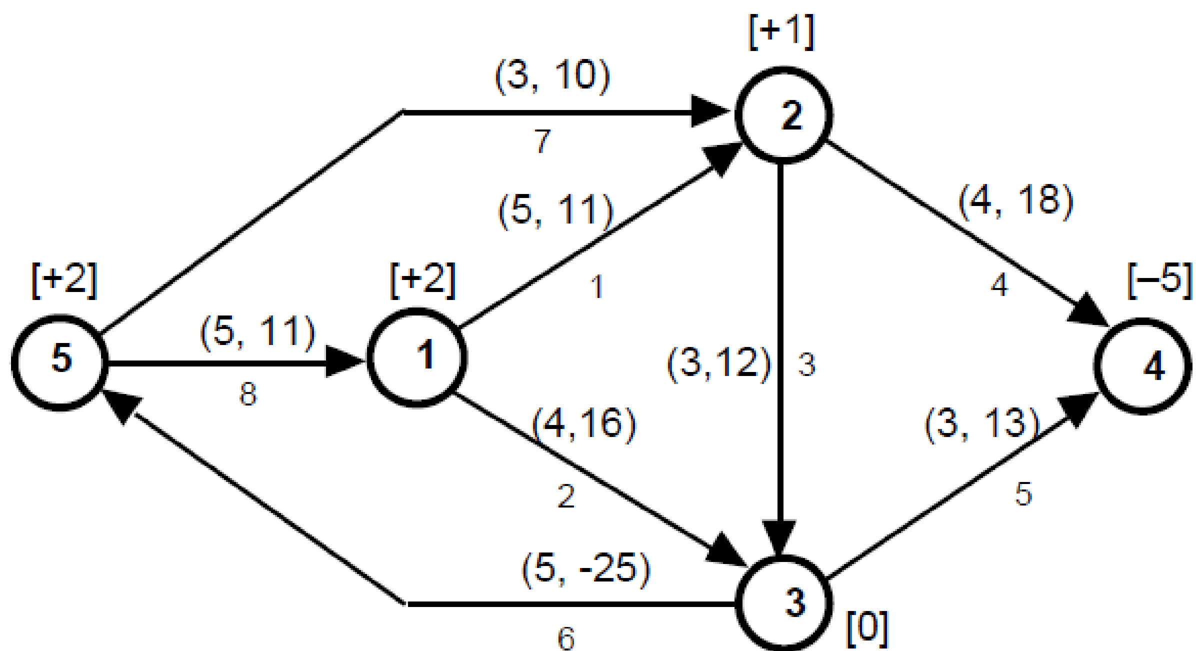


Figure a. An example of a minimum cost flow network

Example

We use the network in Fig a above as an example. Upper bounds and costs are given for each arc as shown in the figure. It is convenient to identify vectors that represent the parameters of the problem. They are respectively, **x**, for arc flows, **u**, for arc flow capacities or upper bounds, **c**, for arc unit costs, and **b**, for node external flow. For the vectors for the example are below

Arc flows: **x** = [$x_{12}, x_{13}, x_{23}, x_{24}, x_{34}, x_{35}, x_{52}, x_{51}$]

Upper bounds: **u** = [5, 4, 3, 4, 3, 5, 3, 5]

Unit costs: **c** = [11, 16, 12, 18, 13, -25, 10, 11]

External flows: **b** = [2, 1, 0, –5, 2]

Basic Solutions

Because an optimal solution must be among the finite set of bases, the simplex algorithm only examines basic solutions as it iterates. When applied to the minimum cost flow problem, the algorithm maintains the needed information more efficiently and hence is able to access it more quickly than when applied to a general linear programming problem. This section describes basic solutions for the network flow programming problem and provides procedures for computing the primal and dual solutions associated with a given basis.

Basis Tree

We know that an optimal solution to the linear program, and to the network flow problem by extension, is a basic solution. The basis is defined by a selection of independent variables equal in number to the number of linearly independent constraints. Since the network model contains m – 1 independent conservation of flow constraints and the variables are the arc flows, a basis is determined by selecting m – 1 independent arc. These are identified by the arc indices of the basic variables. Call the corresponding set n_b. For the pure minimum cost flow problem, we have the interesting characteristic that every basis defines a spanning tree sub-network. To illustrate, let n_b = [2, 4, 7, 8] for the network of Fig. a. Drawing only the selected arcs forms the sub-network shown in Fig. b. Node 5 is defined as the root node of the tree.

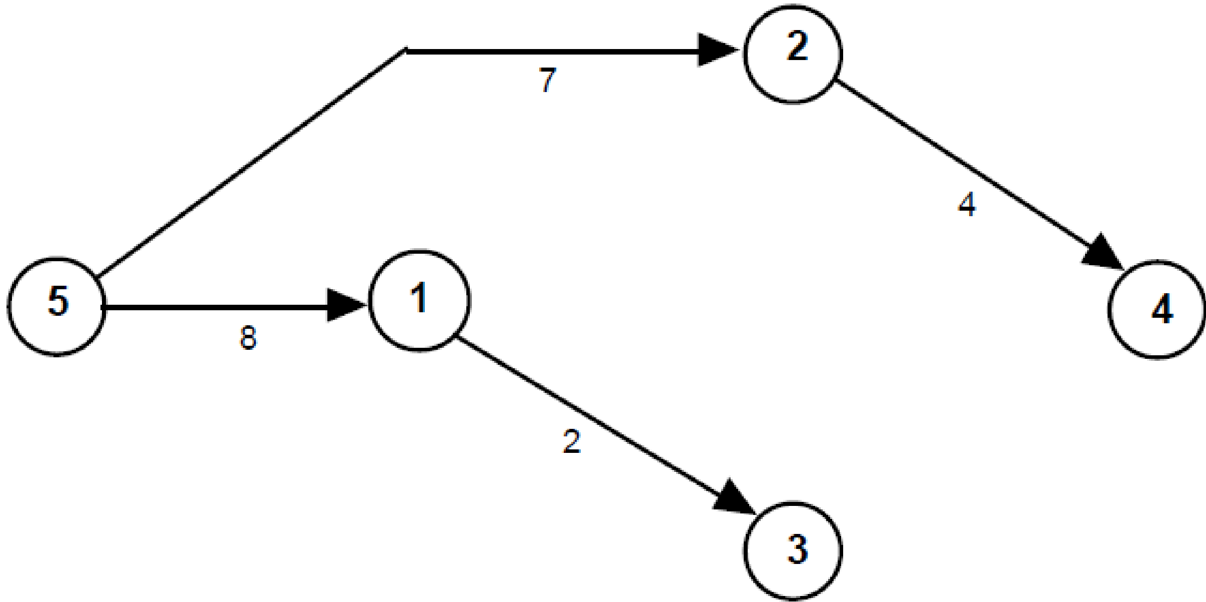


Fig b. Basis tree for n_b = [2, 4, 7, 8]

Non-basic Arcs

The arcs not selected as basic are, by definition, the non-basic arcs. In the bounded variable simplex method of general linear programming, non-basic variables take the value 0 or the value of their upper bound when defining a basic solution. We adapt this method to the network flow problem. In a basic solution, each non-basic arc (i, j), has its flow at 0 or u_{ij}, the upper bound. Let n_0 denoted the set of non-basic arcs with 0 flows, and let n_1 denoted the set of arcs with upper bound flow. To represent a specific case graphically we add the members of n_1 to the basis tree as dotted lines. To illustrate, Fig. c shows the basis tree representing n_b= [2, 4, 7, 8], n_1= [5] and n_0 = [1, 3, 6].

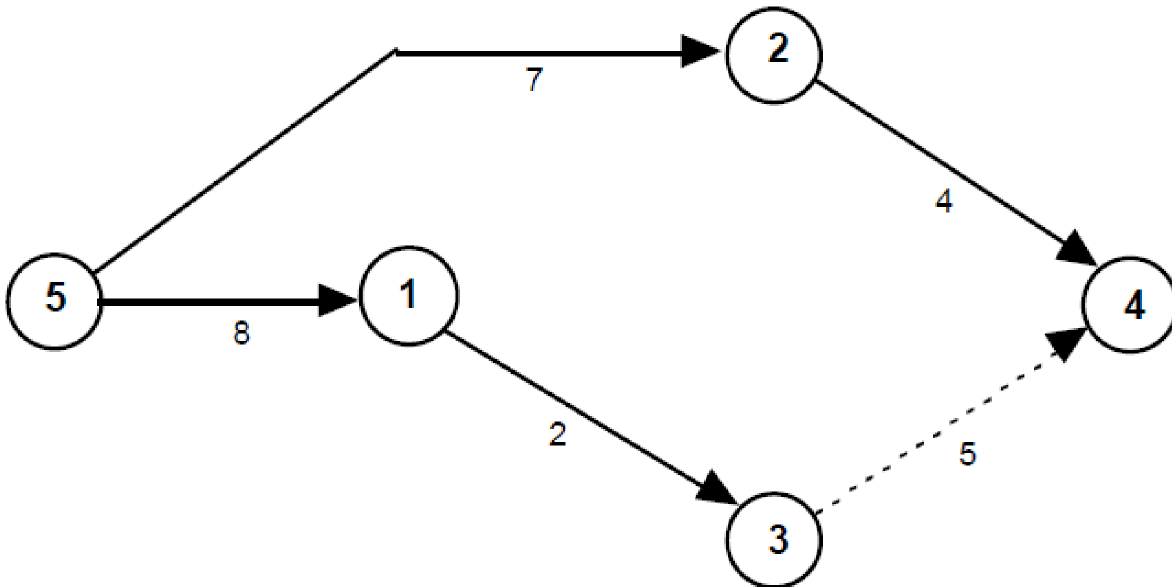


Figure c. Basis tree with a non-basic arc with a flow at its bound

Primal Basic Solution

Given a selection of basic arcs and an assignment of non-basic arcs to either n_0 *or* n_1 , there is a unique assignment of flows to the basic arcs that satisfies the conservation of flow requirement at the nodes. Let x_b be the flows on the basic arcs and x_1 be the flows on the arcs in n_1. The flows on the arcs in n_0 are zero.

To solve for the basic arc flows we must first adjust the external flows for the flows in the non-basic arcs at their upper bounds. One way to do this is to take each member of n_1 in turn. Say arc (i, j) is at its upper bound. To accommodate the flow on arc (i, j) for the following calculations, we reduce the external flow at its origin node *i* by u_{ij} and increase the external flow at its terminal node *j* by u_{ij}. For the network in Fig. a, with the external flow vector:

$$\mathbf{b} = [2, 1, 0, -5, 2].$$

Primal Simplex Algorithm for the Minimal Cost Flow Problem

This part describes the adaptation of the primal simplex algorithm for solving network flow program; that is, the problem of finding the optimal flow distribution for the minimum cost flow network model. We begin with a statement of the algorithm.

Algorithm

Step 1: Start with a primal feasible basis tree consisting of the arcs in the set n_b and the sets $n_0\ and n_1$ corresponding to non-basic arcs at their lower and upper bounds, respectively. Compute the primal solutions and node potential. If necessary, use the phase 1 procedure described below to find an initial feasible basis.

Step 2: Compute the reduced costs, $c_{ij}^{\pi} = c_{ij} + \pi(i) - \pi(j)$, for all nonbasic arcs, (i, j).

Step 3: If one of the following conditions holds for each non-basic arc (i, j), *stop* with the optimal solution.

Optimality conditions:

$$if\ x_{ij} = 0\ then\ c_{ij}^{\pi} \geq 0$$

$$if\ x_{ij} = u_{ij}\ then\ c_{ij}^{\pi} \leq 0$$

Otherwise, select some non-basic arc that violates this condition and call it the entering arc.

Step 4: Find the increase or decrease in the entering arc that will either drive it to its opposite bound or drive some basic arc to one of its bounds. If the entering arc is driven to its bound, go to Step 3. If a basic arc is driven to one of its bounds, let that be the leaving arc.

Step 5: Change the basis by removing the leaving arc and adding the entering arc. Compute the primal solutions and node potential associated with the new basis and go to Step 2.

Initial Solution

As in general linear programming, we have the problem of finding an initial basic feasible solution for the primal simplex method. We use a two-phase approach where an artificial arc with unit cost is introduced for every node but the slack node *m*. The artificial arcs and their initial flows are constructed in the following manner.

For each node $i = 1$ to $m - 1$.

a. If node *i* has $b(i) > 0$, introduce an artificial arc from node *i* to the slack node *m* and assign it the flow $b(i)$. The upper bound on the artificial arc is also $b(i)$.

b. If node *i* has $b(i) < 0$, introduce an artificial arc from the slack node *m* to node *i* and assign it the flow $-b(i)$. The upper bound on the artificial arc is also $-b(i)$. The costs on the artificial arcs depend on the phase of the solution algorithm. Figure 1 shows the example network; Fig.2 shows the spanning tree created by introducing artificial arcs. The tree is rooted at slack node 5. During phase 1 the two networks in the figure are combined. The resultant network consists of 12 arcs but only the unit arc costs on the artificial arcs are used at this stage in the computations. The arcs in the original network carry a unit cost of zero in phase 1.

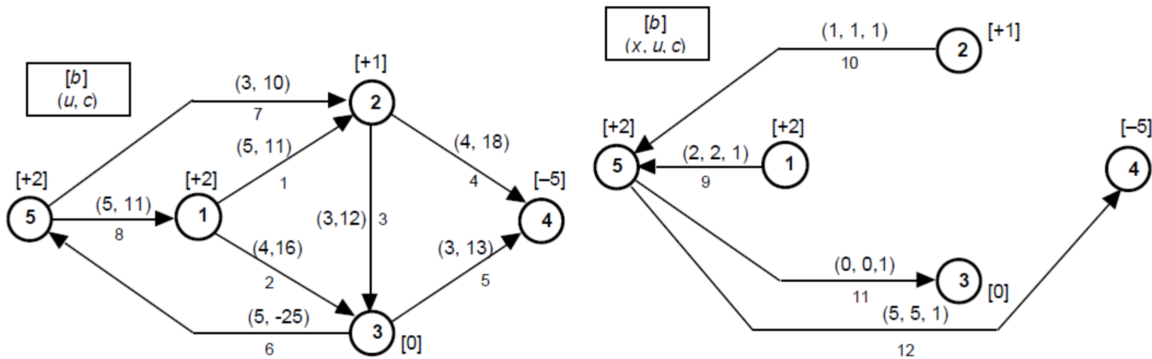


fig 1. Example network **fig 2.**Artificial arcs

Construction of artificial arcs for phase 1

As in general linear programming, we will apply the simplex algorithm in two phases as follows.

Phase 1

1. Assign an arc cost of +1 to each of the artificial arcs and a cost of 0 to each of the original arcs.

2. using the artificial arcs as the initial basis, solve the minimum cost flow problem with the primal simplex algorithm. If the total cost at optimality is greater than zero, an artificial arc has nonzero flow; stop, there is no feasible solution to the original problem. If the total cost at optimality is zero, all the artificial arcs have zero flow and a feasible solution has been found. Proceed with phase 2.

Phase 2

3. Assign the original arc costs to the original arcs. Delete non-basic artificial arcs and assign 0 costs and 0 capacities to each artificial arc remaining in the basis. Starting with the basic solution found in phase 1; solve the network problem with the primal simplex algorithm.

The two-phase procedure applies the primal simplex algorithm twice. In phase 1, a basic feasible solution is found if one exists. Starting with this solution, phase 2 works towards optimality. In most cases all the artificial variables are driven from the basis during phase 1; however, it is conceivable that some artificial arcs remain. Setting their capacity to zero assures that the flows in these arcs remain at zero during phase 2.

Finding the Entering Arc

The reduced cost, c_{ij}^{π} for a non-basic arc (i,j) is the cost of increasing the flow on that arc by one unit, and can be expressed as:

$$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$$

The three terms on the right-hand side of the equation include the unit cost of flow on arc (i, j) c_{ij}; the cost, $\pi(i)$ of bringing a unit of flow to node i from the slack node through a path defined by the basis tree, and the decrease in cost, $-\pi(j)$, achieved by reducing the flow through the basic path to node j. For a non-basic arc with flow at its upper bound, c_{ij}^{π} is the savings in cost (marginal benefit) associated with reducing the flow on arc k by one unit. If the solution is optimal, one of the following conditions must hold for each non basic arc (i, j):

$$if\ x_{ij} = 0\ then\ c_{ij}^{\pi} \geq 0$$

$$if\ x_{ij} = u_{ij}\ then\ c_{ij}^{\pi} \leq 0$$

When the simplex method is applied to the network flow problem, a non basic arc with flow at zero or its upper bound may enter the basis when it fails to satisfy the optimality condition. We select the entering arc with the largest reduced cost.

For large networks, this rule is not efficient because it requires a search over all non basic arcs at every iteration. Alternatives include choosing the first non basic non optimal arc encountered or selecting an arc from some list of candidates

Finding the Leaving Arc

After an arc has been selected to enter the basis, we move to an adjacent basic feasible solution by increasing or decreasing the flow on the entering arc. In this operation, the flows in the basic arcs in the cycle formed by the entering arc must also change in order to maintain conservation of flow at the nodes. The flow changes in the entering arc by an amount that just drives the flow on one of the basic arcs to zero or to its upper bound, or drives the flow on the entering arc to zero or to its upper bound.

By doing so we check the optimality condition after each process if optimality condition is satisfied we stop otherwise we proceed until the optimality condition is satisfied.

Chapter 4

The minimum cost maximum flow problem

Given a flow network *G*= (*N; A; c; u; s; t*) with prices, capacities, a source and a sink. The minimum-cost maximum-flow problem is to compute a maximum flow of minimum cost. We want the maximum flow whose cost is as small as possible. Costs can either be positive, negative, or zero. However, if an edge (i, j) and its reversal (j, i) both appear in the graph, their costs must sum to zero. $c_{ij} = -c_{ji}$. Otherwise, we could make an infinite profit by pushing flow back and forth along the edge.

Each augmentation step in the standard Ford-Fulkerson algorithm both increases the value of the flow and changes its cost. If the total cost of the augmenting path is positive, the cost of the flow decreases; conversely, if the total cost of the augmenting path is negative, the cost of the flow decreases. We can also change the cost of the flow without changing its value, by augmenting along a directed *cycle* in the residual graph. Again, augmenting along a negative-cost cycle decreases the cost of the flow, and augmenting along a positive-cost cycle increases the cost of the flow.

Optimality condition

It follows immediately that a flow *f* is a minimum-cost maximum flow in *G* if and only if the residual graph G_f has no directed paths from *s* to *t and no negative-cost cycles*.

Solution procedures

We can compute a min-cost max-flow using the so-called *cycle cancelling* algorithm first proposed by Morton Klein in 1967. The algorithm has two phases;

in the first, we compute an arbitrary maximum flow *f*, using any method we like. The second phase repeatedly decreases the cost of f by augmenting f along a negative-cost cycle in the residual graph G_f , until no such cycle exists. As in Ford-Fulkerson, the amount of flow we push around each cycle is equal to the minimum residual capacity of any edge on the cycle. In each iteration of the second phase, we can use a modification of shortest path algorithm (often called "Bellman-Ford") to find a negative-cost cycle in $O(VA)$ time. To bound the number of iterations in the second phase, we assume that both the capacity and the cost of each edge is an integer.

As with the Ford-Fulkerson algorithm, the running time is exponential in the complexity of the input, and it may never terminate if the capacities and/or costs are irrational. Like Ford-Fulkerson, more careful choices of which cycle to cancel can lead to more efficient algorithms. Unfortunately, some obvious choices are hard to compute, including the cycle with most negative cost and the negative cycle with the fewest edges. In the late 1980s, Andrew Goldberg and Bob Tarjan developed a min-cost flow algorithm that repeatedly cancels the so-called *minimum-mean cycle*, which is the cycle whose *average cost per edge* is smallest.

The mathematical model of maximum flow minimum cost is divided into phases

$1^{st}\ phase$

Maximize *v*

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v & for\ i = s \\ 0 & for\ all\ \ i \in N - \{s\ and\ t\} \\ -v & for\ i = t \end{cases}$$

$$0 \le x_{ij} \le u_{ij} \quad {}_{for\ each\ (i,j)\in A}$$

After we found the maximum value v then we solve the minimum cost flow as follows.

$$2^{nd}\ phase$$

$$Minimize \sum_{(i,j)\in A} c_{ij}\, x_{ij}$$

Subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v & for\ i = s \\ 0 & for\ all\ \ i \in N - \{s\ and\ t\} \\ -v & for\ i = t \end{cases}$$

$$0 \le x_{ij} \le u_{ij} \quad {\scriptstyle for\ each\ (i,j)\in A}$$

The min-cost circulation problem

Consider a network without a source or a sink. We can define a flow in this network, as long as it is balanced at every node in that network. This kind of flow is called a circulation. The cost of a circulation is defined identically with the cost of a flow.

Observation: Any circulation can be decomposed entirely into cycles. The cost of a circulation x can be expressed as the sum of the costs of all cycles in a decomposition of x.

A minimum cost circulation is a circulation of the smallest possible cost. Note that there is no restriction on the flow through the network. For example, if all costs are positive, the minimum circulation has no flow on all edges. On the other hand, if there are negative cost cycles in the network, the minimum circulation has negative costs and flow has to exist on the edges of the cycle.

Claim: Finding the minimum cost maximum flow of a network is an equivalent problem with finding the minimum cost circulation.

Proof: First, we show that min-cost max-flow can be solved using min-cost circulation. Given a network G with a source s and a sink t, add an edge (t, s) to the network such that $u_{ts} = \infty \quad c_{ts} = -Mc$. Where M is the number of arcs in the network and c is the largest cost in the network. The minimum cost circulation in the new graph will use to the maximum the very inexpensive newly added edge. Any path from s to t forms a negative cost cycle together with (t, s), since $-Mc$ is greater than the cost of any such path. This guarantees that we obtain a maximum flow from s to t "included" in the circulation of the new network. Among all maximum flows, this one is also of minimum cost. All the maximum flows use (t, s) at the same capacity, so they use the edge (t, s) at the same cost. This means that the minimum cost circulation has to be minimum cost on the section from s to t, which makes the max-flow also min-cost. Another reduction from min-cost max-flow to min-cost circulation is to find any maximum flow in the network, regardless of the costs, then find the min-cost circulation in the residual graph. We claim that the resulted flow is a min-cost max-flow. This is because the difference between two max-flows is a circulation, and the cost of that difference circulation is the difference between the costs of the two max-flows. Given x, the initial max-flow, and x^*, the resulting maximum flow, $x - x^*$ is a min-cost circulation in the residual network G_f if and only if x^* is a min-cost max-flow. The second part of the proof is showing that min-cost circulation reduces to min-cost max-flow. Consider a network G for which we want to find a min-cost circulation. Add a source s and a sink t to the network, without any edges to the rest of the network. The maximum flow in this network is 0; therefore the min-cost max-flow is

actually a min-cost circulation. We conclude then that min-cost max-flow and min-cost circulation are equivalent problems.

Converting min-cost max-flow into min-cost circulation: just put an edge from t to s of infinite capacity and large negative cost. Then the min-cost circulation will automatically route as much flow as possible from s to t in the original graph. The analog to the Ford-Fulkerson algorithm for this problem is Klein's Cycle-Canceling algorithm:

Mathematical formulation of minimum cost circulation problem

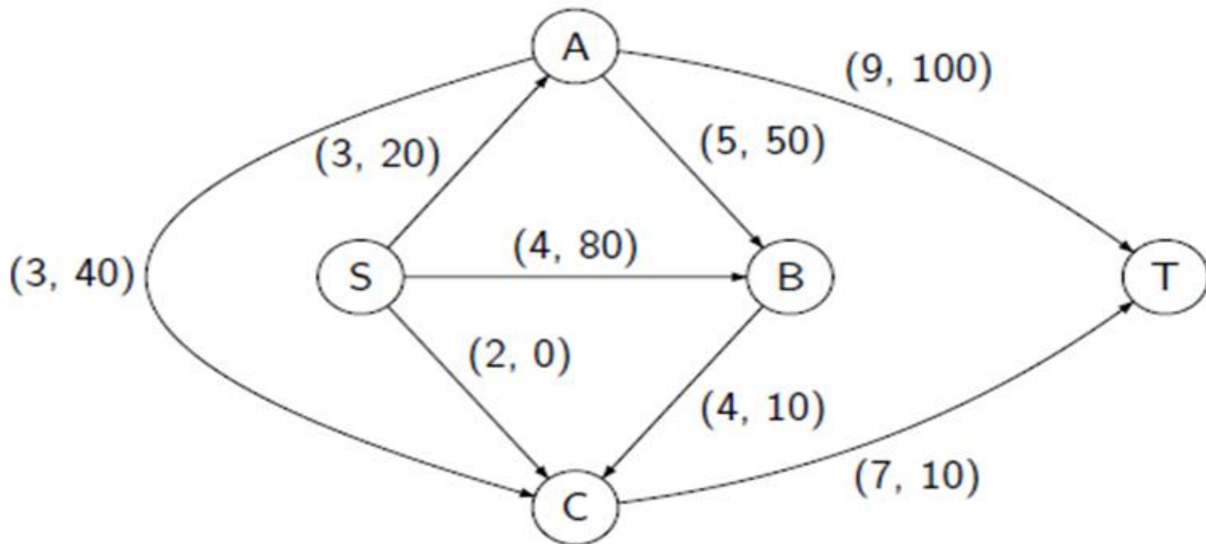
$$Minimize \sum_{(i,j)\in A} c_{ij} x_{ij}$$

Subject to

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

$$0 \leq x_{ij} \leq u_{ij} \quad for\ each\ (i,j)\in A$$

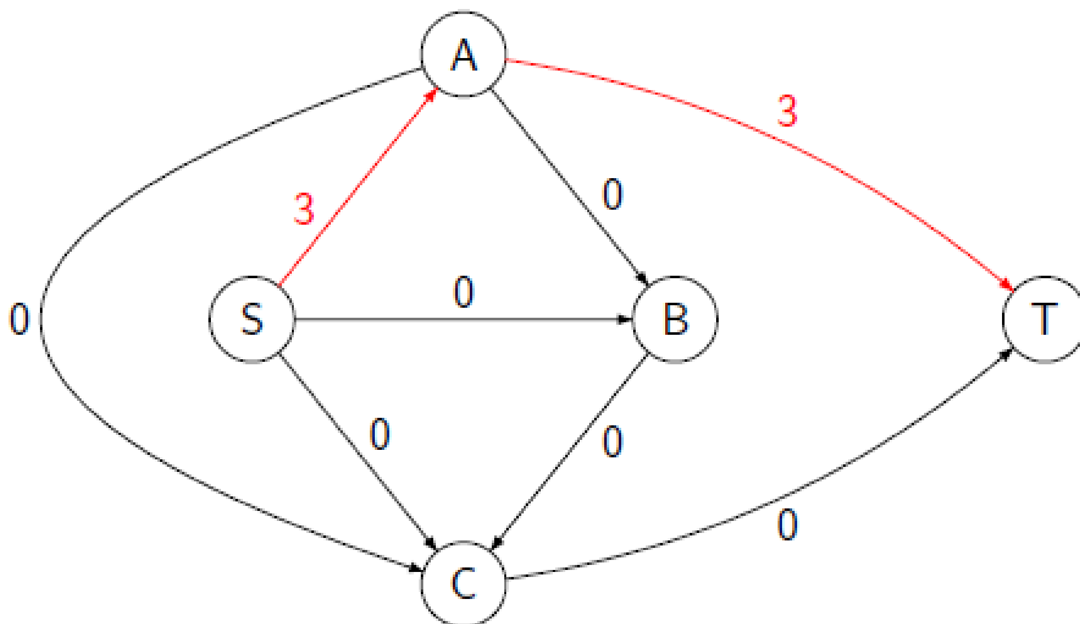
Example; find the maximum flow with a minimum cost in the following graph.



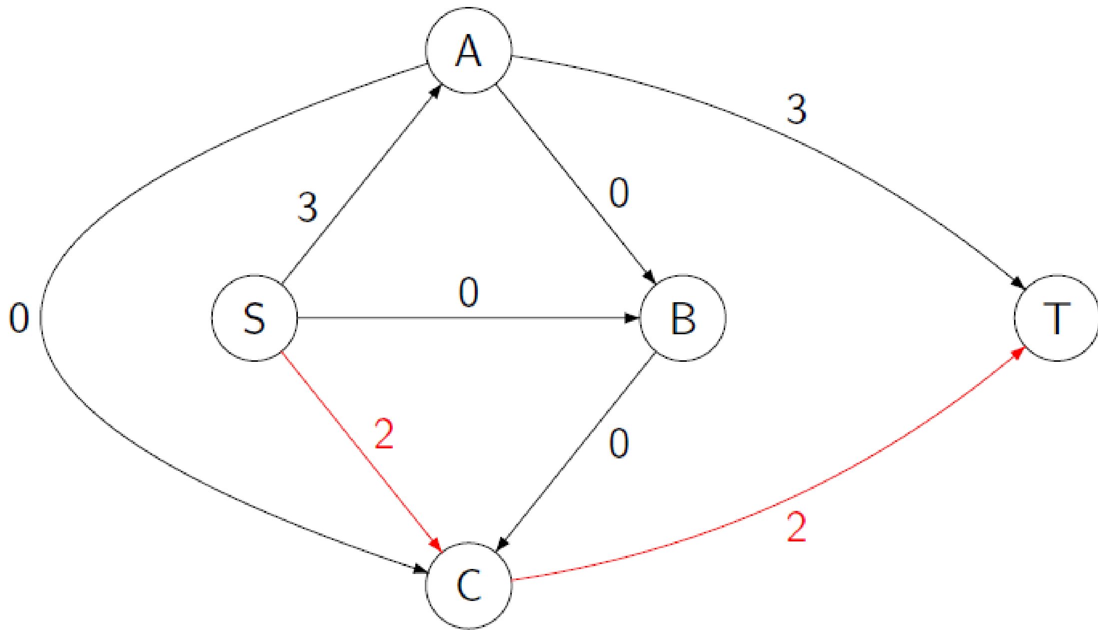
The numbers on the arc represents (capacity (u_{ij}), cost(c_{ij}))

We start by computing maximum flow as follows

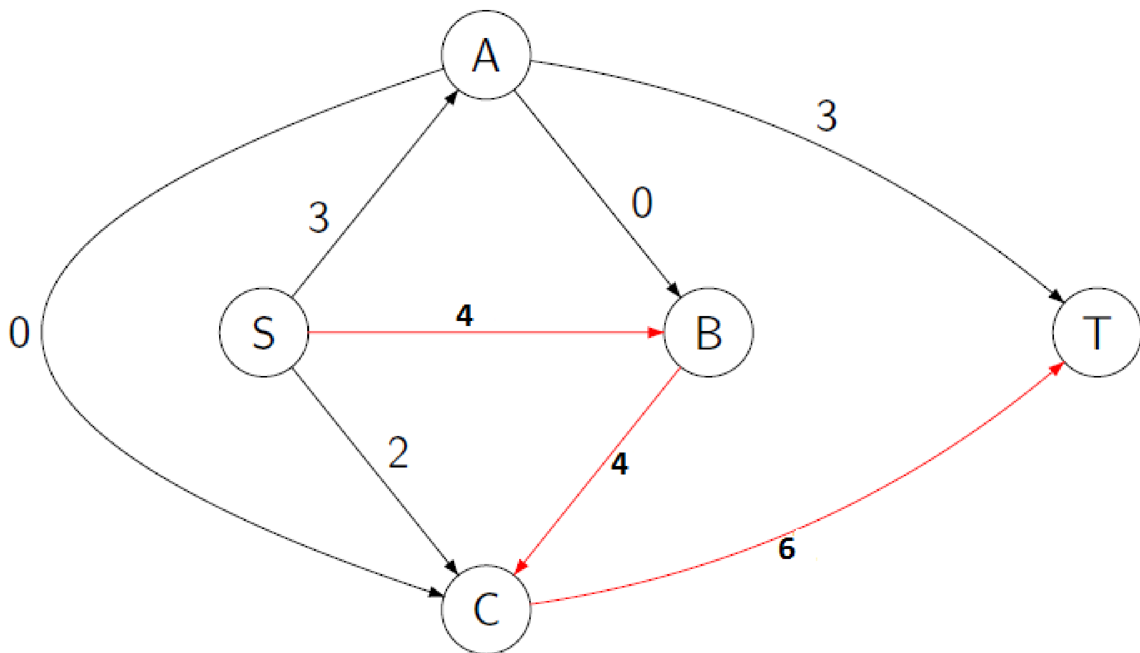
Flow = 3



Flow = 5



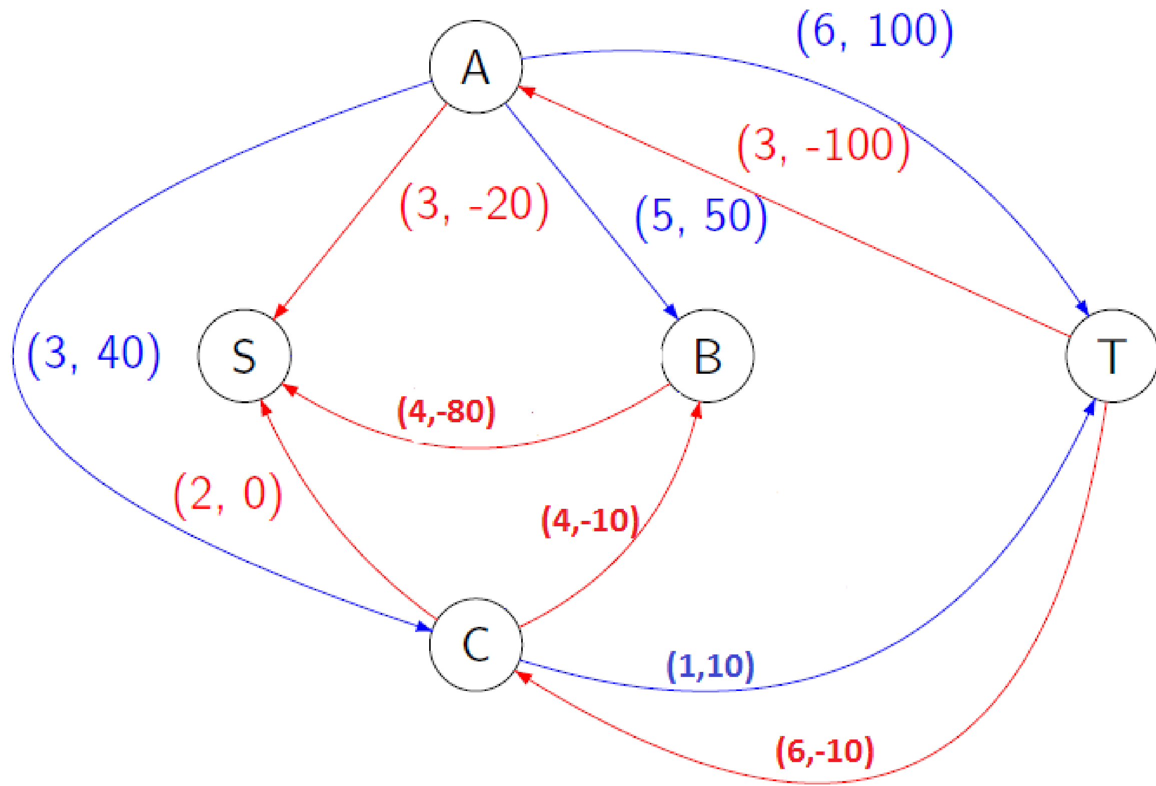
Flow = 9

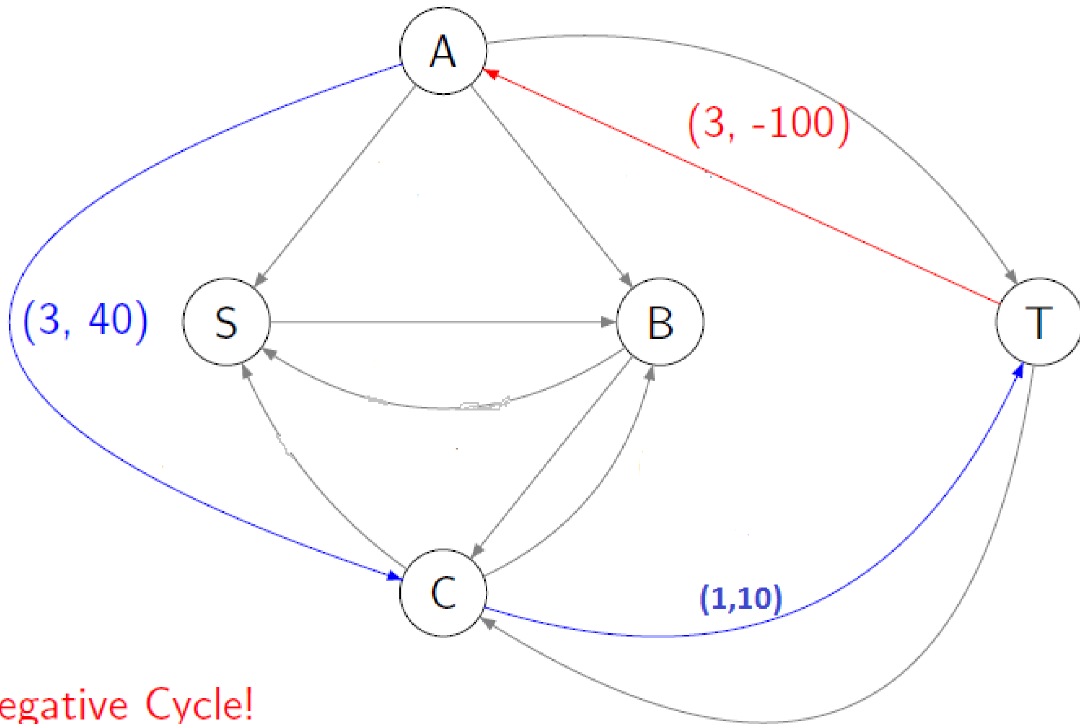


TOTAL COST: 3* 20+4* 80+2*0+3* 100+4*10+6* 10 =

60 + 320 + 0 + 300 + 40 + 60 = 780

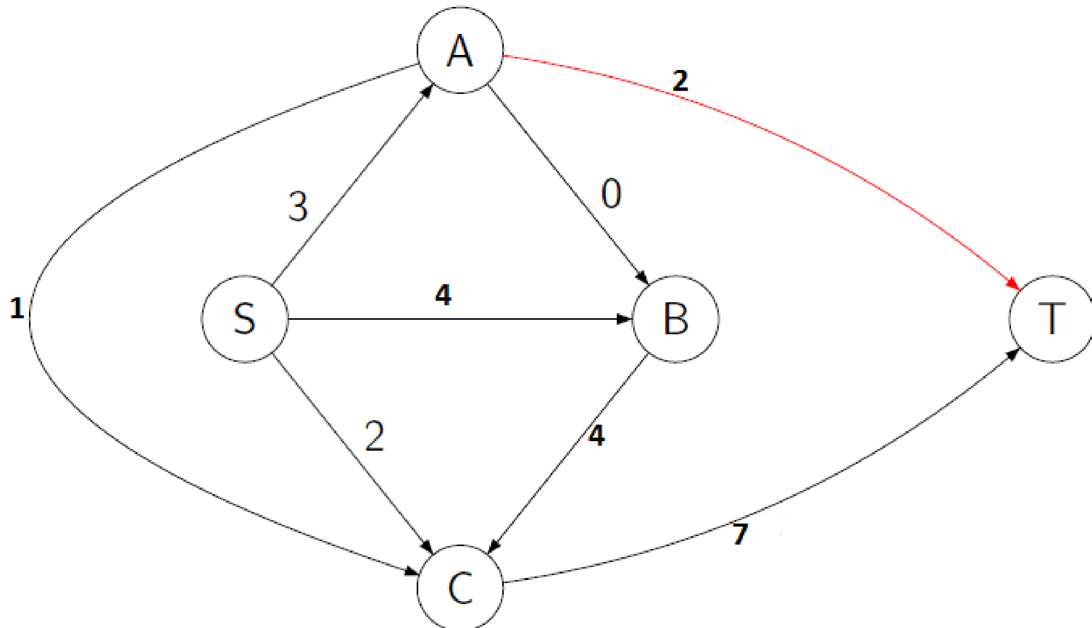
Now we compute the residual network as follows (r_{ij}, c_{ij})



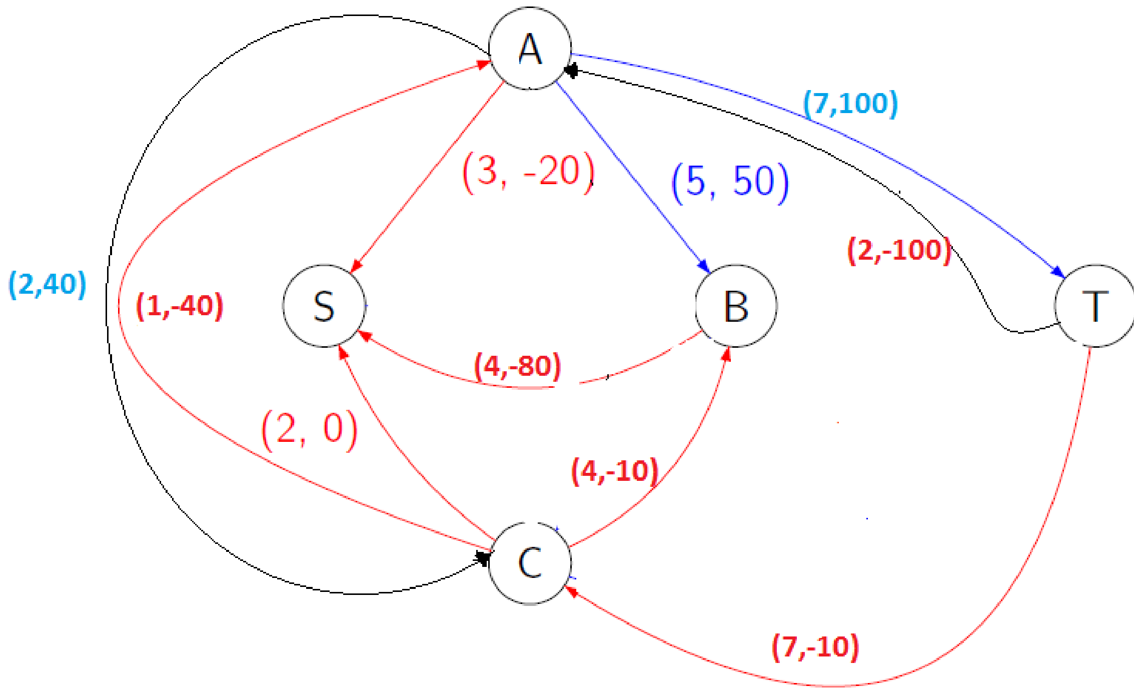


We update the flow as follows

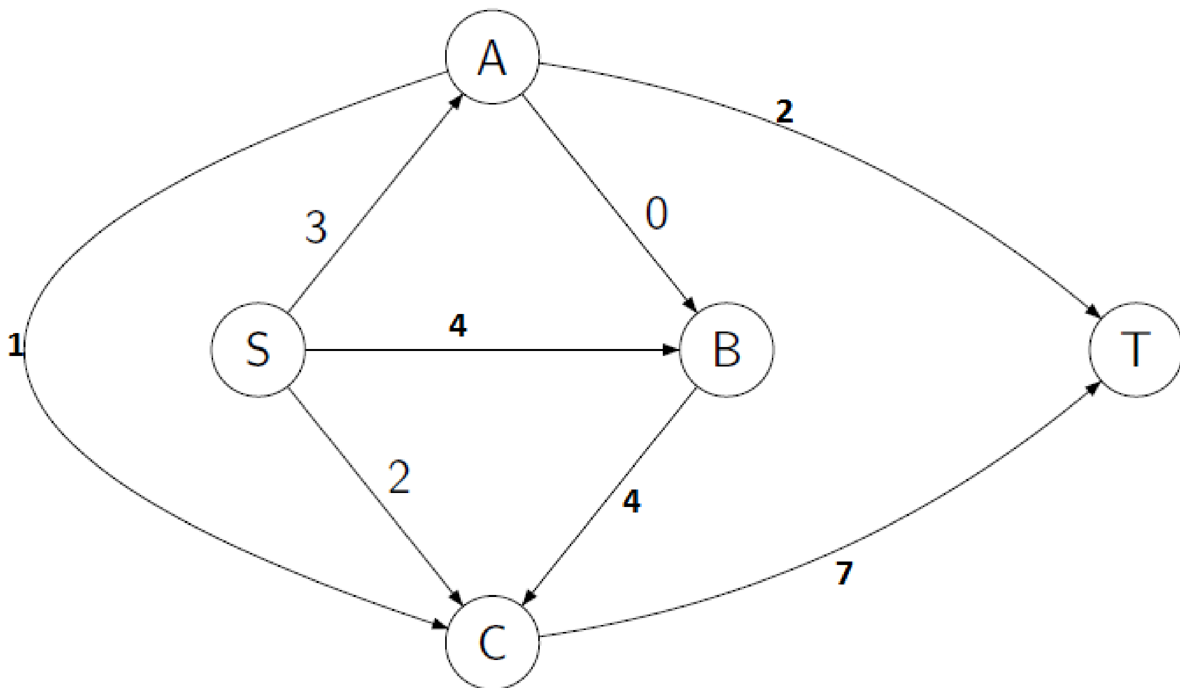
First we compute minimum of {1, 3, 3} =1.and we augment a flow value of 1in the cycle (A, C, T). And we find the next residual graph



Residual network after augmentation



No negative directed cycle is detected therefore it is optimal. With a cost of {3*20+2*100+2*0+4*80+4*10+1*40+7*10 =730}.The flow is as shown in the graph below



Reference

[1]. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Network flows (Theory, algorithms and applications. Prentice Hall, 1993.

[2]. L.R. Ford, Jr. and D.R. Fulkerson. Flows in Networks. Princeton University Press, Princeton, NJ, 1962.

[3]. Bazarra, M.S., Jarvis, J.J., and Sherali, H.D. (1977) Linear Programming and Network Flows (John Wiley & Sons)

[4]. Michel Gondran , Michel Minoux (1984) GRAPHS AND ALGORITHMS (John Wiley & Sons)

[5]. EUGENE L. LAWLER (1976) Combinatorial Optimization: Networks and Matroids (Holt, Rinehart and Winston)