# A NEW APPROACH IN THE MAXIMUM FLOW PROBLEM

A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL
ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENT
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Aysen Eren
July, 1989

# A NEW APPROACH IN THE MAXIMUM FLOW PROBLEM

A THESIS

SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL

ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENT

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Aysen Eren

July, 1989

Aysen Eren

T
57.85
Er 35
1989

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Mustafa Akgül (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Ömer Kirca

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Osman Oguz

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Barbaros Tansel

Approved for the Institute of Engineering and Sciences:

M. Baray

Prof. Mehmet Baray
Director of Institute of Engineering and Sciences

# ABSTRACT

A NEW APPROACH IN THE MAXIMUM FLOW PROBLEM

*AYSEN EREN*

M.S. in Industrial Engineering

Supervisor: Assoc. Prof. Mustafa Akgul

July, 1989

In this study, we tried to approach the maximum flow problem from a different point of view. This effort has led us to the development of a new maximum flow algorithm. The algorithm is based on the idea that when initial quasi-flow on each edge of the graph is equated to the upper capacity of the edge, it violates node balance equations, while satisfying capacity and non-negativity constraints. In order to obtain a feasible and optimum flow, quasi-flow on some of the edges have to be reduced. Given an initial quasi-flow, positive and negative excess, and, balanced nodes are determined. Algorithm reduces excesses of unbalanced nodes to zero by finding residual paths joining positive excess nodes to negative excess nodes and sending excesses along these paths. Minimum cut is determined first, and then maximum flow of the given cut is found. Time complexity of the algorithm is $o(n^2m)$. The application of the modified version of the Dynamic Tree structure of Sleator and Tarjan reduces it to o(nmlogn).

# ÖZET

## MAKSİMUM AKIŞ PROBLEMİNE YENİ BİR YAKLAŞIM

Ayşen Eren
Endüstri Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Doç. Mustafa Akgül
Temmuz, 1989

Bu çalışmada, maksimum akış problemine değişik bir görüş noktasından yaklaşmayı denedik. Bu uğraş, bizi yeni bir maksimum akış algoritmasını geliştirmeye götürdü. Algoritma, serimin her ayrıtındaki ilk akışımsının, ayrıtın üst kapasitesine eşitlendiği zaman, bunun kapasite ile eksi olmama kısıtlarını sağlarken, düğüm denge eşitliklerini bozması fikrini temel alır. Olurlu ve en iyi bir akış elde etmek için, bazı ayrıtlar üzerindeki akışımsılar azaltılmalıdır. Verilen bir ilk akışımsıya göre, artı ve eksi fazlalık ile dengelenmiş düğümler belirlenir. Algoritma, artı fazlalık düğümlerini eksi fazlalık düğümlerine bağlayan artık yollarını bulup, bu yollar boyunca fazlalıkları göndererek, dengelenmemiş olan düğümlerin fazlalıklarını sıfıra indirir. İlk önce, en küçük kesit belirlenir ve sonra verilen kesitin maksimum akışı bulunur. Algoritmanın zamansal karmaşıklığı $O(n^2m)$'dir. Sleator ile Tarjan'ın Dinamik Ağaç yapısının değiştirilmiş şeklinin uygulanması bunu $O(nm \log n)$'e düşürür.

# ACKNOWLEDGEMENT

I would like to thank to Assoc. Prof. Mustafa Akgül for his supervision, guidance, valuable suggestions , comments, and patience throughout the development of this thesis.

I am grateful to the members of Jury: Assoc. Prof. Ömer Kirca, Assoc. Prof. Osman Oguz, and Asst. Prof. Barbaros Tansel for their valuable comments.

My sincere thanks are due to Levent Aksoy for giving me his morale support to complete this study, and for his great help in correcting and printing my thesis.

# TABLE OF CONTENTS

## LIST OF FIGURES

CHAPTER I

# INTRODUCTION AND LITERATURE REVIEW

The problem of finding a maximum flow in a directed graph with edge capacities has interested many people in various fields. It is a well-defined problem and has been known for many years. Its applications cover a wide area extending from minimum cost flow problems to the analysis of transmission networks.

The problem can be stated and formulated as a Linear Program. Given a directed graph, with capacities on the edges and two distinct nodes, a source s and a sink t, the maximum flow problem is to maximize the flow that can be sent from source to sink through the network. A flow is an assignment of real numbers to edges of the graph which satisfy;

i. capacity constraints which make sure that flow on each edge is always smaller than or equal to edge capacity,

ii. node balance constraints which state that total flow coming into each node should pass it without any loss or gain in value, and,

iii. non-negativity constraints.

Let v be the total flow passing through edges of the network. Let f, u and A denote respectively flow, capacity vectors, and the incidence matrix. Then the LP model of the maximum flow problem is;

$$\text{MAX} \quad v$$

s.t.

$$A * f = \begin{cases} -v & i = \text{source} \\ v & i = \text{sink} \\ 0 & \text{otherwise} \end{cases}$$

$$f \leq u$$

$$f \geq 0$$

The first step toward the establishment of the max-flow min-cut theorem has been taken by Menger [5,17] in the early thirties. The theorem of Menger states that if source and sink are disconnected by removing k nodes of the graph, meaning that all paths from source to sink should pass thru at least one of k nodes, there are k source-sink internally disjoint paths on the graph. Originally stated for undirected graphs, it is directly applicable to the theory of max-flow min-cut, when formulated in terms of digraphs.

The max-flow min-cut theorem has been established by two independent groups of people. Shannon, Feinstein and Elias [7] and Ford-Fulkerson [8] have stated the theorem in 1956. Then the first max-flow algorithm has been developed by Ford-Fulkerson in the same year. In the following years, many fast and efficient algorithms have appeared in the literature. Table 1 gives these algorithms in chronological order. Time bounds are given in terms of n -number of nodes, m -number of edges and N -maximum edge capacity.

| number | year | developer | time bound |
|---|---|---|---|
| 1 | 1956 | FORD-FULKERSON [8,9] | - |
| 2 | 1969 | EDMONDS-KARP [6] | $o( nm^2 )$ |
| 3 | 1970 | DINIC [4,15] | $o( n^2m )$ |
| 4 | 1974 | KARZANOV [16] | $o( n^3 )$ |
| 5 | 1977 | CHERKASKY [3] | $o( n^2m^{1/2} )$ |
| 6 | 1978 | MALHOTRA,KUMAR,MAHESHWARI [18] | $o( n^3 )$ |
| 7 | 1978 | GALIL [11] | $o( n^{5/3}m^{2/3} )$ |
| 8 | 1978 | GALIL,NAAMAD; SHILOACH [12;13] | $o( nm(\log n)^2 )$ |
| 9 | 1980 | SLEATOR,TARJAN [21] | $o( nm\log n )$ |
| 10 | 1982 | SHILOACH,VISHKIN [20] | $o( n^3 )$ |
| 11 | 1983 | GABOW [10] | $o( nm\log N )$ |
| 12 | 1984 | TARJAN [22] | $o( n^3 )$ |
| 13 | 1986 | GOLDBERG,TARJAN [13] | $o( nm\log(n^2/m))$ |
| 14 | 1987 | AHUJA,ORLIN [2] | $o( nm+n^2\log N )$ |
| 15 | 1987 | AHUJA,ORLIN [1] | $o( nm\log N )$ |
| 16 | 1988 | GOLDFARB,HAO [14] | $o( n^2m )$ |
| 17 | 1988 | GOLDBERG,GRIGORIADIS, TARJAN [14] | $o( nm\log n )$ |

Table 1. Max-flow algorithms in chronological order.

Five of the listed algorithms have great importance and impact on the history of max-flow algorithms. They are Ford-Fulkerson, Edmonds-Karp, Dinic, Karzanov and Goldberg-Tarjan algorithms.

Being the first max-flow algorithm, Ford-Fulkerson algorithm is of importance. The algorithm starts with any feasible flow and augments or increases the flow by finding a source-to-sink directed path in the residual graph. The algorithm stops when there is no augmenting path. The proof of optimality is given by exhibiting a cut having the capacity that is equal to the given flow. Their algorithm is also known as Labelling algorithm. The Labelling algorithm is a non-deterministic algorithm to find a source-to-sink directed path in the residual graph in which nodes to be scanned are chosen arbitrarily among the labeled nodes. With integral edge capacities, the upper bound on the number of iterations is obviously equal to the value of the flow. Major deficiency of the algorithm is its exponential complexity. When capacities are irrational, the algorithm may not even be finite and may converge to a non-maximum flow as exhibited by Ford-Fulkerson [9].

Thirteen years later, Edmonds and Karp [6] employed breadth-first search method to find shortest residual augmenting paths from source to sink. Their algorithm is the first strongly polynomial maximum flow algorithm; it requires o(nm) augmentations and o(nm$^2$) time.

Dinic's algorithm [4] works on a layered graph which contains all the shortest paths in the current residual graph. In a layered graph, source and sink are placed in distinct layers by themselves. There are other layers between layers of source and sink. Each edge in such a graph goes from one layer to the next layer. Thus all source-sink paths have the same length and the layered graph is acyclic. In each layered graph one finds a maximal or blocking flow. A blocking flow in a layered graph assures that in the next layered graph, length of any source-to-sink path is increased by at least one. Thus there are at most ( n-1 ) layered graphs. By using depth-first search, Dinic finds a blocking flow in o ( nm ) time, giving total bound of o ( $n^2m$ ).

Dinic's algorithm has been modified by many people leading to the development of more efficient and faster algorithms. Sleator and Tarjan [21] designed a new data structure, Dynamic Tree, particularly for Dinic's algorithm. With Dynamic Trees, Sleator-Tarjan succeeded to reduce the running time of the maximum flow algorithm from o($n^2m$) to o(nmlogn). In 1987, Ahuja and Orlin [1] modified Dinic's algorithm and obtained ( nmlogN ) time bound by using a scaling algorithm and utilizing a distance label function instead of maintaining layered graphs. Their bound is weakly polynamial due to dependence on input parameter N which is the maximum edge capacity. However their algorithm requires no complex data structure. Distance label function has been introduced by Goldberg and Tarjan in 1986 [13]. It is a function from node set to positive integers. Distance label of a node is the minimum length of augmenting path from a reference node to that node.

In 1974, Karzanov [16] introduced his algorithm to find a blocking flow in a layered graph and approached maximum flow problem from a different point of view. Until that time, developed algorithms were based on the idea of augmenting flow on residual paths. Residual paths which go from source to sink are found one by one and as much a flow is sent thru each path as possible. The maximum amount of flow which can be sent along a path is equal to the minimum of residual capacities of edges on that path. When a flow is augmented, it saturates at least one edge. Hence no more flow can be sent along that path. Such a path is called saturated or blocked, and such a flow, a blocking flow. In his algorithm, Karzanov uses the idea of Preflow to determine a blocking flow or a maximal flow of a layered graph. The Preflow concept has been introduced by Karzanov. It is a mapping from the edge set to non-negative real numbers that satisfies two types of inequalities. Let p be a Preflow. Then

$$p(\ i,j\ ) \leq \text{capacity}(\ i,j\ ) \quad \text{for all } (\ i,j\ ) \text{ pairs}$$

and

$$\sum_i p(\ i,j\ ) \geq \sum_i p(\ j,i\ ) \qquad \text{for all } j \text{ in the graph}$$

The first inequality makes sure that the Preflow does not disturb capacity constraints. The satisfaction of node balance equations is not guaranteed due to the second type of inequalities. Incoming Preflow may be greater than outgoing Preflow at any node. Karzanov's algorithm consists of phases. Before each phase starts, a layered graph is constructed, then a Preflow is sent through edges of the graph by pushing as much flow from source toward nodes as possible. Unbalanced nodes or nodes with positive excesses are taken one by one and balanced by reducing flows coming into them. A phase ends when all nodes are balanced and a maximal flow in the layered graph is obtained. Dinic's and

Karzanov's algorithms differ in the way they find maximal flows. Time bound of the algorithm is $o( n^3)$ and it is the best for dense graphs.

In 1986 , Goldberg and Tarjan used the Preflow idea of Karzanov. They also maintained distance label function and Dynamic Tree structure and reduced the order to $o( nm\log(n^2/m) )$. It is the best bound both for sparse and dense graphs so far.

Recently Goldfarb and Hao developed the first strongly polynomial primal simplex algorithm for the maximum flow problem. They obtain the maximum flow in at most $o( nm )$ flow augmentations or simplex pivots. They employed the rule that is to select an edge which is closest to source node among the non-basic edges which are candidates to enter the basis.

Goldberg, Grigoriadis, and Tarjan [14] have obtained $o(nm\log n)$ time bound for Goldfarb-Hao's primal simplex algorithm by using a modified version of Dynamic Tree data structure of Sleator and Tarjan.

In this study, we approach the classical maximum flow problem from a different point of view and propose a new maximum flow algorithm. Our algorithm originates from Goldberg-Tarjan [13] and feasible distribution algorithms [19]. It is based on the idea that if the flows on edges are assumed to be equal to upper capacities, node balance equations of some nodes will be violated. In order to find an optimum solution, node balance equations have to be satisfied. Our algorithm does precisely that.

Goldberg and Tarjan uses Karzanov's idea of Preflow. We will introduce a quasi-flow concept and use it. Quasi-flow is a special type of flow, satisfying upper capacity and non-negativity constraints on the edges of the graph. In their algorithm, there are positive excess nodes, called active or unbalanced nodes. By reducing flows on paths which connect source to active nodes, all the nodes are balanced. Both positive and negative excess nodes occur due to quasi-flow in our proposed algorithm. Negative excess nodes have opposite properties of positive excess nodes. In order to balance them, we have to find paths from negative excess nodes to positive excess nodes and reduce flow on them. The global framework of the algorithm is very similar to feasible distribution algorithms. Our algorithm makes use of Karzanov's Preflow concept and also defines Postflow. There are two inequalities that are required to make a flow Postflow. First one makes sure that the flow on each edge is between zero and upper capacity of that edge. The second inequality states that the flow going out of each node is greater than or equal to flow coming into it. In that sense, the Postflow can be seen as the reverse of the Preflow.

There are 6 chapters including introduction. The Next chapter contains definitions and notation. Conceptual description of the algorithm is given in the 3rd Chapter. Chapter 4 consists of the algorithm, proofs showing its polynomiality, correctness, and a sample problem. Dynamic tree version of the algorithm is presented in Chapter 5.

Chapter 6 summarizes the study, findings, and future research.

CHAPTER II

# DEFINITIONS AND NOTATION

G = (N,E) defines a directed graph with node set N and edge set E. There are two distinct nodes, one called source, the other called sink. We assume that all edges incident to the source are directed away from the source and all edges incident to the sink are directed into the sink. Thus source node behaves like a real source and can only send flow to other nodes while sink can only receive flow. Let s stand for the source node and t for the sink. The lower capacity of each edge is taken to be zero. Edges have finite upper capacities; u(i,j) stands for the upper capacity of edge (i,j). The proposed algorithm utilizes the concept of a quasi-flow. A quasi-flow is a real valued function g, defined from the edge set to real numbers that satisfies the condition

$$0 \leq g(i,j) \leq u(i,j) \qquad \forall (i,j) \in E.$$

A quasi-flow does not violate edge capacity constraints, however it may violate node balance equations. Therefore it is not, in general, a feasible flow. Initially, the quasi-flow of every edge is equated to its upper capacity.

It is assumed that one artificial edge ( t,s ) links sink to source. Algorithms , using such an edge, for the sake of simplicity, assume an infinite capacity for it. We, for the sake of the algorithm, assign a finite value to the capacity of that edge. That value will be defined soon. Excess function, which will be defined next, makes use of the capacity assigned to ( t,s ) in defining excesses of source and sink nodes. During the first stage, the existence of ( t,s ) is ignored. Consequently, excesses of source and sink nodes remain fixed because of non-existence of ( t,s ). The algorithm uses it in the second stage to balance node equations, after finding the minimum cut.

The algorithm maintains two functions. The first one is the excess function e, defined from node set to real numbers. For any node i, excess of i is the difference between total quasi-flow coming into node i and total quasi-flow going out of node i. Let us formalize the concept and define the excess function as follows;

$$e(i) = \begin{cases} \alpha - \text{Outflow}(s) & \text{if } i = s \\ \text{Inflow}(t) - \alpha & \text{if } i = t \\ \text{Inflow}(i) - \text{Outflow}(i) & \text{o.w.} \end{cases}$$

where

Inflow(i) : sum of quasi-flows of edges coming into node i on G.

Outflow(i) : sum of quasi-flows of edges going out of node i on G.

$\alpha$ : ( $\sum_{\forall i \in N \setminus \{s,t\}} |e(i)|$ + max ( $u_s$, $u_t$}), where $u_s$ is the total capacity of outgoing edges of s and and $u_t$ is the total capacity of incoming edges of t.

The purpose of using $\alpha$ is to make excess function to take a positive value at source node and a negative value at sink node, and make them connected to supersource and supersink, respectively, throughout the first stage. It will help us to define a cut right after the completion of the first stage. That point will be discussed later. In order to make e(s) a positive number and e(t) a negative number, $\alpha$ has to be greater than max{ $u_s$, $u_t$ }. Instead of saying that $\alpha$ is any number greater than max{ $u_s$, $u_t$ }, it is set to be equal to the sum of absolute values of excesses of nodes excluding source and sink and max{ $u_s$, $u_t$ }.

Two artifical nodes are created and added to graph, a supersink ss and a supersource st node. New node set is $N'' = N \cup \{ss,st\}$. Supersink is connected to each negative excess node by an artificial edge directed from st to that node. Similarly, artificial edges directed from positive excess nodes to ss are used to connect ss to the graph. Each artificial edge has a capacity that is equal to the excess of the node, to which it is adjacent. Edge set is enlarged by the addition of artificial edges and denoted by $E''$. The new graph becomes $G'' = ( N'' , E'' )$. Supersink st becomes a negative excess node whose excess is the sum of negative excesses. Supersource ss has a positive excess that is equal to the sum of positive excesses. In the beginning, except these two, nodes with excesses are temporarily balanced.

The algorithm works on residual graphs. $G''_r(N'',E''_r)$

indicates residual graph with node set $N''$ and residual edge set $E''_r$. If there is a quasi-flow on the edge ( i,j ), it results in two residual edges. r(i,j) is the capacity of a forward residual edge directed from i to j and r(j,i) is the capacity of a backward residual edge directed from j to i. r(i,j) and r(j,i) have contrary meanings with respect to edge (i,j). r(i,j) is the additional possible flow increase on edge (i,j) while r(j,i) is the possible flow decrease on edge (i,j). They are defined symbolically as,

$$r(i,j) = u(i,j) - g(i,j)$$
$$r(j,i) = g(i,j)$$

A search tree is costructed by breadth-first search on the residual graph. The algorithm maintains it to determine residual paths connecting supersource to supersink. A search tree can be defined as a special type of distance directed branching. Let us first give the definition of a directed branching. It is a tree in which every node, other than root, has exactly one incoming edge. Hence, every path from the root to any other node in a branching is unique. Besides that, in a search tree every unique path is a shortest path in the residual graph. Search tree concept is very similar to the shortest path tree concept used in Ahuja-Orlin's algorithm [1]. The main point that distinguishes it from our algorithm is that a shortest path tree is a spanning tree whereas a search tree does not have to hold all of the nodes. Our search tree is rooted at supersource. While the algorithm proceeds, constructed tree goes under frequent structural changes. Several tree operations like deletion of edges and addition of new nodes, are carried out to reflect those changes by using distance label function. Let's now introduce the distance label function.

The Distance Label function is the second function used by our algorithm. It is defined from the node set to non-negative integers and the primary reason for using it is to construct and update the tree structure which is under frequent change during the execution of the algorithm. Distance label of node i, d(i), is the minimum number of edges that are on the path of the search tree, connecting i to supersource. The function is valid if it satisfies the following conditions:

i. $d(i) = d(j) + 1$ if $r(j,i) > 0$, $(j,i) \in T$,

ii. $d(i) \geq d(j) + 1$ if $(j,i) \in E''_r$.

C H A P T E R III

# CONCEPTUAL DESCRIPTION OF THE ALGORITHM

Before explaining the algorithm conceptually let us first describe the graph we will work with.

A few additions are made to the original graph G. An artificial edge ( t,s ) with finite lower and upper capacities is assumed to join sink to source. For notational and algorithmic convenience two dummy nodes, supersource and supersink are introduced and as mentioned in Chapter II, graph $G''$ is formed. Then an initial flow called quasi-flow is assigned to the edges of $G''$. Node excesses are determined and then node set is divided into three disjoint subsets: set of positive excess nodes N(+), set of negative excess nodes N(-) and set of balanced nodes N(0). The excesses of the nodes have to be zeroed to obtain a feasible and optimum flow on the graph. Accordingly, the problem definition is modified slightly and classical maximum flow problem becomes problem of sending positive excess of supersource to supersink through residual graph or equivalently sending positive excesses of nodes of N(+) to nodes of N(-) along residual paths.

The structure of the algorithm is quite similar to the feasible distribution algorithm applied to the maximum flow problem. That is investigated in detail by Rockafeller [19]. Before giving evidence for the equivalence of our algorithm and the feasible distribution algorithm, let us first define the feasible distribution algorithm. It is stated as follows: Given the capacity bounds on edges, the supply value of each node is determined by subtracting its inflow from its outflow. Let b(i) be the supply value of node i. Positivity of b(i) indicates that node i behaves like a supply node and sends some amount of flow. Feasible distribution problem is to find a flow f, such that f conserves capacity constraints of the edges and also satisfies supply constraints. The conceptual algorithm designed to solve the feasible distribution problem is modified and applied to the maximum flow problem. In that case, an initial flow x, which satisfies edge capacity constraints is given. Looking at x, supply values or in our words excess values ( $b(i) = - e(i)$ ) of nodes are determined and, N(+) and N(-) are defined. If two sets are empty, initial flow x is feasible and optimal, otherwise excesses of nodes should be zeroed. The painted network algorithm [19] that employes a graph search in a suitably constructed residual graph is applied to reduce excesses to zero.

Our approach is similar to the one discussed above. In the above case, there is no restriction on the initial flow, we assume that the initial quasi-flow on each edge is equal to its upper capacity. Instead of using the painted network algorithm, we propose our algorithm which is easier to understand and manipulate. For convenience and easiness in defining a minimum cut set, instead of applying the algorithm to residual graph of $G''$ once, we repeat our algorithm twice.

The algorithm has two stages. In the first stage, the algorithm ignores existence of (t,s) and finds out residual paths of $G_r''$ which go from N(+) to N(-) and join supersource to supersink. The first stage terminates, when N(+) is disconnected from N(-) in the residual graph shown in Figure III.1. Let us now consider G(+) and G(-) corresponding to residual subgraphs $G_r(+)$ and $G_r(-)$ of disconnected node sets.

G(+) is a subgraph of $G''$ that consists of N(+), $N_+(0)$ that is a subset of balanced nodes which are reachable from N(+), and edges joining any pair of nodes of these sets. G(-) is another subgraph of $G''$. N(-), $N_-(0)$ that is a set

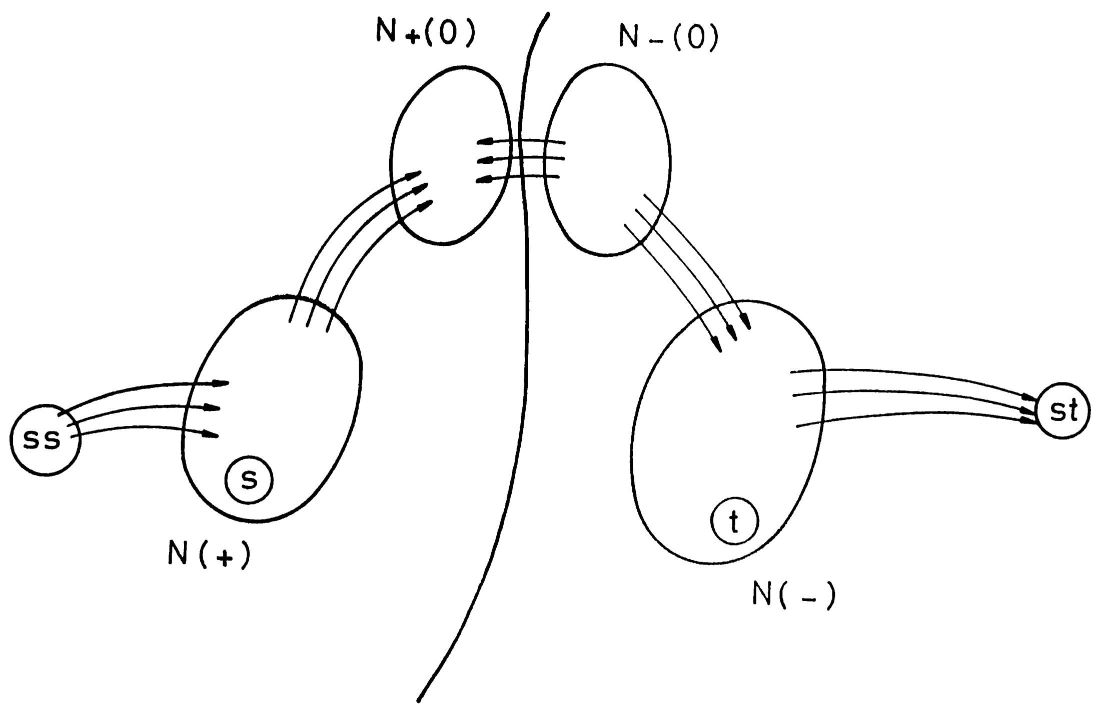

Figure III.1. Node sets of $G''$ and residual edges connecting them at the end of the first stage.

of nodes which can be reached from N(-), and the edges connecting nodes of these sets belong to this subgraph. G(+) and G(-) are connected by an artificial edge (t,s) and a set of edges Q, which consists of the the edges between these subgraphs.See **Figure III.2.** Let us now investigate the flows on G(+) and G(-), and derive some interesting results.

The quasi-flow on edges of G(+) can be redefined as a Preflow. It satisfies the conditions required to make a flow a Preflow. Quasi-flow on each edge is between zero and given upper bound and inflow of each node of G(+) is greater than or equal to its outflow.

Let us now investigate the other subgraph G(-). We call the quasi-flow on G(-) Postflow. It satisfies capacity constraints of edges of G(-) and outflow of each node is greater than or equal to its inflow.

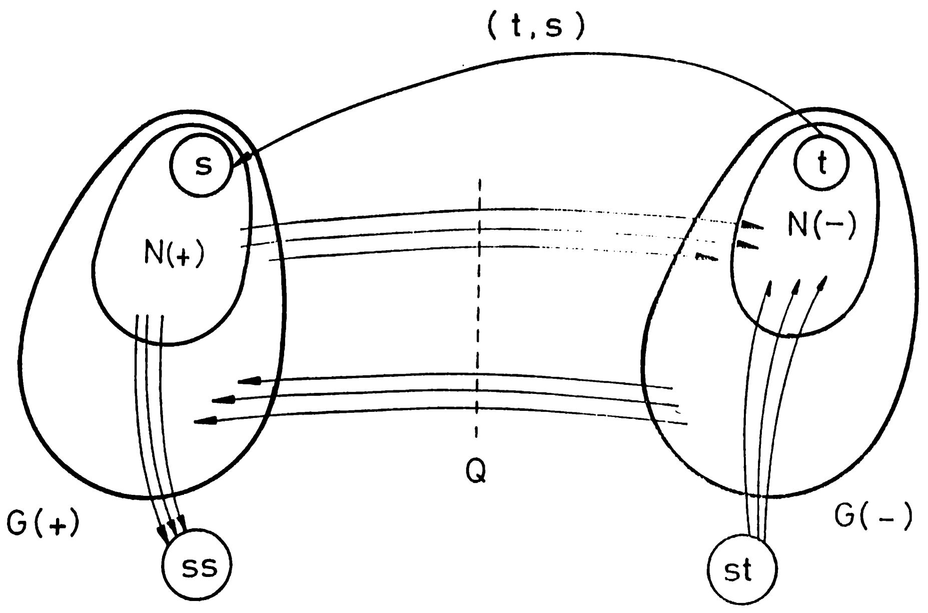

Figure III.2 The subraphs of $G''$.

We must somehow manipulate Preflow and Postflow on edges of G(+) and G(-) such that excesses are reduced to zero. In G(+), source node sends flow to other nodes, hence all positive excess nodes are connected to source by paths that carry Preflow. These paths correspond to residual paths going from nodes of N(+) to source. Balancing a positive node means to push its excess toward source along the residual path.

Similarly, sink absorbs Postflow coming from nodes of N(-). Negative excess nodes are connected to sink by means of paths which carry Postflow. These paths correspond to residual paths which are directed from sink to nodes of N(-). In order to send negative excess of any node to sink, we either reduce flow on paths that carry Postflow or send flow through residual paths that reach that node. We will prefer to use residual paths for the stability of the algorithm and make use of search tree that is on hand at the end of the first stage.

Goldberg and Tarjan's Reduce/Relabel step [13] could be modified and extended to take care of both negative and positive nodes and applied to the problem as well. This alternative procedure will be discussed in the next chapter.

In the second stage, instead of reducing excesses of $G_r(+)$ and $G_r(-)$ graphs independently, we connect them by residual of ( t,s ). The procedure of the first stage is applied to the new residual graph. Positive excesses are pushed toward negative excess nodes along residual paths which pass thru ( s,t ) edge.

The details about the algorithm, its stages, the procedure used to carry out balancing operation and related results will be given in the next chapter.

CHAPTER IV

# THE MAXIMUM FLOW ALGORITHM

The algorithm determines the max-flow of a network in two stages. The task of the first stage is to get a min-cut set and that of the second stage is to obtain the corresponding max-flow. The point that distinguishes it from many other well-known algorithms is that instead of finding a max-flow first and then defining a min-cut set , algorithm first defines min-cut set and then finds max-flow. In this respect, it is similar to Goldberg-Tarjan [13]. The algorithm approaches an optimum solution from dual side of the model. The first stage provides dual optimum solution of the problem. When the first stage ends, some nodes may remain unbalanced and consequently the problem may still be primal infeasible. During the progress of the second stage, the algorithm approaches a primal optimum solution of the dual optimum. In this chapter, the first two stages of the algorithm will be described. Later on, the convergence of the algorithm toward the optimum will be shown to take a polynomial number of steps, and a sample problem will be given.

## i. THE FIRST STAGE

Initially edge flows are set to be equal to upper capacities and excesses of nodes are determined. Two dummy nodes, namely, supersink and supersource are created and added to node set N . Positive excess nodes are connected by forward artificial edges to supersource each having a capacity that is equal to the excess of the node it is coming out of. In a similar way, artificial edges directed to negative excess nodes are used to connect supersink to these nodes. Since source has a positive excess , it is also connected to supersource. Meanwhile sink with a negative excess is adjacent to an artificial edge coming from supersink. New graph $G''$ with dummy nodes and artificial edges is constructed. Next step is to find the initial distance labels of nodes. Distance label of supersource is zero and it remains as zero throughout the progress of the algorithm. Starting from supersource, initial distance labels of nodes are determined by breadth-first search on $G''_r$ which takes o(m) time. The initial search tree is constructed by using these distance labels of nodes. Therefore the resulting tree is a shortest path tree of the residual graph.

In the first stage, source and sink nodes are treated as distinct nodes and they are connected to dummy nodes depending on their excesses. The algorithm basically searches for shortest residual paths, starting from supersource and ending up at supersink by making use of the distance label function. These residual paths can be called flow-augmenting paths, since once such a path is found, as much flow is sent through it as possible. From this aspect, the algorithm can be considered to be a flow-augmenting algorithm.

The distance label of supersink *k* gives the lengths of the flow-augmenting paths. Similarities between Edmonds-Karp and this algorithm, and, between Dinic's and this algorithm can be seen at the first sight. Like Edmonds-Karp algorithm, it searches shortest paths by breadth-first search and like Dinic's algorithm the search continues until all paths of length *k* are found. When there remains no path of length *k*, meaning that it is required to increase the distance label of supersink, its distance label is updated and it becomes *l*. The algorithm guarantees that *l* is strictly greater than *k*. Then the algorithm proceeds by searching new augmenting paths of length *l*. At this point, phase concept can be brought into the picture. A phase finds flow-augmenting paths of a given length, which is the distance label of supersink,and sending flow from ss to st along them. The main advantage of using the phase concept is to differentiate lengths of augmenting paths and search them in ascending order of their lengths and therefore give a concrete structure to path-searching process.

In any phase, the algorithm proceeds by maintaining a search tree rooted at supersource and pushing flow along residual paths of length k, connecting ss to st. Whenever a path is determined, flow is sent thru. The amount of flow, denoted by $\Delta$ , is the minimum of residual capacities of edges on the path. Flow is sent and residual capacites of edges are updated. $\Delta$ is either residual capacity of any one of artificial edges or residual of an original edge of the graph. Depending on whether residual edge is artificial or not, two cases occur;

I. Saturation of the artificial residual edges.

There are two artificial edges on the path. If the one which is adjacent to supersource is saturated, then the corresponding positive excess node is balanced . If the other one, adjacent to supersink, is saturated, then a negative excess node is balanced. Once any artificial residual edge is saturated and deleted from the graph, it is never added to subsequent search tree. Push is said to be excess zeroing.

II. Saturation of any edge.

If $\Delta$ equals minimum residual capacity of an original edge on the path, then push is said to be excess non-zeroing. In other words, a unique path of the tree is blocked. No more flow could be sent through it. Since the residual capacity of an edge is used up, a part of the tree which consists of that edge and its successors are disconnected from the search tree. They are temporarily deleted from search tree.

In the case of ties, the one which is closest to supersource is processed first.

Tree is updated to reflect changes in the tree structure. Let ( i,j ) be the edge that is saturated and deleted from the tree. Edges going into node j are examined. If an edge ( k,j ) satisfying conditions of $d(j) = d(k) + 1$ and $r(k,j) > 0$ is found, no relabeling takes place. It is called a replacement. New edge ( k,j ) is added to the tree. If not, node j is relabeled and its distance label becomes,

$$d(j) = \min \{ d(i) + 1 \mid r(i,j) > 0 \text{ and } i \in T \}$$

Deletion of edge (i,j) disconnects a sub-tree from the search tree. Node j becomes the root of that sub-tree. If the sub-tree has more than one node, it indicates that distance labels of nodes of sub-tree have been determined according to d(j). Hence after updating the distance label of node j, distance labels of nodes of sub-tree should be revised. Nodes are placed in a first come last serve stack according to decreasing order of their distance labels so that node with smallest distance label is at the top. Until stack becomes empty, following procedure is repeated. Node, say z, is taken from top of the stack and its distance label is updated according to

$$d(z) = \min \{ d(i) + 1 \mid r(i,z) > 0 \}.$$

The following Lemma shows that relabeling does not violate validity of distance labels of nodes.

**Lemma IV.1:** In every phase a valid search tree of shortest paths is constructed.

**Proof:** The initial tree is a valid search tree. During execution, the tree is grown, new augmenting paths are found and excess zeroing or non-zeroing pushes are made. The tree grows while keeping validity conditions of the search tree satisfied. Pushes cause structural changes, since residual capacity of some edges are reduced to zero and deleted from the tree. These edges and successors are taken and for such an edge $(i,j)$, $d(i)$ is updated and by using new distance labels, the search tree is expanded. Validity is preserved. ■

When all edges going out of nodes of the tree are fully saturated and no more nodes can be reached from the nodes of the tree, the search tree can not be expanded any further. The tree spans positive excess nodes including source and some of balanced nodes which are reached from them. The remaining nodes, including supersink and sink belong to a second set. Hence all the nodes of $G''$ form two disjoint node sets. Then a cut naturally appears and first stage terminates. The following Lemma will show the existence of a cut at the end of the first stage.

**Lemma IV.2:** When first stage terminates, $\exists$ at least one edge $(i,ss) \in E''$ st. $g(i,ss) > 0$ (i.e $r(ss,i) > 0$) and $S$ is a cut consisting of nodes of search tree $T$ and $\bar{S} = N'' \setminus S$. $\forall\ (h,l) \in (S \times \bar{S}) \cap E''$, $g(h,l) = u(h,l)$.

**Proof:** First stage terminates, when $\exists$ no path going from supersource to supersink on $G''_r$. Let $S = \{j \mid j \in N'' \setminus \{ss\},\ j \in T\}$ $\forall\ j \in S$, there is no node $k$, s.t. $d(k) = d(j) + 1$ and $r(j,k) > 0$. Since $e(s) < 0$, source is also in the set.

Therefore S defines a cut in residual graph and it is also cut in $G''$.

In the next step, we must prove that residual edges corresponding to the edges of original graph connecting nodes of search tree to the rest of nodes are saturated. $\forall( h,l ) \in ( S \times \bar{S} ) \cap E''$, assume $r_{hl}>0$ and $d(l) = d(h) + 1$, then $l \in S$. But it has been assumed that $l \in \bar{S}$, contradiction. Therefore, $r_{hl} =0$ and $l \in \bar{S}$. ■

Later we will show that this cut is the minimum cut. In order to do that it must be proved that flow across $(S,\bar{S})$ is not disturbed during second stage. It will be done in the second stage.

Let us now show the polynomiality of the proposed algorithm.

**Lemma IV.3:** Each node is relabeled at most ( n+1 ) times and upperbound to number of relabelings is $( n+1 )^2$.

**Proof:** When a node is cut off from the tree and relabeled, its distance label increases at least by one unit. A valid label must satisfy the condition; $0 < d(i) < (n+1)\ \forall\ i \in N''$. There are ( n+2 ) nodes and each node is relabeled (n+1) times. Total number of relabelings is $( n+1 )^2$. ■

The following Lemma is taken from Goldberg and Tarjan [13].

**Lemma IV.4:** All of excess zeroing and non-zeroing pushes are saturating pushes and number of saturating pushes is at most $(n^o m^o)$, where $n^o= (n+2)$ and $m^o= (m+n+1)$.

Proof: New graph $G''$ consists of n original and two dummy nodes and m original and (n+1) artificial edges. In each push at least one edge is saturated.

Now consider any saturated edge ( i,j ) s.t.( i,j ) $\in E''$. Again to push flow from i to j requires first pushing flow from j to i, which can not happen until d(j) increases by at least two. Similarly, d(i) must increase by at least two between saturating pushes from j to i. Since $d(i)+d(j) \geq 3$ when first push between i and j occurs and $d(i)+d(j) \leq 2n^o-1$ when the last such push occurs, total number of saturating pushes between i and j is at most ( $n^o-1$ ). Thus the total number of saturating pushes is at most Max { 1, $n^o-1$ } per edge for a total of max { 1, ( $n^o-1$ ) } $m^o \leq n^o m^o$. ■

Let E(i) be the edge list of i consisting of all edges that are adjacent to i. Every edge (i,j) of graph is both in E(i) and in E(j).

Result IV.1: Algorithm runs $o((n^o)^2 m^o)$ times in the first stage.

Proof: By Lemma IV.4 , the number of saturating pushes is at most ( $n^o m^o$) and time spent in pushing steps is $((n^o)^2 m^o)$. Each push will saturate at least one edge. Hence number of pushing steps will give a bound on the number of distance label updating steps or in other words, number of times search tree is updated. o( $n^o m^o$) saturating pushes will result updating of tree o( $n^o m^o$) times.

When distance label of node is updated, it either does not change or increases at least by one. In the first case, replacement operation is carried out and edge ( k,i ) of node i is added to tree, s.t. $d(i) = d(k) + 1$ $r(k,i) > 0$ and $k \in T$. In each layer, edge list of i is scanned once. Therefore, total number of replacements is $o(\sum_i (n+1)*E(i)) = o( 2(n+1)m^O ) = o( n^O m^O )$. In the case of relabeling, each node i is relabeled at most ( n+1 ) times and each relabeling requires single scanning of E(i). If we sum over all nodes of graph, then total time spent in relabeling will be $o ( \sum_i (n+1)*E(i) ) = o( 2(n+1)m^O )= o( n^O m^O)$.
Total of time requirements is $o((n^O)^2 m^O)$. ■

## ii. THE SECOND STAGE

In the beginning of the second stage, there are two distinct sets of nodes. S consists of both positive and some of the balanced nodes. Negative excess and remaining balanced nodes are elements of the second set $\bar{S}$. The second stage balances remaining unbalanced nodes.

The second stage determines residual paths connecting supersource to source and balances positive nodes by returning their excesses along these paths. The algorithm balances negative nodes by determining residual paths connecting sink to supersink and pushing negative excesses toward supersink along residual paths. These two tasks could be handled together by introducing an artificial edge

( t,s ). Before going into that, it is worth to say that if nodes of set S would be considered only, problem could be simplified to the problem of returning positive excesses to source and can be solved by the procedure Reduced/Relabel which has been developed by Goldberg and Tarjan. In simple terms, this procedure starting from a positive excess node reduces Preflow on edges of original graph coming into that node until the node is completely balanced. Positive excesses are pushed back toward source on the residual graph. In their algorithm, there are only positive excess and unbalanced nodes. The existence of negative nodes makes the situation more complex. They have to be handled somehow. As mentioned before, residual paths going from sink to supersink have to be determined and as much flow as possible has to be sent to st in order to balance them.

During the second stage, the algorithm works on a new graph $G'$. Node set remains as it is but it is assumed that edges of ( $S,\bar{S}$ ) cut are deleted and an artificial edge ( t,s ) is added to edge set. New residual graph $G_r'$ is constructed. The reason for the deletion of ( $S,\bar{S}$) edges is to handle the problem created by double reachable nodes. When residual edge ( s,t ) is added to the graph, some nodes of S can be reached from nodes of $\bar{S}$ by means of the shorter residual paths. For the sake of simplicity and to avoid occurence of such situations edges of ( $S,\bar{S}$ ) cut are deleted from original graph.

The search tree of the first stage is conserved. new distance labels of nodes of $\bar{S}$ are determined on $G_r'$ by breadth-first search and the tree is enlarged only from source node. Due to symmetry, negative excess nodes are reached from sink on $G_r'$. From this point, the second stage is carried out like first stage. Complete residual paths which start from ss and end up at st are found one by one and then as much flow as possible is sent along them.

The second stage terminates when nodes are balanced. Following results first prove that second stage balances unbalanced nodes of the first stage and the cut obtained after completion of the first stage is a minimum cut.

**Lemma IV.5:** If there is a node i, s.t. $i \in T$ and $e(i) > 0$, then there exist a node j s.t. $e(j) < 0$ and there is a residual path which is either on $G_r''$ or on $G_r'$ joining i to j.

**Proof:** Let A be incidence (node-edge) matrix and f flow vector, then e excess vector is;

$$e = A * f$$

Let $e^o = ( 1,1,1,1,... )$, then;

$$e^0 * e = \sum_i e(i) = e^0 * A * f = 0 * f = 0$$

If there is a node i, s.t. $e(i) > 0$, there must be at least one node j, s.t. $e(j) < 0$, so the sum of excesses could be zero.

Now let us prove second part of lemma.

If $i \in T$ and $e(i) > 0$, there are two cases that can occur. Either a path connecting i to a node j s.t. $e(j) < 0$ is

found or not. If a path is found, then two nodes with opposite signs are connected. Otherwise, tree can not be grown from node i or from the subtree which is rooted at node i, then i remains unbalanced during first stage. When second stage starts, $s \in S$, $e(s) > 0$ and $\mathrm{Outflow}(s) > 0$. On the other hand $\forall$ i s.t. $i \in S$ and $e(i) > 0$, $\mathrm{Inflow}(i) > 0$. Therefore, there should exist paths which carry positive flow from source to positive excess nodes and consequently due to flow-symmetry, there exist residual paths going from positive excess nodes to source. Same logic is applicable to the nodes of negative excess. Hence there are residual paths going from sink to negative excess nodes. On $G_r'$, source and sink are joined by an artificial edge ( t,s ) and it results a residual edge ( s,t ). $\forall$ i s.t. $i \in T$ and $e(i) > 0$, there exist a residual path going from i to s, s to t and t to j, s.t. $e(j) < 0$.
As a result, there always exist a residual path either on $G_r''$ or on $G_r'$ , connecting a positive node to a negative node.■

Lemma IV.6: In the second stage, each residual path connecting supersource to supersink passes through source-sink nodes.
Proof: In the second stage, tree can grow only from source-sink since at least one outgoing edge of source and incoming edge of sink have positive flow. If $\exists$ node i, s.t.$e(i) > 0$, at the beginning of stage 2 due to previous Lemma IV.5, $\exists$ at least one node j st $e(j) < 0$ and a residual path connecting them. Therefore each residual path must go

through source-sink nodes. ■

**Result IV.2:** During the second stage, cut ( $S$ , $\bar{S}$ ) is conserved and it gives a minimum-cut. When the second stage terminates, a maximum flow is found.

**Proof:** Due to structure of $G'_r$, Lemma IV.5 and min-cut max-flow theorem of Ford-Fulkerson [9].

The activities carried out during both of the stages are the same. The only difference between them is the graph on which they are working. Hence , results and assigned time bounds to the activities of first stage are also analogous to those of the second stage. We restate them without proof.

**Lemma IV.7:** The number of excess zeroing and non-zeroing pushes is at most ( $n^O m^O$), where $n^O$= (n+2) and $m^O$= (m+n+1).

**Lemma IV.8:** Upperbound to number of relabelings is $(n+1)^2$ .

**Result IV.3:** Algorithm runs $o((n^O)^2 m^O)$ times in the second stage.

**Result IV.4:** Overall complexity of the algoritm is again $o((n^O)^2 m^O)$.

**Proof:** Due to Result IV.1 and IV.3. ■

The order of the proposed algorithm matches with the order of Dinic's algorithm.

The bottleneck operation of the algorithm is excess zeroing and non-zeroing pushes. They increase the order to

$((n^o)^2m^o)$. If they are somehow handled more carefully , running time of the algorithm can be improved.

## iii. A FORMAL VERSION OF THE ALGORITHM

After describing the stages of the algorithm in detail and discussing the time bounds of the activities carried out during the stages, in this section, a brief version of the algorithm will be given.

All the steps followed by the algorithm are stated one by one in the preceding order below.

The First Stage

a. Construct $G''$.

b. $g(i,j) = u(i,j) \; \forall (i,j) \in G$.

c. Determine node excesses.

d. Assign values to upper capacities of the artificial edges used to connect ss and st to the G.

e. Determine the upper capacity of (t,s).

f. Let quasi-flow on each artificial edge be its upper capacity.

g. Determine initial distance labels of nodes by applying breadth-first search on the $G''_r$.

h. Construct the initial search tree.

i. Start to find residual paths which connect ss to st and then send as much flow as possible.

j. Continue until, no ss-to-st residual path remains.

A $(S,\bar{S})$ cut appears.

The Second Stage

a. Construct $G'$ by deleting edges of $(S,\bar{S})$ cut and reconsidering the existence of (t,s).

b. Do again breadth-first search on $G'_r$ to obtain distance labels of nodes of $\bar{S}$.

c. Update search tree.

d. Start to determine residual paths which are directed from ss to st and then send as much flow as possible.

e. Continue until all the excesses of nodes are zeroed.

## iv. AN APPLICATION OF THE PROPOSED ALGORITHM

In this section, the maximum flow algorithm presented in the previous sections will be applied to a simple network given in Figure IV.iv.1.

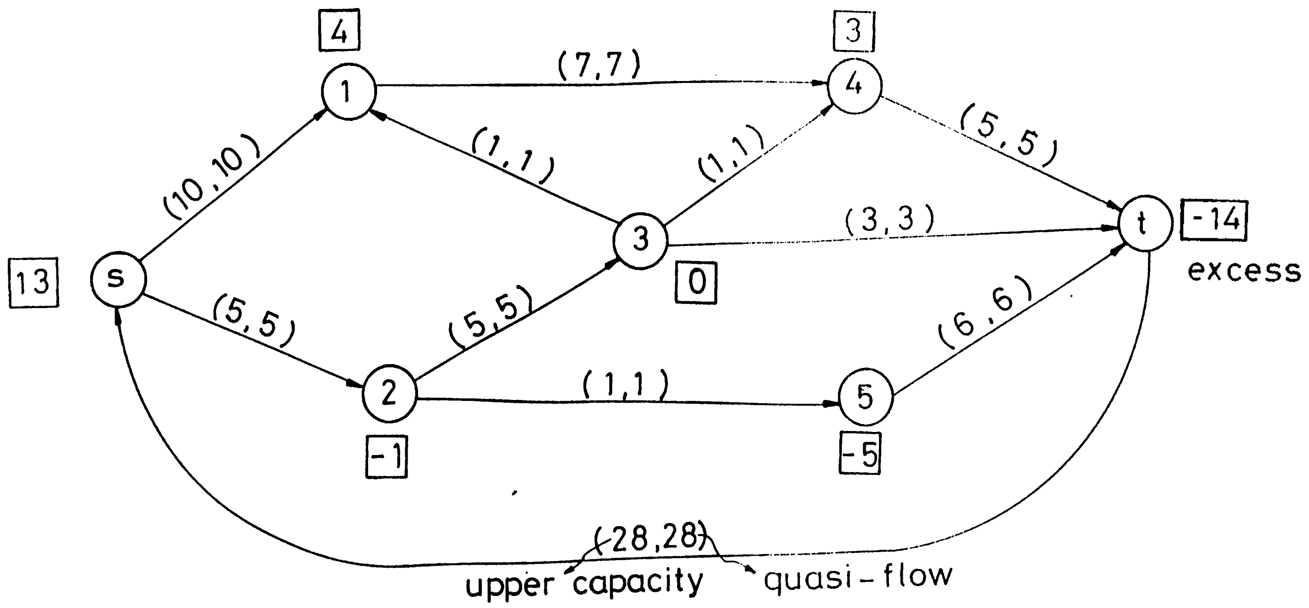

Figure IV.iv.1. A sample network G.

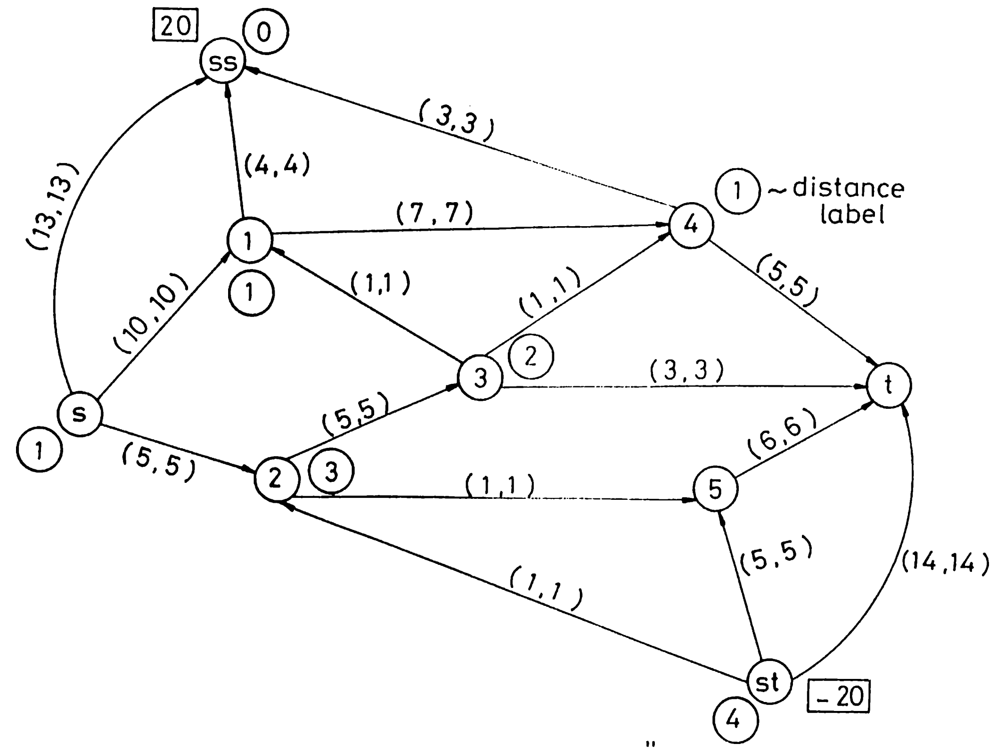

Figure IV.iv.2. Graph $G''$.

First $G''$ is constructed. See Figure IV.iv.2. b,c,d,e,f, and g steps of the first stage which are given in the previous section are carried out. According to the initial distance labels of nodes, initial search tree shown in Figure IV.iv.3 is constructed. Then whenever a ss-to-st

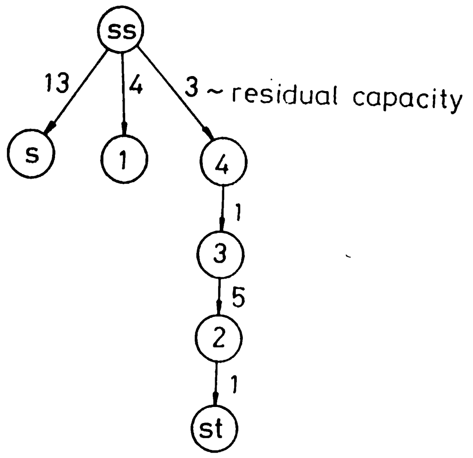

Figure IV.iv.3. Initial search tree.

residual path is found, as much flow as possible is sent through it. The first stage terminates, when no such path remains. The search tree which is on hand at the end of the first stage is given in **Figure IV.iv.4.** Nodes belonging to

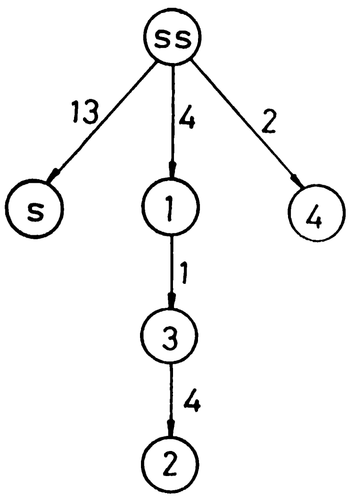

**Figure IV.iv.4.** Search tree at the end of first stage.

the tree are on the set S and remaining nodes are in $\bar{S}$. $(S,\bar{S})$ cut appears as illustrated in **Figure IV.iv.5.**

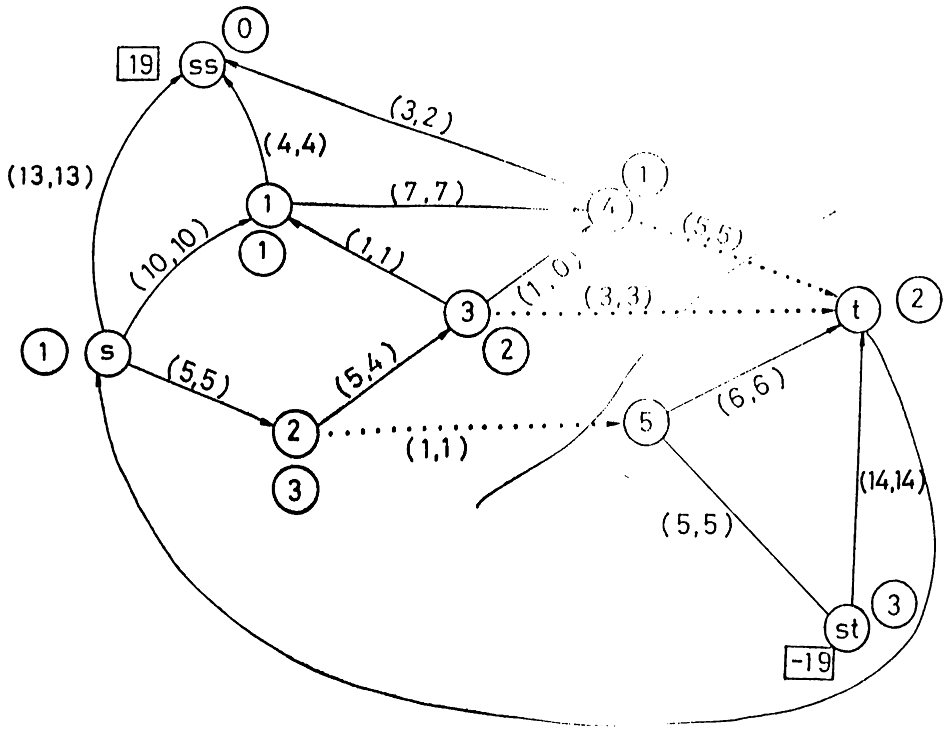

Figure IV.iv.5. $(S,\bar{S})$ cut on $G'$.

The second stage is started to balance remaining unbalanced nodes. Graph G' given in Figure IV.iv.5 is obtained. When b and c steps of the second stage are done, the search tree is obtained as shown in Figure IV.iv.6. Path finding and flow augmentating are repeadetly carried out until all the excesses are zeroed, and the maximum flow is found. Final flows on the edges of the graph are given in Figure IV.iv.7.

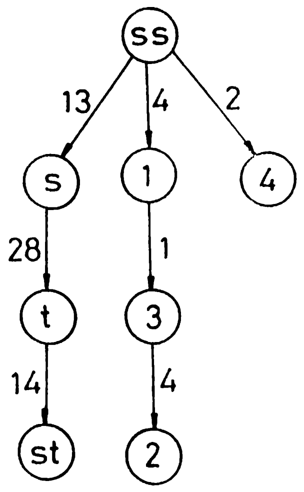

Figure IV.iv.6. Search tree at the beginning of second stage

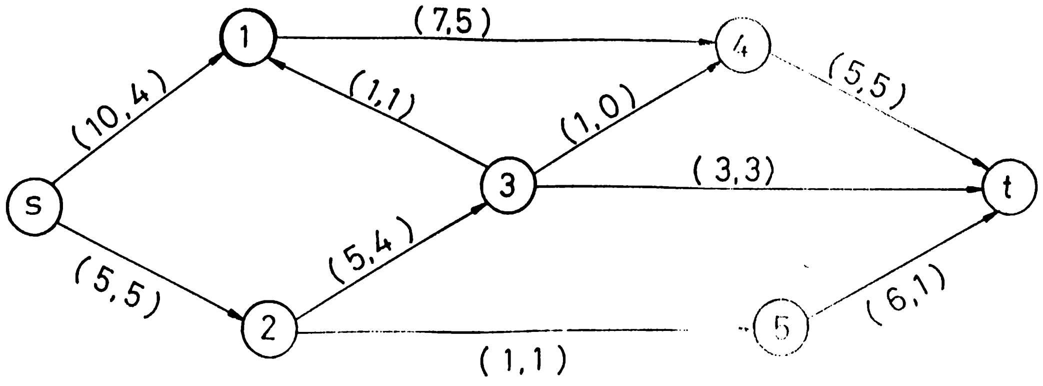

Figure IV.iv.7. Final flows on the edges of G.

## V. AN ALTERNATIVE PROCEDURE FOR THE SECOND STAGE

The second stage of the algorithm can be carried out by an extended version of Reduce/Relabel step of Goldberg and Tarjan's algorithm [13]. In their case, given positive excess nodes and Preflow on edges, along shortest paths of positive flow which go from source to unbalanced nodes, flow is reduced to make the positive excesses zero. In our case, there exist both positive and negative excess nodes and by means of paths of positive flow source is connected to positive nodes and sink can be reached from negative excess nodes. As mentioned before, Preflow is defined in the subgraph which contains positive excess nodes, source and some of the balanced nodes. Reduce/Relabel step is directly applicable to that subgraph. Excess flow is returned to source along shortest paths. In order to apply Reduce/Relabel step to send flow excesses of negative nodes to sink, a few modifications have to be made. Subgraph which will be worked on has to be redefined so that for a given Postflow p, $E_p = \{ (i,j) \mid p(i,j) > 0 \}$ and $G_p = ( N, E_p )$. By breadth-first search, starting from sink, initial distance labels of nodes are determined s.t. $d(t)=0$ and $0 < d(i) < n$, $\forall\ i \neq t$. Then Reduce/Relabel step becomes:

Reduce: Select any node $j \neq s$ with $0 < d(i) < n$ and $e(j)<0$. Select any edge $(j,i) \in E_p$ with $d(i) = d(j) - 1$. Send $\delta = \min \{ |e(j)| , p(j,i) \}$ units of flow from j to i. If $\delta$ is equal to $|e(j)|$, step is excess zeroing otherwise excess non-zeroing.

Relabel: Select any node j with $0 < d(i) < n$ . Replace $d(j) = \min \{ d(i) + 1 , | (j,i) \in E_p \}$.

They have shown that time requirement of that step is always smaller than $o( n^2m)$ which dominates the time complexity of their algorithm. When they have used the Dynamic Trees , they have reduced the time complexity to $o( nm\log(n^2/m) )$. Even in that case, time requirement of that step is smaller than above bound. Hence it is sure that running time of our Reduce/Relabel stage will be smaller than $o( nm\log n )$ which is the smallest bound we obtained in this study. Therefore this does not improve the overall running time of our algorithm.

CHAPTER V

# DYNAMIC TREE VERSION OF THE ALGORITHM

The running time of the proposed maximum flow algorithm is $o(n^2m)$. Activities of distance label updating require o(nm) time. Time spent in excess zeroing and non-zeroing pushes is $o(n^2m)$ and it dominates overall time complexity of our algorithm. If time required to push flow along residual paths from supersource to supersink is reduced from $o(n^2m)$ to o(nmlogn), overall complexity of the algorithm will be reduced. Therefore, our further effort will be concentrated on that. We adopt and apply Sleator-Tarjan's dynamic tree data structure to handle pushing steps more efficiently and obtain a bound of o(nmlogn).

In dynamic tree structure, tree operations are defined and used either to get information from the tree or structurally update the tree. Tree operations that will be used in this version are listed below.

| | |
|---|---|
| parent(i) | returns the parent node of i. If i is the root, null is returned. |
| root(i) | return root of the tree containing node i. |
| cap(i) | return the residual capacity of the edge (parent(i),i). |
| mincap(i) | return the node j whose edge (parent(j),j) has minimum residual capacity among edges of path which goes from root to node i. |
| update(i,x) | update residual capacities of edges on the tree path from root to node i by adding x amount to each of them. |
| link(i,j,x) | combine trees containing node i and j by adding new edge (i,j) with residual capacity x, making i be the parent of j. |
| cut(i) | divide the tree containing node i |

into two trees by deleting the edge (parent(i),i), return residual capacity of it.

Among them, root(i), mincap(i), cap(i) are used to get information from tree, others change the tree structure. Cut(i), link(i,j,x), cap(i) operations are carried out at o(1) time per operation. Update(i,x), mincap(i), root(i) require o(n) time per operation in the worst case. Dynamic tree structure handles each operation in o(log n) time, if tree contains at most n nodes at any time.

The algorithm maintains a collection of disjoint trees. Each node has a single parent node and several children nodes. We assign a pointer variable point(i) to each node i. If point(i) is 1, node is a candidate node to enlarge tree, otherwise it is not. Initially supersource is the only candidate node and cap(i) of each node is equal to infinity indicating that each node is a tree of single node.

The dynamic tree version of the algorithm is as follows.

Step 1. If there is a candidate node go to Step 2, otherwise compute residual capacity of each edge and STOP.

Step 2. Take a candidate node i, s.t. point(i)=1. If there is an edge (i,j) in edge list of i s.t. d(j)=d(i)+1 and r(i,j) > 0, go to Step 3. Otherwise point(i)=0 and go to Step 1.

Step 3. If j = st, go to Step 4, otherwise go to Step 5.

Step 4. Enlarge tree from node i, by making i the parent of j and performing link(i,j,r(i,j)). Node j becomes another candidate node, point(j)=1. If parent(j) still is a candidate node, go to Step 2.

Step 5. A path from supersource to supersink is found, then a flow is sent thru. Let k=mincap(st), then $\Delta$ =cap(k) and perform update(st, $-\Delta$ ). Go to Step 6.

Step 6. Delete edges with zero residual capacities. Repeat following step until cap(k) > 0 where k=mincap(st).

Let k=mincap(st), perform cut(k) and update distance label of node k. If node k has to be relabeled, then apply cut(i) and update(i, $\infty$ ) operation to each node of sub-tree that is disconnected from the search tree. After relabeling of node k, update their distance labels.

Then, go to Step 1.

Lemma V.1: Time complexity of the dynamic tree version of the algorithm is $o(n^o m^o \log n^o)$ where $n^o=(n+2)$ and $m^o=(m+n+1)$.

Proof: Since each push is a saturating type and reduces residual capacity of at least one edge to zero, number of pushing steps is equal to number of cut operations. Time required by cut operations is $o(n^o m^o)$.

Total number of link operations is $(n^o+n^o m^o)$. Each node

initially can be linked to the tree once. In addition to that, every node which was disconnected from the tree can later be linked again. This reasoning gives the above bound. Total number of link operations gives an upper bound to the number of times nodes become candidate. It is $o(n^o+n^om^o)$. Total relabeling time is $o(n^om^o)$ and total time spent in replacement steps is again $o(n^om^o)$.
The number of excess zeroing and non-zeroing pushes gives number of paths from ss to st. To assign a bound to operations of Step 5, we use that result. Hence update operations require $o(n^om^o\log n^o)$ time.
Total complexity of the algorithm is $o(n^om^o\log n^o)$. ■

We obtained the order of Sleator and Tarjan's algorithm [21]. Modified Dynamic Tree version of the algorithm is applied twice on the residual graph. The overall running time of the proposed maximum flow algorithm is o(nmlogn).

# CHAPTER VI
# CONCLUSIONS AND FUTURE RESEARCH

In this study, we tried to approach the classical maximum flow problem from a different point. The idea behind the algorithm was originated from the following intuitive fact. If the flow on each edge of the graph is equated to upper capacity on that edge , in order to obtain a feasible and optimal flow, flows on some of the edges have to be reduced. Our algorithm answers the question by how much flows on the edges have to be reduced.

Algorithm defines min-cut first, then finds max-flow. When the first stage terminates, dual optimum solution will be on hand, but, primal optimum solution will not. The task of the second stage is to find primal optimum solution of a given dual optimum. It is a general question, whether there is a relation between the procedure of the second stage and Dual Simplex method.

We wonder whether two stages of the algorithm can be combined and handled together. In that case, it will be difficult to define min-cut before obtaining max-flow.

The running time of the proposed algorithm is o( $n^2m$ ). We have used a modified version of Dynamic Tree data structure of Sleator and Tarjan, and time complexity of the algorithm is reduced to o( nmlogn ). It is applied two times on two graphs which have equal size in terms of n and m. Therefore overall complexity of maximum flow algorithm is o( nmlogn ).

For the second stage of the algorithm we present an alternative procedure to obtain maximum flow of a given min-cut. It employes Reduce/Relabel step of Goldberg and Tarjan's algorithm to send positive excesses of nodes to source. With a little modification, Reduce/Relabel step is applied once more to reduce negative excesses of nodes by pushing the toward sink. The procedure does not improve the time complexity of the algorithm. We believe that although theoretical bound of alternative procedure is smaller than original procedure, our algorithm will possibly beat the alternative in terms of practical applicability.

The maximum flow problem defined in Chapter 1 can be generalized by assigning positive lower capacity bounds on the edges. We believe that with a few modifications in determining the residual capacities of the edges, amount of flow which can be sent along a residual path, and capacity of a cut, our algorithm can handle the generalized problem. However more work required to show how the algorithm recognizes the infeasibility of the primal problem. It is potentially a future research topic.

We believe that time complexity of our algorithm can be reduced to o(nm) by a different way of detection of min-cut set and avoiding usage of dynamic structure.

## R E F E R E N C E S

[1] R.K. Ahuja and J.B. Orlin, "New distance-directed algorithms for maximum flow and parametric maximum flow problems," Working Paper, Sloan School of Management, M.I.T., Cambridge, MA 1987.

[2] R.K. Ahuja and J.B. Orlin, "A fast and simple algorithm for the maximum flow problem," Working Paper, Sloan School of Management, M.I.T., Cambridge, MA 1987.

[3] R.V. Cherkasky, "Algorithm of construction of maximal flow in networks with complexity of $o(n^2m^{1/2})$ operations," *Mathematical Methods of Solution of Economic Problems* 7(1977) 117-125 (in Russian).

[4] E.A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Doklady* 11(1970) 1277-1280.

[5] G.A. Dirac, "Short proof of Menger's Theorem," *Mathematiks* 13(1966) 42-44.

[6] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM* 19(1972) 248-264.

[7] D. Elias, A. Feinstein and C.E. Shannon, "Note on maximum flow through a network," *IRE Trans. on Information Theory* IT-2(1956) 117-119.

[8] L.R.Jr. Ford and D.R. Fulkerson, "Maximal flow through a network," *Canadian J. of Mathematics* 9(1956) 399-404.

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

[10] H.M. Gabow, "Scaling algorithms for network problems," *Proc. 10th ACM Symp. on Theory of Computing* (1983) 248-258.

[11] Z. Galil, "An $o(n^{5/3}m^{2/3})$ algorithm for the maximal flow problem," *Acta Informatica* 14(1980) 221-242.

[12] Z. Galil and A. Naamad, "An $o(mn\log^2 n)$ algorithm for the maximal flow problem," *J. Comput. System Sci.* 21(1980) 203-217.

[13] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," *18th Annual ACM Symp. on Theory of Computing* (1986) 136-146.

[14] D. Goldfarb and J. Hao, "A primal simplex algorithm that solves the maximum flow problem in at most num pivots and $o(n^2m)$ time," Technical Report, Dept. of Industrial Engineering and Operations Research, Columbia University, New York, NY, 1988.

[15] T.C. Hu, *Integer programming and Network Flows*, Addison-Wesley, Mass., 1969.

[16] A.V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Math. Doklady* 15(1974) 434-437.

[17] E.L.Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, N.Y.,1976.

[18] V.M. Malhotra, M. Pramodh Kumar and S.N. Maheshwari, "An $o(|n|^3)$ algorithm for finding maximum flows in networks," *Inform. Process. Lett.* 7(1978) 277-278.

[19] R.T. Rockafellar, *Network Flows and Monotropic*

*Optimization*, John Wiley & Sons, Canada, 1984.

[20] Y. Shiloach and V. Vishkin, "An o(logn) parallel connectivity algorithm," *J. Algorithms* 3(1982) 57-67.

[21] D.D. Sleator and R.E. Tarjan, "A data structure for dynamic trees," *J. Computer and System Sciences* 26(1983) 362-391.

[22] R.E. Tarjan, "A simple version of Karzanov's blocking flow algorithm," *Operations Research Letters* 2(1984) 265-268.