MIT/LCS/TR-230

THE COMPLEXITY OF THE MAXIMUM NETWORK FLOW PROBLEM

Alan Edward Baratz

THE COMPLEXITY OF THE MAXIMUM NETWORK FLOW PROBLEM

Alan Edward Baratz

March 1980

This report was prepared with the support of the National Science Foundation research grant MCS78-05849.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

THE COMPLEXITY OF THE MAXIMUM NETWORK FLOW PROBLEM

by

Alan Edward Baratz

Submitted to the Department of Electrical Engineering
and Computer Science on May 21, 1979 in partial fulfillment
of the requirements for the Degree of

Master of Science

Abstract: This thesis deals with the computational complexity of the maximum network flow problem. We first introduce the basic concepts and fundamental theorems upon which the study of "max-flow" has been built. We then trace the development of max-flow algorithms from the original "labeling algorithm" of Ford and Fulkerson, through a recent $O(V \cdot E \cdot \log^2 V)$ algorithm due to Galil and Naamad. We include a description of each of these algorithms, along with a proof of correctness and proof of running time for most of them. Finally we turn our attention to the problem of establishing lower bounds on the complexity of max-flow. We show that a straightforward application of the polyhedral lower bound technique developed by Yao, Avis and Rivest fails to produce a non-linear lower bound on max-flow. In the process, however, we prove several interesting results concerning the facial structure of a class of polyhedra very closely related to the maximum network flow problem.

Key words: computational complexity, geometric complexity, network flow, polyhedral decision problem

Thesis Supervisor: Ronald L. Rivest

Title: Associate Professor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

I am deeply indebted to my thesis supervisor, Professor Ronald Rivest, for first introducing me to the maximum network flow problem and for his continued support and guidance. I believe that my association with him has been the single most important factor in my graduate education.

I would also like to thank Jeffrey Jaffe and Michael Loui for numerous enlightening discussions. I am especially grateful for their constructive criticism of my work.

Finally, I would very much like to thank my mother Adele, my uncle Charles, and my fiancee Raquel for their constant love and support. They always seem to be available when I need them. I am especially grateful to Raquel for her faith and understanding through these past few months. This thesis is, in spirit, as much hers as it is mine.

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

1.1 An Overview of the Thesis

The problem of determining a maximum steady state flow from one point to another in a network with edge capacity constraints, has come to be known as the maximum network flow problem ("max-flow" in short). L.R. Ford and D.R. Fulkerson [9] were the first to study max-flow as a computational problem. They developed the first max-flow algorithm in the mid 1950's and laid the groundwork for much of the research that was to follow. Since that time max-flow has been widely studied and has developed great practical application, especially in the analysis of transportation and communication networks. This thesis will be concerned with the computational complexity of the maximum network flow problem, investigating both upper and lower bounds on the problem.

The remainder of this chapter will be devoted to a development of the foundation necessary for any coherent study of the maximum network flow problem. We will begin by presenting the basic definitions and concepts that have become standard in the max-flow literature. We shall then introduce the fundamental theorems upon which the study of max-flow has been built. These theorems, due originally to Ford and Fulkerson [9], are known as the Augmenting Path Theorem, the Integral Flow Theorem and the Max-Flow Min-Cut Theorem.

The second chapter in this thesis will deal with upper bounds on the complexity of the maximum network flow problem. The long and intriguing history of the search for such bounds is summarized below in

Table 1.1.

Upper Bounds on Max-Flow

1)	Ford and Fulkerson (1956)	Unbounded
2)	Edmonds and Karp (1969)	$O(V \cdot E^2)$
3)	Dinic (1970)	$O(V^2 \cdot E)$
4)	Karzanov (1973)	$O(V^3)$
5)	Cherkasky (1976)	$O(V^2 \cdot E^{1/2})$
6)	Galil (1978)	$O(V^{5/3} \cdot E^{2/3})$
7)	Malhotra, Kumar and Maheshwari (1978)	$O(V^3)$
8)	Galil and Naamad (1978)	$O(V \cdot E \cdot \log^2 V)$

Table 1.1

Each of these bounds has been demonstrated by the construction of a max-flow algorithm which has the specified worst case running time. Chapter 2 will consist of a survey of each of these algorithms. We note here that the algorithm developed by Malhotra, Kumar and Maheshwari [16], which we will call the MKM algorithm, does not result in an asymptotic improvement over the previous three algorithms. However, this algorithm is very simple and is probably the best algorithm to use on dense networks ($E \approx V^2$).

The determination of lower bounds on the computational complexity of the maximum network flow problem has thus far received little attention in the literature. In Chapter 3, however, we shall investigate one particular approach to establishing a non-linear lower bound on the complexity of max-flow. The technique we shall deal with is the polyhedral technique developed in 1977 by A.C. Yao, D.M. Avis and R.L. Rivest [21]. We will show that a straightforward application of this technique fails to

produce a non-linear lower bound on max-flow. In the process, however, we shall answer several questions concerning the facial structure of a class of polyhedra very closely related to the max-flow problem.

1.2 Basic Definitions and Concepts

This section will be devoted to a presentation of the basic definitions and concepts that have become standard in the study of max-flow. We will begin by defining a network and a legal flow function on a network. These concepts will then be used to develop a formal definition of the maximum network flow problem. Finally we will introduce the basic notions of a cut and a flow augmenting path.

Definition 1.1:

A network $\mathcal{N} = (G,s,t,c)$ is a 4-tuple with the properties:

1) $G = (V,E)$ is a finite directed graph composed of a set of vertices, V, and a set of edges, E.

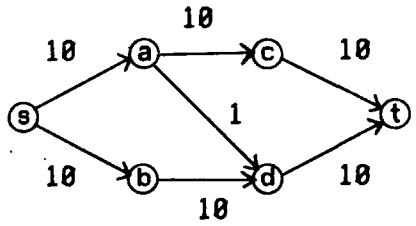
2) Two distinct vertices $s \in V$ and $t \in V$ are specified as the source and sink respectively.

3) Each edge $e \in E$ is assigned a non-negative real number $c(e)$, called the capacity of edge e.

For any vertex $v \in V$, $In(v)$ denotes the set of all edges incoming to v and $Out(v)$ denotes the set of all edges outgoing from v. An edge e, directed from a vertex u to a vertex v in a network $\mathcal{N}$, will often be denoted by the ordered pair $e=(u,v)$.

Example 1.1: (Network $\mathcal{N}_0$)

$V = \{s, a, b, c, d, t\}$
$E = \{(s,a), (s,b), (a,c), (a,d), (b,d), (c,t), (d,t)\}$
$c(e) = 10$ [for all $e \in E$ s.t. $e \neq (a,d)$]
$c(a,d) = 1$
$In(t) = \{(c,t), (d,t)\}$
$Out(s) = \{(s,a), (s,b)\}$



Definition 1.2:

A legal flow function f on a network $\mathcal{N}$ associates with each edge $e \in E$ a real number f(e) satisfying the conditions:

C1) $0 \leq f(e) \leq c(e)$, for each edge $e \in E$

C2) $\sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e) = 0$, for each vertex $v \in V - \{s, t\}$.

The value of a legal flow function f is defined to be:

(1.1) $$v(f) = \sum_{e \in In(t)} f(e) - \sum_{e \in Out(t)} f(e).$$

Informally we say f is a steady state flow from s to t and f(e) is the steady state flow through edge e. Thus, condition C1 tells us that the steady state flow through any edge must be non-negative and not exceed the capacity of that edge. Further, condition C2 states that any steady state flow from s to t must have the property that flow is conserved at every vertex other than the source and the sink.

Definition 1.3:

An edge e, incident upon vertices u and v in a network $\mathcal{N}$, is said to be useful from vertex u to vertex v with respect to a legal flow

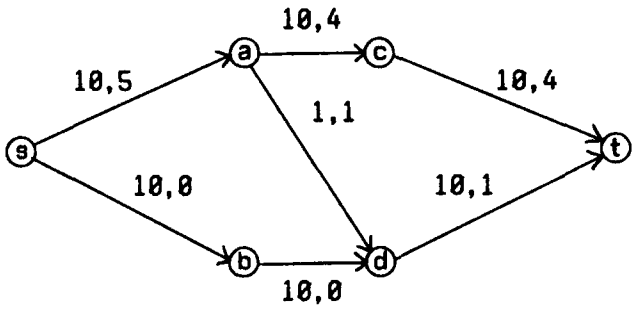
function f defined on $\mathcal{N}$, if either:

1) e=(u,v) and f(e)<c(e)
or
2) e=(v,u) and f(e)>0.

Example 1.2:

The following legal flow function f_0 is defined on network $\mathcal{N}_0$ from Example 1.1:

$f_0(s,a) = 5$ $\quad f_0(s,b) = 0$
$f_0(a,c) = 4$ $\quad f_0(a,d) = 1$
$f_0(b,d) = 0$ $\quad f_0(c,t) = 4$
$f_0(d,t) = 1$



$$v(f_0) = f_0(c,t) + f_0(d,t) - 0 = 4+1 = 5$$

Note - (s,b) is useful from s to b but not useful from b to s
(a,d) is useful from d to a but not useful from a to d
(a,c) is useful from a to c and from c to a

Let us now consider a legal flow function, defined on a network $\mathcal{N}$, whose value is maximum over the set of values of all legal flow functions defined on $\mathcal{N}$. Such a flow function is said to be maximum with respect to $\mathcal{N}$. Notice that the existence of a maximum flow function on a network is not open to question since a network is composed of only finite capacity edges. Further, notice that there may exist more than one distinct maximum flow function on a network.

Definition 1.4: (Max-Flow)

The maximum network flow problem is defined as the problem of

computing a maximum flow function on a network given as input.

We now turn our attention to the basic concepts underlying the theory on which the study of the maximum network flow problem has been built. In particular, we present the notions of a cut and a flow augmenting path.

Definition 1.5:

A cut in a network $\mathcal{N}$ is a set of vertices X with the properties:

1) $X \subset V$.

2) $s \in X$.

3) $t \in \bar{X}$ where $\bar{X} = V - X$.

The set of all edges $e \in E$ which are directed from a vertex in X to a vertex in $\bar{X}$ is denoted by $(X;\bar{X})$. The capacity of a cut X is defined to be:

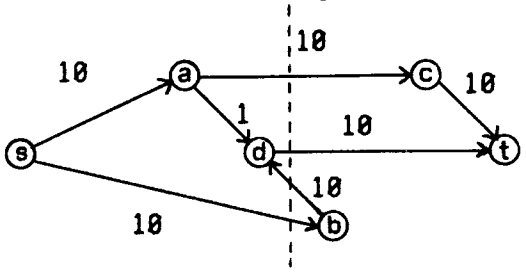
(1.2) $$C(X) = \sum_{e \in (X;\bar{X})} c(e).$$

Example 1.3:

The following cut X_0 is defined on network $\mathcal{N}_0$ from Example 1.1:

$X_0 = \{s, a, d\}$

$\bar{X}_0 = \{b, c, t\}$



$$C(X_0) = c(s,b) + c(a,c) + c(d,t) = 10+10+10 = 30$$

Definition 1.6:

A path p, from vertex v_1 to vertex v_n in a network $\mathcal{N}$, is a sequence of distinct vertices and edges $p = v_1e_1v_2e_2 \cdots v_{n-1}e_{n-1}v_n$ $(n\geq 2)$ such that $v_i \in V$ (for each $i=1,\ldots,n$), $e_i \in E$ (for each $i=1,\ldots,n-1$) and either $e_i = (v_i, v_{i+1})$, in which case e_i is said to be a forward edge in p, or $e_i = (v_{i+1}, v_i)$, in which case e_i is said to be a reverse edge in p, for each $i=1,\ldots,n-1$. Any path composed of only forward edges is called a chain. Note that a path is defined to be acyclic.

Definition 1.7:

A flow augmenting path, with respect to a legal flow function f on a network $\mathcal{N}$, is a path p′ with the property that each edge $e_i \in p'$ is useful from vertex $v_i \in p'$ to vertex $v_{i+1} \in p'$. A flow augmenting path from the source to the sink is called an s-t flow augmenting path. We shall see in the next section that an s-t flow augmenting path is a path along which f can be augmented (i.e. the value of f can be increased).

Example 1.4:

The following s-t flow augmenting path p_0 is defined with respect to the legal flow function f_0 from Example 1.2:

$$p_0 = s(s,b)b(b,d)d(a,d)a(a,c)c(c,t)t$$

Note - (s,b), (b,d), (a,c) and (c,t) are all forward edges along p_0
(a,d) is a reverse edge along p_0.

1.3 Fundamental Theorems

The study of the maximum network flow problem has been based on three fundamental theorems first proven by Ford and Fulkerson [9] in the mid 1950's. These theorems are known as the Augmenting Path Theorem, the Integral Flow Theorem and the Max-Flow Min-Cut Theorem. This section will be used to develop formal proofs of each of these theorems. We will begin by proving the following principle lemma which we alluded to at the end of the last section.

Lemma 1.1:

If there exists an s-t flow augmenting path p', with respect to a legal flow function f on a network $\mathcal{N}$, then there exists a legal flow function f' on $\mathcal{N}$ such that $v(f')>v(f)$.

Proof:

Let $p'=v_1e_1v_2e_2\ldots v_{n-1}e_{n-1}v_n$ $(n\geq 2)$ be an s-t flow augmenting path with respect to a legal flow function f on a network $\mathcal{N}$. Define the sets E_0, E_1 and E_2 as follows:

$$E_0 = \{e\in E \mid e \text{ is not an edge in } p'\}$$

$$E_1 = \{e\in E \mid e \text{ is a forward edge in } p'\}$$

$$E_2 = \{e\in E \mid e \text{ is a reverse edge in } p'\}.$$

Note that $E_1 \cup E_2$ is the set of all edges in p' and that $E_0 \cup E_1 \cup E_2$ is the set of all edges in $\mathcal{N}$ (i.e. $E_0 \cup E_1 \cup E_2 = E$).

Since p' is an s-t flow augmenting path, it follows from Definition 1.7 that $v_1=s$, $v_n=t$ and e_i is useful from v_i to v_{i+1}, for each $i=1,\ldots,n-1$. More explicitly, applying Definition 1.3 yields:

(1.3) $f(e)<c(e) \Rightarrow c(e)-f(e)>0$, for each edge $e \epsilon E_1$

and

(1.4) $f(e)>0$, for each edge $e \epsilon E_2$.

Now construct the function f' as follows:

(1.5) $\hat{c}(e) \leftarrow c(e)-f(e)$, for each $e \epsilon E_1$

(1.6) $\hat{c}(e) \leftarrow f(e)$, for each $e \epsilon E_2$

$$\delta \leftarrow \min_{e \epsilon E_1 \cup E_2} \hat{c}(e)$$

(1.7) $f'(e) \leftarrow f(e)$, for each $e \epsilon E_0$

(1.8) $f'(e) \leftarrow f(e)+\delta$, for each $e \epsilon E_1$

(1.9) $f'(e) \leftarrow f(e)-\delta$, for each $e \epsilon E_2$.

It must now be shown that f' is a legal flow function on $\mathcal{N}$ and that $v(f')>v(f)$. (Notice that f' is defined on all edges $e \epsilon E$.)

To show that f' is a legal flow function on $\mathcal{N}$, we must simply prove that f' satisfies both condition C1 and condition C2 of Definition 1.2. Let us first consider condition C1. Since f is a legal flow function on $\mathcal{N}$, it follows from Definition 1.2 that:

(1.10) $0 \leq f(e) \leq c(e)$, for all $e \epsilon E$.

Thus, applying (1.7) yields:

(1.11) $0 \leq f'(e) \leq c(e)$, for all $e \epsilon E_0$.

We now notice, from (1.3) and (1.4), that $\hat{c}(e)>0$ for all $e \epsilon E_1 \cup E_2$ and thus $\delta>0$. Combining this with (1.8) and (1.9), we have:

(1.12) $f'(e)>f(e)$, for all $e \epsilon E_1$

and

(1.13) $f'(e)<f(e)$, for all $e \epsilon E_2$.

We can now apply (1.10) and (1.12) to (1.8) to obtain:

(1.14) $$0<f'(e)\leq c(e), \text{ for all } e\in E_1.$$

Similarly, we can apply (1.10) and (1.13) to (1.9) to obtain:

(1.15) $$0\leq f'(e)<c(e), \text{ for all } e\in E_2.$$

Therefore, combining (1.11), (1.14) and (1.15), we have:

$$0\leq f'(e)\leq c(e), \text{ for all } e\in E$$

and thus f' satisfies condition C1 of Definition 1.2.

We must now prove that f' satisfies condition C2 of Definition 1.2. This is accomplished by considering any vertex $v'\in V-\{s,t\}$. Since f is a legal flow function on $\mathcal{N}$, it follows from Definition 1.2 that:

(1.16) $$\sum_{e\in In(v')} f(e) - \sum_{e\in Out(v')} f(e) = 0.$$

Further, if v' is not a vertex along the path p', then all of the edges incident upon v' must be contained in the set E_0. Thus, combining (1.7) and (1.16) yields:

$$\sum_{e\in In(v')} f'(e) - \sum_{e\in Out(v')} f'(e) = \sum_{e\in In(v')} f(e) - \sum_{e\in Out(v')} f(e) = 0.$$

$$\text{(for all } v'\in V-\{s,t\} \text{ and } v'\notin p')$$

If, however, v' is a vertex along the path p', then let i be the index of v' along p' (i.e. $v' = v_i\in p'$). Since p' is acyclic and $v'\notin\{s,t\}$, there must be exactly two distinct edges incident upon v' which are contained in p', namely e_{i-1} and e_i. All other edges incident upon v' are therefore contained in the set E_0. We must now consider each of the following four possible cases:

1) $e_{i-1} \in E_1$ and $e_i \in E_1$

2) $e_{i-1} \in E_1$ and $e_i \in E_2$

3) $e_{i-1} \in E_2$ and $e_i \in E_1$

4) $e_{i-1} \in E_2$ and $e_i \in E_2$.

For case 1, we combine (1.7), (1.8) and (1.16) to obtain:

$$\sum_{e \in In(v')} f'(e) - \sum_{e \in Out(v')} f'(e) = \left[\left(\sum_{e \in In(v')} f(e)\right) + \delta\right] - \left[\left(\sum_{e \in Out(v')} f(e)\right) + \delta\right]$$

$$= \left(\sum_{e \in In(v')} f(e)\right) - \left(\sum_{e \in Out(v')} f(e)\right) + \delta - \delta$$

$$= \sum_{e \in In(v')} f(e) - \sum_{e \in Out(v')} f(e) = 0.$$

The same result is proven for cases 2, 3 and 4 similarly. We therefore have that:

$$\sum_{e \in In(v')} f'(e) - \sum_{e \in Out(v')} f'(e) = 0, \quad \text{for all } v' \in V - \{s, t\}$$

and thus f' satisfies condition C2 of Definition 1.2.

To show that $v(f') > v(f)$, we first notice that there is exactly one edge incident upon t which is contained in the path p', namely e_{n-1}. All other edges incident upon t are therefore contained in the set E_0. We must now consider each of the following two possible cases:

1) $e_{n-1} \in E_1$

2) $e_{n-1} \in E_2$.

For case 1, we can simply combine (1.1) of Definition 1.2, (1.7) and (1.8) to obtain:

$$v(f') = \sum_{e \in In(t)} f'(e) - \sum_{e \in Out(t)} f'(e)$$

$$= \left[\left(\sum_{e \in In(t)} f(e)\right) + \delta\right] - \sum_{e \in Out(t)} f(e)$$

$$= \sum_{e \in In(t)} f(e) - \sum_{e \in Out(t)} f(e) + \delta$$

$$= v(f) + \delta > v(f). \qquad \text{(since } \delta > 0\text{)}$$

The same result is proven for case 2 similarly. We therefore have that $v(f') > v(f)$.

□

The proof of Lemma 1.1 reveals a fairly simple procedure for augmenting an existing legal flow function on a network, given a corresponding s-t flow augmenting path. The procedure consists of determining the "excess capacity" along each edge in the flow augmenting path, as defined by (1.5) and (1.6), and then increasing the flow along all forward edges and decreasing the flow along all reverse edges in the path. The amount by which flow is increased or decreased along each edge is simply the minimum excess capacity over all edges in the flow augmenting path.

We now present a lemma which will be useful in demonstrating a relationship between the capacity of a cut and the value of a legal flow function on a network.

Lemma 1.2:

Given any cut X and any legal flow function f on a network $\mathcal{N}$,

$$v(f) = \sum_{e \in (X;\bar{X})} f(e) - \sum_{e \in (\bar{X};X)} f(e).$$

Proof:

Let X be any cut and f be any legal flow function on a network $\mathcal{N}$.

Then by condition C2 of Definition 1.2 and the fact that $s \notin \bar{X}$, we have:

$$(1.17) \qquad \sum_{v \in \bar{X} - \{t\}} \left(\sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e) \right) = 0.$$

Combining (1.1) of Definition 1.2 and (1.17) yields:

$$(1.18) \qquad v(f) = \sum_{e \in In(t)} f(e) - \sum_{e \in Out(t)} f(e) + \sum_{v \in \bar{X} - \{t\}} \left(\sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e) \right).$$

Simplifying (1.18) we obtain:

$$v(f) = \sum_{v \in \bar{X}} \left(\sum_{e \in In(v)} f(e) - \sum_{e \in Out(v)} f(e) \right)$$

$$= \sum_{v \in \bar{X}} \sum_{e \in In(v)} f(e) - \sum_{v \in \bar{X}} \sum_{e \in Out(v)} f(e).$$

Thus, we have that the value of f is equal to the sum of the flow along all edges directed into a vertex in $\bar{X}$ minus the sum of the flow along all edges directed out of a vertex in $\bar{X}$. If we now consider separately those edges directed from a vertex in X to a vertex in $\bar{X}$ and those edges directed from a vertex in $\bar{X}$ to a vertex in X, we obtain:

$$v(f) = \left(\sum_{e \in (X;\bar{X})} f(e) + \sum_{e \in (\bar{X};\bar{X})} f(e) \right) - \left(\sum_{e \in (\bar{X};X)} f(e) + \sum_{e \in (\bar{X};\bar{X})} f(e) \right)$$

$$= \sum_{e \in (X;\bar{X})} f(e) - \sum_{e \in (\bar{X};X)} f(e).$$

□

Corollary 1.1:

If the value of a legal flow function f′ is equal to the capacity of some cut X′ on a network $\mathcal{N}$, then f′ is maximum on $\mathcal{N}$ and X′ has minimum capacity over all cuts on $\mathcal{N}$.

Proof:

Let X be any cut and f be any legal flow function on the network

$\mathcal{N}$. Then by Lemma 1.2 we have that:

$$v(f) = \sum_{e\in(X;\bar{X})} f(e) - \sum_{e\in(\bar{X};X)} f(e).$$

Applying condition C1 of Definition 1.2 we obtain:

$$v(f) \leq \sum_{e\in(X;\bar{X})} c(e)$$

and thus by (1.2) of Definition 1.5:

(1.19) $$v(f) \leq C(X).$$

Therefore, if (1.19) holds by equality for some legal flow function f' and some cut X' on the network $\mathcal{N}$, then f' must be maximum on $\mathcal{N}$ and X' must have minimum capacity over all cuts on $\mathcal{N}$.

□

We are now ready to present the three fundamental theorems upon which the study of max-flow has been built.

Theorem 1.1: (Augmenting Path Theorem)

A legal flow function f on a network $\mathcal{N}$ is maximum if and only if there exists no s-t flow augmenting path with respect to f on $\mathcal{N}$.

Proof:

Clearly, if there exists an s-t flow augmenting path with respect to a legal flow function f on a network $\mathcal{N}$, then by Lemma 1.1 f is not maximum on $\mathcal{N}$. Assume now that there is no s-t flow augmenting path with respect to f on $\mathcal{N}$ and define the set S as follows:

$$S = \{v\in V \mid \exists \text{ an s-v flow augmenting path with respect to f on } \mathcal{N}\} \cup \{s\}.$$

Since there is no s-t flow augmenting path with respect to f on $\mathcal{N}$, it can easily be seen from Definition 1.5 that S forms a cut in $\mathcal{N}$. Further, from Definition 1.3 and Definition 1.7 we have that f(e)=c(e) for each edge $e\in(S;\bar{S})$ and f(e) = 0 for each edge $e\in(\bar{S};S)$. We can now apply Lemma 1.2 to obtain:

$$\begin{aligned} v(f) &= \sum_{e\in(S;\bar{S})} f(e) - \sum_{e\in(\bar{S};S)} f(e) \\ &= \sum_{e\in(S;\bar{S})} c(e) \\ &= C(S). \end{aligned}$$

Thus, by Corollary 1.2 we have that f is maximum on $\mathcal{N}$.

□

Theorem 1.2: (Integral Flow Theorem)

There exists an integral valued maximum flow function on any network defined by an integral valued capacity function.

Proof:

Let $\mathcal{N}$ be any network defined by an integral valued capacity function and let f_ϕ be the zero flow function on $\mathcal{N}$, defined by $f_\phi(e)=0$ for each edge $e\in E$ (Notice that such a flow function will be a legal flow function on any network). We can now compute a maximum flow function on $\mathcal{N}$ as follows:

```
while there exists an s-t flow augmenting path
        with respect to f_φ on N
    do augment f_φ as outlined in
        the proof of Lemma 1.1.
```

An examination of (1.5) through (1.9) reveals that the legal flow function generated at each iteration of this procedure will be integral valued. Thus by Lemma 1.1, the value of each successive legal flow function generated must be at least one integral unit greater than the value of the previous legal flow function. Combining this fact with (1.19) and Theorem 1.1, we now have that our procedure must halt within a finite number of steps, yielding an integral valued maximum flow function on $\mathcal{N}$.

□

Theorem 1.3: (Max-Flow Min-Cut Theorem)

The value of any maximum flow function on a network $\mathcal{N}$ is equal to the minimum cut capacity over all cuts on $\mathcal{N}$.

Proof:

Let f be any maximum flow function defined on a network $\mathcal{N}$. Applying Theorem 1.1 we have that there is no s-t flow augmenting path with respect to f on $\mathcal{N}$. It now follows immediately from the proof of Theorem 1.1 that there exists a cut S on $\mathcal{N}$ such that:

$$v(f) = C(S).$$

Further, by Corollary 1.2 we have that the cut S must have minimum capacity over all cuts on $\mathcal{N}$. Thus, the value of any maximum flow function on $\mathcal{N}$ is equal to the minimum cut capacity over all cuts on $\mathcal{N}$.

□

We shall see in Chapter 2 how the previous three theorems form the basis for all the max-flow algorithms that have thus far been developed.

In fact, we will see that each algorithm is actually a variation of the procedure given in the proof of Theorem 1.2.

CHAPTER 2 - UPPER BOUNDS ON MAX-FLOW

2.1 Introduction

The formal definition of the maximum network flow problem, as presented in Section 1.2, can be explicitly stated as follows. Given any network $\mathcal{N}$ as input, compute values of the variables X_e [for each $e \in E$] so as to maximize the objective function

$$\sum_{e \in In(t)} X_e - \sum_{e \in Out(t)} X_e$$

subject to the constraints

$$X_e \geq 0, \quad \text{for each } e \in E$$

$$X_e \leq c(e), \quad \text{for each } e \in E$$

$$\sum_{e \in In(v)} X_e - \sum_{e \in Out(v)} X_e = 0, \quad \text{for each } v \in V-\{s,t\}.$$

Thus the maximum network flow problem can be viewed as an optimization problem in which a linear function must be maximized subject to a system of linear equations and linear inequalities. G.B. Dantzig [4] developed an algorithm in the early 1950's, known as the simplex method, which could be used to solve such linear programming problems. Although it would not be incorrect to consider the simplex method to be the first max-flow algorithm, it is usually not treated as such. The simplex method is a very general algorithm which has an unbounded worst case running time. We have included it here only for the sake of completeness.

L.R. Ford and D.R. Fulkerson [9] were the first to produce significant research results concerning, specifically, the maximum network flow problem. In the mid 1950's they proved the Augmenting Path Theorem,

the Integral Flow Theorem, and the Max-Flow Min-Cut Theorem. These fundamental results led directly to their development of the labeling algorithm for solving the maximum network flow problem. The labeling algorithm is a straightforward algorithm which simply augments an existing legal flow function along some s-t flow augmenting path in a network. This process is then repeated until there no longer exist any s-t flow augmenting paths in the network. The labeling algorithm, although it also has an unbounded worst case running time, remained in successful use for almost 15 years.

In 1969, J. Edmonds and R.M. Karp [7] developed a variation of the labeling algorithm which utilized a Breadth First Search in picking out the s-t flow augmenting paths in order of increasing length. This resulted in a much more efficient algorithm with a bounded $O(|V|\cdot|E|^2)$ worst case running time. Independently and a short time later, E.A. Dinic [5] developed an improved version of Edmonds and Karp's algorithm. Dinic also utilized the technique of Breadth First Search but he developed an algorithm with time complexity $O(|V|^2\cdot|E|)$.

A.V. Karzanov [14] modified Dinic's algorithm in 1973 to obtain an $O(|V|^3)$ max-flow algorithm. Karzanov's algorithm was unique in that it simultaneously augmented an existing legal flow function along several s-t flow augmenting paths. In 1976, B.V. Cherkasky [10] showed how to combine Dinic's algorithm with Karzanov's algorithm to produce a new and very complex $O(|V|^2\cdot|E|^{1/2})$ max-flow algorithm. Two years later, Zvi Galil [10] improved Cherkasky's algorithm to $O(|V|^{5/3}\cdot|E|^{2/3})$ by developing a technique for retaining useful information about the structure of the

network.

In 1978 V.M. Malhotra, M. Pramodh Kumar and S.N. Maheshwari [16] discovered a very simple $O(|V|^3)$ max-flow algorithm which we shall call the MKM algorithm. Their algorithm, similar to Karzanov's algorithm in that it simultaneously augments along several s-t flow augmenting paths, requires very little overhead. Although the MKM algorithm does not result in an asymptotic improvement over the previous three algorithms, its simplicity makes it perhaps the best algorithm to use on very dense networks ($E \approx V^2$).

Finally, Galil and A. Naamad [12] have recently developed a modification to the original Dinic algorithm which results in an algorithm with time complexity $O(|V| \cdot |E| \cdot \log^2 |V|)$. Their modification of Dinic's algorithm is similar to Galil's modification of Cherkasky's algorithm. Once again a technique is developed for retaining useful information about the structure of the network.

The remainder of this chapter will be devoted to a closer examination of each of the maximum network flow algorithms.

2.2 Ford and Fulkerson

Ford and Fulkerson [9] developed the first maximum network flow algorithm, known as the labeling algorithm, in 1956. Their algorithm simply augments, along some s-t flow augmenting path, the existing legal flow function on a network. This augmentation is then repeated until there no longer exist any s-t flow augmenting paths on the network. In practice the zero flow function (i.e. $f(e)=0$ for all edges e) is used as the initial existing legal flow function.

The labeling algorithm is composed of two basic routines which are iterated until a maximum flow function is computed. The first routine essentially searches in a systematic way for an s-t flow augmenting path on the network. The second routine then augments the existing legal flow function along this flow augmenting path. The actual flow augmentation is performed exactly as outlined in the proof of Lemma 1.1. The following explanation of the labeling algorithm is taken from [13]:

Step 1. Labeling Process.

Every vertex is always in one of three states, labeled and scanned, labeled and unscanned, or unlabeled. A vertex is labeled and scanned if it has a label and we have inspected all vertices adjacent to it. A vertex is labeled and unscanned if it has a label but not all vertices adjacent to it have been inspected. A vertex is unlabeled if it has no label.

Initially, all vertices are unlabeled. A label for a vertex v_j always has two parts. The first part is the index of a vertex v_i, which indicates that we can send flow from v_i to v_j, and the second part is a number which indicates the maximum amount of flow we can send from the source to v_j without violating the capacity constraints. We first assign the label $[s^+, \epsilon(s)=\infty]$ to the source, v_s. The first label simply says that we can send flow from the source to itself; the number ∞ indicates that there is no upper bound on how much can be sent. The source is now labeled and unscanned and all other vertices are unlabeled. In general, select a vertex v_j which is labeled and unscanned. Assume v_j has a label of the form $[i^+, \epsilon(j)]$ or $[i^-, \epsilon(j)]$. For all adjacent vertices v_k which are unlabeled, adjacent to v_j via an edge directed from v_j to v_k, and for which

the edge $e=(v_j,v_k)$ is useful from v_j to v_k (i.e. $f(v_j,v_k)<c(v_j,v_k)$), assign the label $[j^+,\epsilon(k)]$ to v_k, where:

$$\epsilon(k) = \min[\epsilon(j),\ c(v_j,v_k)-f(v_j,v_k)].$$

For all adjacent vertices v_k which are unlabeled, adjacent to v_j via an edge directed from v_k to v_j, and for which the edge $e=(v_k,v_j)$ is useful from v_j to v_k (i.e. $f(v_k,v_j)>0$), assign the label $[j^-,\epsilon(k)]$ to v_k, where:

$$\epsilon(k) = \min[\epsilon(j),\ f(v_k,v_j)].$$

The + and the - signs in the labels indicate whether the corresponding edges appear as forward or reverse edges in the s-t flow augmenting path. Now all the vertices adjacent to v_j have labels; v_j is considered to be labeled and scanned and may be disregarded during the rest of this step. (If one inspects all the vertices adjacent to v_j and cannot label all these vertices, then v_j is also considered to be a labeled and scanned vertex.) All the vertices v_k are now labeled and unscanned.

Continue to assign labels to vertices adjacent to labeled and unscanned vertices until either the sink is labeled or no more labels can be assigned and the sink is unlabeled. If the sink cannot be labeled, no s-t flow augmenting path exists and, hence, the existing flow function is maximum. If the sink is labeled, an s-t flow augmenting path has been found and the flow augmentation can be performed using step 2.

Step 2. Flow Change.

Assume that the sink is labeled $[k^+,\epsilon(t)]$. Let $f(v_k,v_t)\leftarrow f(v_k,v_t)+\epsilon(t)$ and turn to v_k. If v_k is labeled $[j^+,\epsilon(k)]$, let $f(v_j,v_k)\leftarrow f(v_j,v_k)+\epsilon(t)$ and turn to v_j. If v_k is labeled $[j^-,\epsilon(k)]$, let $f(v_k,v_j)\leftarrow f(v_k,v_j)-\epsilon(t)$ and turn to v_j. Continue until the source is

reached. Erase the labels on all the vertices and go back to step 1.

When the labeling algorithm terminates, the set of all labeled vertices clearly forms a cut in the network. Further, applying Lemma 1.2 reveals that the capacity of this cut must be equal to the value of the existing legal flow function on the network. Thus by Corollary 1.1, the existing legal flow function must be maximum. It now remains to be shown that the labeling algorithm will always terminate within a finite number of steps.

Let us first consider the case in which the network is defined by an integral valued capacity function. By the same argument as that used in the proof of Theorem 1.2, it can easily be seen that the labeling algorithm will terminate after at most $v(f_{max})$ iterations, where $v(f_{max})$ is the finite value of a maximum flow function on the network. Further, each iteration will require at most $O(|E|)$ operations since each edge is examined at most twice in the labeling procedure and at most once in the augmenting procedure. Thus we have that the labeling algorithm will correctly compute, in time $O(|E|\cdot v(f_{max}))$, a maximum flow function on any network defined by an integral valued capacity function. It can also be shown, however, that there actually exist networks which force the labeling algorithm to perform $v(f_{max})$ iterations. Consider for example the network n_1 in Figure 2.1.

Network n_1

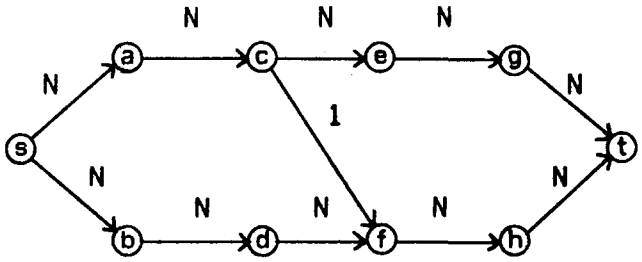


Figure 2.1

If the labeling algorithm, beginning with the zero flow function on n_1, augments only along the paths s(s,a)a(a,c)c(c,f)f(f,h)h(h,t)t and s(s,b)b(b,d)d(d,f)f(c,f)c(c,e)e(e,g)g(g,t)t in alternating order, then $2 \cdot N = v(f_{max})$ flow augmentations will be required. Thus the algorithm will iterate $v(f_{max})$ times. Notice that the inefficiency in this example is based on the fact that the labeling algorithm permits the augmentation, at each iteration, along any one of several existing s-t flow augmenting paths.

We shall now consider the case in which the network is not defined by an integral valued capacity function. Ford and Fulkerson [9] were able to demonstrate the somewhat surprising result that their labeling algorithm might fail to terminate if the network was composed of irrational edge capacities. This result was based on interpreting the labeling process broadly enough to permit the selection of any s-t flow augmenting path at each iteration of the computation. Thus the labeling algorithm essentially has an unbounded worst case running time. We remark, however, that computers only deal with rational numbers and thus in practice we could expect the labeling algorithm to halt and yield a correct answer. In fact,

despite its weaknesses, the labeling algorithm was successfully used for almost 15 years.

2.3 Edmonds and Karp

In 1969 Edmonds and Karp [7] showed how the labeling algorithm could be modified to obtain a bounded worst case running time. In light of our previous remarks, it should not be surprising to learn that their modification was essentially an ordering on the selection of the s-t flow augmenting paths. Edmonds and Karp suggested augmenting along the shortest s-t flow augmenting path (i.e. an s-t flow augmenting path containing a minimum number of edges) at each iteration. This can be easily accomplished by modifying the labeling process so that the vertices are scanned in the same order in which they receive labels (i.e. by imposing a Breadth First Search on the labeling process). The remainder of the labeling algorithm is unchanged. The running time bound on Edmonds and Karp's "first labeled, first scanned" modification of the labeling algorithm is derived from the following results [15].

Consider any network N upon which there is defined a legal flow function f. Let $\sigma_u^{(k)}$ denote the minimum number of edges in an s-u flow augmenting path after k augmentations of f. Similarly, let $\tau_u^{(k)}$ denote the minimum number of edges in a u-t flow augmenting path after k augmentations of f.

Lemma 2.1:

If each flow augmentation of f is made along an s-t augmenting path

with a minimum number of edges, then:

$$\sigma_u^{(k+1)} \geq \sigma_u^{(k)}$$

and

$$\tau_u^{(k+1)} \geq \tau_u^{(k)}$$

for all u,k.

Proof: (From [15])

Assume that $\sigma_u^{(k+1)} < \sigma_u^{(k)}$, for some u,k. Moreover, let:

$$\sigma_u^{(k+1)} = \min_v \{\sigma_v^{(k+1)} \mid \sigma_v^{(k+1)} < \sigma_v^{(k)}\}. \tag{2.1}$$

Clearly $\sigma_u^{(k+1)} \geq 1$ (only $\sigma_s^{(k+1)} = 0$), and there must be some final edge (u,v) or (v,u) in a shortest s-u flow augmenting path after the $(k+1)^{st}$ augmentation of f. Suppose this edge is (v,u), a forward edge, with $f(v,u)<c(v,u)$ (the proof is similar for (u,v)). Then $\sigma_u^{(k+1)} = \sigma_v^{(k+1)} + 1$ and by (2.1),

$$\sigma_u^{(k+1)} \geq \sigma_v^{(k)} + 1. \tag{2.2}$$

Further, it must have been that $f(v,u)=c(v,u)$ after the k^{th} augmentation of f; otherwise $\sigma_u^{(k)} \leq \sigma_v^{(k)}+1 \leq \sigma_u^{(k+1)}$, contrary to the assumption. But if $f(v,u)=c(v,u)$ after the k^{th} augmentation of f and $f(v,u)<c(v,u)$ after the $(k+1)^{st}$ augmentation of f, it follows that (v,u) was a reverse edge in the $(k+1)^{st}$ s-t flow augmenting path along which f was augmented. Since that path contained a minimum number of edges,

$$\sigma_v^{(k)} = \sigma_u^{(k)} + 1.$$

Combining this with (2.2), however, we obtain:

$$\sigma_u^{(k)}+2 \leq \sigma_u^{(k+1)},$$

contrary to our assumption. The assumption that $\sigma_u^{(k+1)} < \sigma_u^{(k)}$ is

therefore false.

The proof that $\tau_u^{(k+1)} \geq \tau_u^{(k)}$ parallels the above.

□

Theorem 2.1:

If each flow augmentation of f is made along an s-t augmenting path with a minimum number of edges, then a maximum flow function is obtained after no more than $|V| \cdot |E|/2$ augmentations of f.

Proof: (From [15])

Each time an augmentation of f is made, at least one edge in the s-t augmenting path is "critical" in the sense that it limits the amount of augmentation. The flow through such an edge (u,v) is either increased to capacity or decreased to zero. Suppose (u,v) is a critical edge in the $(k+1)^{st}$ s-t augmenting path. The number of edges in the augmenting path is $\sigma_u^{(k)} + \tau_u^{(k)} = \sigma_v^{(k)} + \tau_v^{(k)}$.

The next time edge (u,v) appears in an s-t augmenting path, say the $(\ell+1)^{st}$, it will be with the opposite orientation. That is, if it was a forward edge in the $(k+1)^{st}$, it is a reverse edge in the $(\ell+1)^{st}$, and vice versa. If (u,v) was a forward edge in the $(k+1)^{st}$ s-t augmenting path (the proof is similar for a reverse edge), then:

$$\sigma_v^{(k)} = \sigma_u^{(k)} + 1$$

and

$$\sigma_u^{(\ell)} = \sigma_v^{(\ell)} + 1.$$

By Lemma 2.1, however, $\sigma_v^{(\ell)} \geq \sigma_v^{(k)}$ and $\tau_u^{(\ell)} \geq \tau_u^{(k)}$. Thus we have that:

$$\sigma_u^{(\ell)} + \tau_u^{(\ell)} \geq \sigma_u^{(k)} + \tau_u^{(k)} + 2.$$

It follows that each succeeding s-t augmenting path in which (u,v) is a critical edge is at least two edges longer than the preceding one.

No flow augmenting path may contain more than |V|-1 edges. Therefore, no edge may be a critical edge more than |V|/2 times. But each s-t augmenting path along which f is augmented contains a critical edge. Therefore there can be no more than $|V| \cdot |E|/2$ successive s-t flow augmenting paths and this completes the proof.

□

Since Edmonds and Karp's algorithm differs from the original labeling algorithm only in the order in which the unscanned vertices are scanned, it follows that their algorithm will yield a maximum flow function if it halts and that their algorithm will require only O(|E|) operations per iteration. By Theorem 2.1, however, Edmonds and Karp's algorithm is guaranteed to halt after at most $|V| \cdot |E|/2$ iterations. Further, this result is independent of the capacity function; holding for irrational valued capacity functions as well as integral valued capacity functions. Thus Edmonds and Karp's algorithm will correctly compute, in time $O(|V| \cdot |E|^2)$, a maximum flow function on any network given as input.

2.4 Dinic

In 1970, E.A. Dinic [5] developed a maximum network flow algorithm with a bounded $O(|V|^2 \cdot |E|)$ worst case running time. Dinic's algorithm, like Edmonds and Karp's algorithm, is based on successive flow

augmentations along s-t flow augmenting paths of minimum length. Dinic, however, noticed that a single breadth first search of a flow network could be used to isolate all minimum length s-t flow augmenting paths. Based on this observation, Dinic's max-flow algorithm is much more efficient than the "first labeled, first scanned" algorithm of Edmonds and Karp. In order to present Dinic's algorithm, we must first introduce the notion of a layered network.

Definition 2.1:

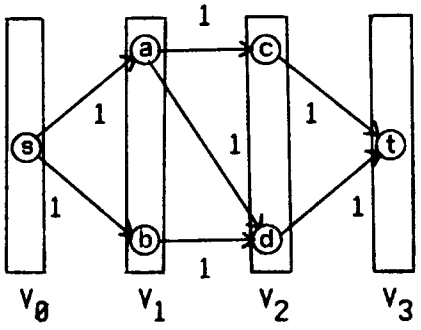
A layered network LN is a network whose vertex set is partitioned into disjoint subsets $V_0, V_1, \ldots, V_\ell$ such that $V_0 = \{s\}$ and $V_\ell = \{t\}$. We say that V_i is the i^{th} layer in LN (for $0 \le i \le \ell$) and that ℓ is the length of the layered network LN. Each edge $e=(u,v)$ in LN has the property that if $u \in V_i$ then $v \in V_{i+1}$ (i.e. every edge in LN is directed from one layer to the next).

Example 2.1: (Layered Network LN_0)

$V_0=\{s\}$, $V_1=\{a,b\}$, $V_2=\{c,d\}$, $V_3=\{t\}$

$E=\{(s,a),(s,b),(a,c),(a,d),(b,d),(c,t),(d,t)\}$

$c(e)=1$ [for all $e \in E$]



Each iteration of Dinic's algorithm is called a phase and each phase is divided into two procedures. The first procedure generates a layered network from the original input network, in such a way that the

layered network isolates all existing minimum length s-t flow augmenting paths. The second procedure then uses this layered network to successively augment the existing legal flow function along these minimum length s-t flow augmenting paths. We will first describe how Dinic's algorithm performs each of these procedures and then we will state the entire algorithm.

The first task performed during each phase of Dinic's algorithm is the construction of the layered network. The layers composing this network are created from a breadth first search of the input network. This breadth first search begins at the source and traverses only forward directed edges with flow less than capacity or backward directed edges with flow greater than zero (i.e. only "useful" edges). The search terminates when either the sink is reached or no new vertices can be visited and the sink has not been reached. When the sink cannot be reached, however, the existing legal flow function is a maximum flow function and Dinic's algorithm halts. Notice that performing the search in this way assures that there exists a flow augmenting path from the source to each vertex visited. The layers, V_i, are finally formed by partitioning the vertices visited according to the length of their path of discovery from the source. Every vertex $v \in V_i$ will then have the property that the shortest s-v flow augmenting path, with respect to the existing legal flow function on the input network, is of length i. Therefore, the length of the layered network constructed will be equal to the length of the shortest existing s-t flow augmenting path. The following procedure formalizes this construction [8].

```
procedure LN(N, f):
    begin
    .  V_0 := {s};
    .  i := 0;
    .  while (i := i+1) > 0 do
    .      begin
    .      .  T := {v∈V | v∉V_j for j<i and there exists a
    .      .       useful edge from a vertex in V_{i-1} to v};
    .      .  if T=φ then halt (the existing f is maximum)
    .      .      else if t∈T then begin
    .      .                        .  ℓ := i;
    .      .                        .  V_ℓ := {t};
    .      .                        .  i := -1
    .      .                        end
    .      .           else V_i := T
    .      end
    end.
```

Once the layers have been generated, the layered network's edge set and capacity function are constructed. The edge set is simply constructed from all edges in the original input network which are useful from a vertex in V_i to a vertex in V_{i+1} (for all $0 \le i \le \ell$). Every edge in the layered network, however, is directed from the i^{th} layer to the $(i+1)^{st}$ layer, regardless of its orientation in the input network. Any edge in the layered network whose orientation is different in the input network is said

to be a reverse edge in the layered network. Similarly, any edge in the layered network whose orientation is the same in the input network is said to be a forward edge in the layered network. The capacity function $\hat{c}$ is then created from the excess capacity along each edge in the layered network as follows:

$$\hat{c}(e) \leftarrow c(e) - f(e), \quad \text{for all forward edges in the layered network}$$

and

$$\hat{c}(e) \leftarrow f(e), \quad \text{for all reverse edges in the layered network,}$$

where f is the existing legal flow function and c is the capacity function on the input network. It should now be clear that every s-t chain in the layered network corresponds to an existing minimum length s-t flow augmenting path in the input network. Further, every minimum length s-t flow augmenting path in the input network is represented by an s-t chain in the layered network. Thus the layered network constructed during each phase of Dinic's algorithm essentially isolates all existing minimum length s-t flow augmenting paths.

The second task performed during each phase of Dinic's algorithm is the flow augmentation. Starting at the sink in the newly constructed layered network, the algorithm follows edges backward to the source to find an s-t flow augmenting path. (Recall that every s-t chain in the layered network corresponds to an existing s-t flow augmenting path in the input network.) The existing legal flow function is then augmented along this path as outlined in the proof of Lemma 1.1. Notice that the excess capacity along each edge in the flow augmenting path is given by the layered network capacity function, $\hat{c}$. After augmenting along this path, the algorithm

adjusts the capacity function $\hat{c}$ to reflect the new excess capacity along each edge in the path. It also deletes from the layered network all edges in the path whose excess capacity drops to zero. Finally, it deletes from the layered network all vertices and their incident edges which are no longer reachable from the source or the sink. This is accomplished by deleting all vertices which either have no incoming edges or have no outgoing edges, and continuing to delete such vertices until every vertex left in the layered network has at least one incoming and one outgoing edge. After all necessary deletions have been performed, the algorithm searches for another flow augmenting path and the augmentation process is repeated. This continues until there no longer exist any s-t chains in the layered network.

Dinic's complete max-flow algorithm is: (From [20])

```
procedure DINIC(N):
  begin
  . initialize existing legal flow function f, on input network N, to 0;
  . while "true" do
  .    begin
  .    . construct layered network, LN;
  .    . for each vertex v in LN do
  .    .    begin
  .    .    . calculate indegree (v);
  .    .    . calculate outdegree (v);
  .    .    . if (indegree (v)=0) or (outdegree (v)=0) then
  .    .    .      add v to nullist
```

```
. . end;
. . while t is a vertex in LN do
. . begin
. . . trace back from t to s to find an augmenting path;
. . . augment f along this path;
. . . update ĉ with new excess capacity along each edge in path;
. . . delete from LN all edges along path which now have zero
. . .   excess capacity (i.e. ĉ(e)=0), updating indegrees,
. . .   outdegrees, and nullist;
. . . while some vertex v is on nullist do
. . .   delete v and incident edges from LN and from nullist,
. . .   updating indegrees, outdegrees, and nullist
. . end
. end
end.
```

Recall that Dinic's algorithm terminates when the breadth first search performed in constructing each layered network fails to reach the sink. When this occurs, however, the set of all visited vertices clearly forms a cut in the input network. Further, applying Lemma 1.2 reveals that the capacity of this cut must be equal to the value of the existing legal flow function. Thus by Corollary 1.1, the existing legal flow function must be maximum. It now remains to be shown that Dinic's algorithm will always terminate within time $O(|V|^2 \cdot |E|)$. This is proven as a consequence of the following lemma [8] which shows that the number of phases is bounded

by $|V|$.

Let ℓ_k denote the length of the layered network constructed during the k^{th} phase of Dinic's algorithm.

Lemma 2.2:

If the $(k+1)^{st}$ phase of Dinic's algorithm is not the last, then $\ell_{k+1} > \ell_k$.

Proof: (From [8])

Consider any s-t chain in the layered network constructed during the $(k+1)^{st}$ phase of Dinic's algorithm:

$$s \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots v_{\ell_{k+1}-1} \xrightarrow{e_{\ell_{k+1}}} t.$$

First, let us assume that all the vertices in this chain appear in the k^{th} layered network. Let V_j be the j^{th} layer of the k^{th} layered network. We claim that if $v_a \epsilon V_b$ then $a \geq b$. This is proven by induction on a. For $a=0$, $(v_0=s)$ the claim is obviously true. Now assume $v_{a+1} \epsilon V_c$. If $c \leq b+1$ the inductive step is trivial. If, however, $c>b+1$ then the edge e_{a+1} was not used in the k^{th} phase since it was not even in the k^{th} layered network, in which only edges between adjacent layers appear. But if e_{a+1} was not used in the k^{th} phase and is useful from v_a to v_{a+1} in the beginning of the $(k+1)^{st}$ phase, then it was useful from v_a to v_{a+1} in the beginning of the k^{th} phase. Thus, v_{a+1} cannot belong to V_c (by procedure LN). Now, in particular, $t=v_{\ell_{k+1}}$ and $t \epsilon V_{\ell_k}$. Therefore, $\ell_{k+1} \geq \ell_k$. Further, equality cannot hold because then the entire s-t chain would have

been in the k^{th} layered network, and if all its edges are still useful at the beginning of the $(k+1)^{st}$ phase then we have a contradiction to the termination of the k^{th} phase.

If not all the vertices in the s-t chain appear in the k^{th} layered network then let $v_a \xrightarrow{e_{a+1}} v_{a+1}$ be the first edge such that for some b, $v_a \in V_b$ but v_{a+1} is not in the k^{th} layered network. Thus, e_{a+1} was not used in the k^{th} phase. Since it is useful in the beginning of the $(k+1)^{st}$ phase, however, it was also useful in the beginning of the k^{th} phase. Thus the only possible reason for v_{a+1} not to belong to V_{b+1} is that $b+1=\ell_k$. Further, by the argument of the previous paragraph $a \geq b$. Therefore, $a+1 \geq \ell_k$ and so $\ell_{k+1} > \ell_k$.

□

Corollary 2.1:

Given any network $\mathcal{N}$ as input, the number of phases performed by Dinic's algorithm must be less than or equal to $|V|$.

Proof:

Any layered network constructed by Dinic's algorithm must contain no more than $|V|$ layers. Thus by Lemma 2.2 there can be at most $|V|$ phases.

□

We now notice that the time required during each phase of Dinic's algorithm, to construct the layered network and initialize the indegrees, outdegrees, and nullist is bounded by $O(|E|)$. Further, each flow augmentation requires time $O(|V|)$ and there can be at most $O(|E|)$ such

augmentations since each augmentation causes the deletion of at least one edge from the layered network. Finally, the total time required during each phase of Dinic's algorithm to delete edges, delete vertices, and update indegrees, outdegrees, and nullist is bounded by $O(|E|)$. This results from the fact that each edge and each vertex can be deleted from the layered network at most once. It should now be clear that each phase (iteration) of Dinic's algorithm has time complexity $O(|V|\cdot|E|)$. Thus by Corollary 2.1, we have that Dinic's algorithm will always terminate within time $O(|V|^2\cdot|E|)$. Therefore, Dinic's algorithm will correctly compute, in time $O(|V|^2\cdot|E|)$, a maximum flow function on any network given as input.

2.5 Karzanov

A.V. Karzanov [8,14] modified Dinic's algorithm in 1973 to obtain an $O(|V|^3)$ maximum network flow algorithm. Karzanov noticed that the layered network constructed during each phase of Dinic's algorithm could be used to <u>simultaneously</u> augment along all existing minimum length s-t flow augmenting paths. He then showed how this simultaneous augmentation could be performed in time $O(|V|^2)$. Karzanov's algorithm is the result of replacing Dinic's $O(|V|\cdot|E|)$ successive flow augmentation procedure with this new $O(|V|^2)$ simultaneous flow augmentation procedure.

Karzanov's results are based on the notion of a <u>maximal</u> flow function. A maximal flow function $\hat{f}$, on a network $\mathcal{N}$, is defined to be any legal flow function on $\mathcal{N}$ which has the property that every s-t chain in $\mathcal{N}$ contains at least one saturated edge (i.e. at least one edge e such that $\hat{f}(e)=c(e)$). From the following example it can be seen that a maximal flow

function on a network need not be a maximum flow function on that network.

Example 2.2:

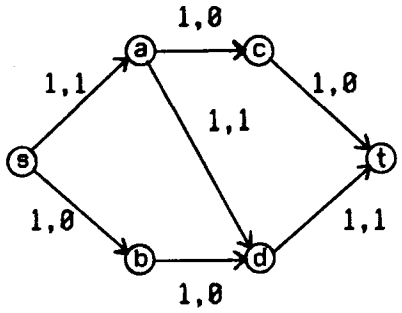
The following maximal flow function $\hat{f}_0$ is defined on the layered network LN_0 from Example 2.1:

$\hat{f}_0(s,a)=1$ $\hat{f}_0(s,b)=0$

$\hat{f}_0(a,c)=0$ $\hat{f}_0(a,d)=1$

$\hat{f}_0(b,d)=0$ $\hat{f}_0(c,t)=0$

$\hat{f}_0(d,t)=1$



$v(\hat{f}_0)=1$ is not maximum.

Karzanov noticed that the flow augmentation required during each phase of Dinic's algorithm could be achieved by simply augmenting the existing legal flow function f with any maximal flow function $\hat{f}$ on the current layered network. Once such a maximal flow function had been computed, the flow modification could be performed as follows:

$f'(e) \leftarrow f(e)+\hat{f}(e)$, for all forward edges in the layered network

and

$f'(e) \leftarrow f(e)-\hat{f}(e)$, for all reverse edges in the layered network.

It can easily be seen that performing the augmentation in this manner is essentially the same as simultaneously augmenting along all existing minimum length s-t flow augmenting paths, where each individual augmentation is performed exactly as outlined in the proof of Lemma 1.1. Karzanov used this clever augmentation technique to develop the following modification of Dinic's algorithm.

```
procedure KARZANOV (n):
    begin
    . initialize existing legal flow function f, on input network n, to 0;
    . while "true" do
    .    begin
    .    .  construct layered network, LN;
    .    .  compute maximal flow function f̂ on LN;
    .    .  for each edge e in n do
    .    .     if e is a forward edge in LN then f(e):=f(e)+f̂(e)
    .    .        else if e is a reverse edge in LN then f(e):=f(e)-f̂(e)
    .    end
    end.
```

Notice that Karzanov's algorithm, like Dinic's algorithm, terminates when the breadth first search performed in constructing each layered network fails to reach the sink. By the same argument as that used for Dinic's algorithm, however, the existing legal flow function must be maximum when this occurs. Therefore if Karzanov's algorithm halts, then the existing legal flow function must be a maximum flow function on the input network. Next we notice that the proof of Lemma 2.2 is valid for Karzanov's algorithm as well as Dinic's algorithm. Thus given any network n as input, the number of phases performed by Karzanov's algorithm must be less than or equal to $|V|$. It should also be clear, however, that the time required to construct each layered network and perform each flow modification is bounded by $O(|E|)$. Therefore if we let t denote the time required to compute each maximal flow function, then Karzanov's algorithm

is guaranteed to halt within time $O(|V|\cdot|E| + |V|\cdot\ell)$. We will now show how Karzanov's algorithm computes each maximal flow function in time $O(|V|^2)$ to yield an $O(|V|^3)$ max-flow algorithm.

Karzanov's algorithm computes each maximal flow function by successively improving an existing illegal flow function, called a preflow function, on the current layered network LN [8,10,14]. For each vertex v in LN, let In'(v) denote the set of all edges incoming to v in LN and let Out'(v) denote the set of all edges outgoing from v in LN. A preflow function $\tilde{f}$, on the layered network LN, associates with each edge e in LN a real number $\tilde{f}(e)$ satisfying the conditions:

C3) $0 \le \tilde{f}(e) \le \hat{c}(e)$, for each edge e in LN

C4) $\sum_{e \in In'(v)} \tilde{f}(e) \ge \sum_{e \in Out'(v)} \tilde{f}(e)$, for each vertex $v \ne s,t$ in LN.

Thus a preflow function is simply a flow function which satisfies the capacity constraint but not necessarily the conservation constraint of a legal flow function. For each vertex $v \ne s,t$ in LN, we define excess(v) to be the excess flow entering v:

$$\text{excess}(v) = \sum_{e \in In'(v)} \tilde{f}(e) - \sum_{e \in Out'(v)} \tilde{f}(e).$$

If excess(v)>0 then v is said to be unbalanced; otherwise v is said to be balanced. The source and the sink are always considered balanced.

Throughout the execution of Karzanov's maximal flow procedure, every edge in LN is declared either open or closed. Initially all edges are declared open. As the algorithm proceeds, however, some of the edges

will be declared closed. Once an edge is declared closed, the flow through it will remain unchanged to the end of the procedure.

Karzanov's maximal flow procedure alternates between pushing additional flow from unbalanced vertices and balancing the unbalanced vertices that are generated during these pushes. The pushing of flow is achieved through repeated calls to a procedure PUSH(i), with increasing i [10]. The procedure PUSH(i) considers in turn each unbalanced vertex in layer V_i, attempting to push flow from it to vertices in layer V_{i+1}. For each unbalanced vertex $v \in V_i$, the procedure considers in turn each open edge in Out'(v) and sends through it the maximum possible amount of flow. (The two constraints that exist are the current excess of v and the amount of flow needed to saturate the edge.) The push from v ends when either v becomes balanced or every edge in Out'(v) becomes either saturated (i.e. $\tilde{f}(e)=\hat{c}(e)$) or closed. For each vertex u in LN there is a <u>stack</u> (push-down store) on which the history of additions of incoming flow into u is recorded. When the flow in an edge e=(v,u) is incremented by an amount δ, the pair (v,δ) is added to the top of the stack for vertex u. The procedure PUSH(i) is said to be <u>successful</u> if flow is pushed to layer V_{i+1}. If PUSH(i) is successful then PUSH(i+1) is called, and so on.

A procedure BALANCE(i) is the tool through which unbalanced vertices become balanced. The procedure BALANCE(i) uses the stacks to shift back flow from vertices in layer V_i to vertices in layer V_{i-1}. It balances in turn each of the unbalanced vertices $v \in V_i$ by canceling the most recent additions of flow into v. Clearly the last canceled addition of flow into v may only be partial. After each unbalanced vertex v is

balanced, all the edges in $In'(v)$ are declared closed. The procedure BALANCE(i) is always followed by a call to PUSH(i-1).

Karzanov's complete maximal flow procedure is: (From [10])

```
procedure MAXIMAL (LN):
    begin
    .  initialize existing preflow function f~, on layered network LN, to 0;
    .  empty the stacks of all vertices in LN;
    .  i:=0;
PLOOP:  PUSH(i);
    .  while the previous push was successful and i+1<ℓ do
    .     begin
    .     .  i:=i+1;
    .     .  PUSH(i)
    .     end;
    .  if there exist unbalanced vertices in LN then
    .     begin
    .     .  i:=number of highest layer V_j (0<j<ℓ) containing
    .     .        unbalanced vertices;
    .     .  BALANCE(i);
    .     .  i:=i-1;
    .     .  goto PLOOP
    .     end;
    .  for each edge e in LN do f^(e):=f~(e)
    end.
```

It should now be clear that every vertex in the layered network LN will be balanced when Karzanov's maximal flow procedure halts. Thus the final existing preflow function $\tilde{f}$ will in fact be a legal flow function on LN. In order to show that it will also be a maximal flow function on LN, we must introduce the notion of a blocked vertex. A vertex v in LN is said to be blocked with respect to the existing preflow function if every v-t chain in LN contains at least one saturated edge. Notice that the source s becomes blocked after the first execution of PUSH(∅), since every edge in Out'(s) becomes saturated.

Lemma 2.3:

If a vertex in LN becomes blocked at some point in the execution of Karzanov's maximal flow procedure, then it remains blocked to the end of the procedure. (A proof of this lemma appears in [8].)

Lemma 2.4:

Every vertex in LN is balanced at most once throughout the execution of Karzanov's procedure. (A proof of this lemma appears in [8].)

We can now see that the final existing preflow function on LN will be a legal flow function which has the property that every s-t chain in LN contains at least one saturated edge. Therefore when Karzanov's maximal flow procedure halts, the existing preflow function $\tilde{f}$ will in fact be a maximal flow function on the layered network LN. It now remains to be shown that the procedure will always halt within $O(|V|^2)$ steps.

The total number of steps performed by Karzanov's maximal flow procedure is clearly bounded by the total number of flow additions and flow reductions performed. The number of flow reductions performed, however, is bounded by the number of flow additions performed since each vertex is balanced at most once and the history of flow additions in the stacks is used to perform the flow reductions. Thus it suffices to show that the number of flow additions performed by Karzanov's procedure is bounded by $O(|V|^2)$. We first notice that there can be at most one saturating flow addition per edge in the layered network. Since the number of edges in the layered network is bounded by the number of edges in the original input network, however, there can be at most $O(|E|)$ saturating flow additions. Next we notice that there can be at most one non-saturating flow addition per vertex in the layered network, between any two successive calls to BALANCE(i). (When flow is pushed from a vertex v in layer V_i, only the last edge considered in $Out'(v)$ does not necessarily become saturated.) Since the number of vertices in the layered network is bounded by the number of vertices in the original input network, however, there can be at most $O(|V|)$ non-saturating flow additions between any two successive calls to BALANCE(i). From Lemma 2.4, however, there can be at most $O(|V|)$ calls to BALANCE(i). Thus the total number of non-saturating flow additions is bounded by $O(|V|^2)$ and hence the total number of flow additions is bounded by $O(|V|^2)$.

It should now be clear that Karzanov's maximum network flow algorithm will correctly compute, in time $O(|V|\cdot|E| + |V|\cdot|V|^2) = O(|V|^3)$, a maximum flow function on any network given as input.

2.6 Further Improvements

We saw in the last section that any $O(t)$ algorithm for computing a maximal flow function on a layered network could be used to develop an $O(|V|\cdot|E| + |V|\cdot t)$ maximum network flow algorithm. (Simply modify Karzanov's max-flow algorithm by replacing his "push and balance" procedure with the new maximal flow procedure.) Notice, however, that any max-flow algorithm developed through this technique can be no faster than $O(|V|\cdot|E|)$. Despite this fact, each of the four most recent maximum network flow algorithms have been based on developing new maximal flow procedures. We shall now briefly describe each of these new maximal flow procedures.

In 1976 B.V. Cherkasky [10] showed how Karzanov's $O(|V|^2)$ push and balance routine could be modified to run in time $O(|V|\cdot|E|^{1/2})$. Cherkasky's procedure partitions the layered network into blocks of consecutive layers called superlayers. It then applies Karzanov's push and balance techniques to these superlayers. Within the superlayers, however, Dinic's flow augmentation techniques are used. The result is an asymptotically faster but very complex maximal flow procedure.

A little over a year later, Z. Galil [10] improved Cherkasky's routine to obtain a maximal flow procedure with time complexity $O(|V|^{2/3}\cdot|E|^{2/3})$. Galil's procedure differs from Cherkasky's procedure in the techniques used within the superlayers. Galil's routine maintains a special data structure containing information about the current "usefulness" of chains within the layered network. This data structure is

used to expedite the push of flow through edges within the superlayers. Like Cherkasky's routine, Galil's procedure is very complex and requires a great deal of overhead.

In 1978 a very simple $O(|V|^2)$ maximal flow procedure was developed by three Indians named V.M. Malhotra, M. Pramodh Kumar, and S.N. Maheshwari [16]. Their procedure is based on successively augmenting an existing legal flow function on the layered network. The procedure begins by determining the maximum amount of flow that can be pushed through each vertex v in the layered network. This value is called the flow potential $p_f(v)$ of the vertex v. Each flow augmentation is then performed in three steps. First a vertex with minimum non-zero flow potential over all vertices in the layered network is selected as the reference vertex, r. Next, $p_f(r)$ units of flow are pushed from r to t and from s to r. The pushing is performed essentially as outlined in Karzanov's algorithm. Finally, the procedure updates the flow potential of each vertex through which flow has been pushed, closing all edges which become either saturated or unreachable from s or t. Although this procedure is not an asymptotic improvement over Karzanov's push and balance routine, it is extremely simple and results in perhaps the best max-flow algorithm for use on dense networks ($|E| \approx |V|^2$).

Finally, Galil and A. Naamad [12] have recently developed an $O(|E| \cdot \log^2|V|)$ maximal flow procedure. This procedure is similar to Galil's $O(|V|^{2/3} \cdot |E|^{2/3})$ maximal flow procedure in that a data structure is maintained for processing chains within the layered network. The new algorithm, however, does not partition the layered network into

superlayers.

This concludes our discussion of algorithms for computing a maximum flow function on a network. We will now turn our attention to the question of lower bounds on the computational complexity of the maximum network flow problem.

CHAPTER 3 - LOWER BOUNDS ON MAX-FLOW

3.1 Introduction

Chapter 2 dealt with the establishment of upper bounds on the computational complexity of the maximum network flow problem. We traced the development of max-flow algorithms from the original labeling algorithm of Ford and Fulkerson [9], through the recent $O(|V|\cdot|E|\cdot\log^2|V|)$ algorithm of Galil and Naamad [12]. The very fact that the search for new max-flow algorithms has been so fruitful leads us to now ask the question, "Can we do better?". Can we develop a max-flow algorithm which is asymptotically faster than $O(|V|\cdot|E|\cdot\log^2|V|)$? Galil has shown [11,12] that any algorithm which uses Dinic's technique of dividing the problem into phases (as do all the known algorithms developed since Dinic's algorithm) must have time complexity at least $O(|V|\cdot|E|)$. Further, he has conjectured an $\Omega(|V|\cdot|E|)$ lower bound on the computational complexity of the maximum network flow problem. At this time, however, there is no known non-linear lower bound on max-flow.

The determination of lower bounds on the computational complexity of a problem is generally much more difficult than the establishment of upper bounds on the problem. In the latter case we can simply demonstrate an algorithm for solving the problem within the specified running time. In the former case, however, we must prove that <u>any</u> algorithm for solving the problem must require at least the specified running time, regardless of how clever the algorithm. In order to somewhat simplify the lower bound problem, many authors have chosen to work with restricted models of

computation. One such model which has received considerable attention in the recent literature is the linear decision tree model [6,18,21,22]. This model tends to underestimate total time complexity but nevertheless enables us to study non-trivial lower bounds. In this chapter we shall investigate one particular approach to establishing non-linear lower bounds on the computational complexity of the maximum network flow problem relative to the linear decision tree model of computation. The technique we shall deal with is the polyhedral technique developed by A.C. Yao, D.M. Avis and R.L. Rivest [21,22].

3.2 The Model of Computation

The linear decision tree model of computation is based on the notion of a linear decision tree algorithm. A linear decision tree algorithm, operating on input $(x_1,\ldots,x_n)$, is simply a finite ternary tree with each internal node representing a test of the form "$\sum \alpha_i \cdot x_i : z$" and each leaf containing a possible output. Given any input, the algorithm begins at the root and proceeds by moving down the tree until a leaf is reached. At each internal node the algorithm performs the specified test and then branches according to the result of this test ($<$, $=$, or $>$). Once a leaf is reached, the information contained in that leaf is output as the result of the computation and the algorithm halts. The time complexity of any such algorithm is simply defined to be the height of the corresponding tree.

The computational complexity of any problem relative to the linear decision tree model of computation can now be defined as the minimum height

over all decision trees which solve the problem. It should be clear that the linear decision tree model measures time complexity solely in terms of the number of comparisons and branchings required. Thus the model tends to underestimate the total time complexity of a problem. Despite this fact, the linear decision tree model has proven useful in establishing several non-trivial lower bounds.

3.3 Polyhedral Decision Problems

Let $P = \{\vec{x}\in\mathbb{R}^n \mid \ell_i(\vec{x})\leq 0 \text{ for each } i=1,2,\ldots,m\}$ be a set of points in $\mathbb{R}^n$, where $\vec{x}=(x_1,\ldots,x_n)$, m is an integer and

$$\ell_i(\vec{x}) = \sum_{j=1}^{n} \lambda_{ij}\cdot x_j$$

for real numbers λ_{ij}. The set P is said to be a <u>polyhedron</u> in $\mathbb{R}^n$. If d is the dimension of the smallest subspace of $\mathbb{R}^n$ containing P, then P is also said to be a polyhedron of dimension d. (Notice that we are restricting our attention to <u>homogeneous</u> polyhedra, i.e. cones.) On each subset H of the set $\{1,2,\ldots,m\}$, we define the set of points $F_H(P)\subseteq P$ as follows:

$$F_H(P) = \{\vec{x}\in\mathbb{R}^n \mid \ell_i(\vec{x})<0 \text{ for each } i\in H,$$
$$\ell_i(\vec{x})=0 \text{ for each } i\notin H\}.$$

The set $F_H(P)$ is called a <u>face</u> of the polyhedron P. If s is the dimension of the smallest subspace of $\mathbb{R}^n$ containing $F_H(P)$, then $F_H(P)$ is said to be a face of dimension s. (The empty face has dimension -1 by convention.) We shall let $F_s(P)$ denote the set of all faces of dimension s of P. Notice that every point in P lies on some face of P and that the intersection of

any two faces of P is empty.

The polyhedral decision problem B(P) can now be defined as the problem of determining whether an input point $\vec{x} \in \mathbb{R}^n$ lies in the polyhedron P (i.e. Given any input $\vec{x} \in \mathbb{R}^n$, is $\vec{x} \in P$?). A linear decision tree algorithm for solving this problem will be a decision tree which contains a "yes" or "no" decision at every leaf. The computational complexity of B(P) relative to the linear decision tree model of computation will be denoted by L(P).

In 1977 A.C. Yao, D.M. Avis and R.L. Rivest [21] proved the following fundamental theorem relating the complexity of the polyhedral decision problem B(P) to the facial character of the polyhedron P.

Theorem 3.1:

Let $P = \{\vec{x} \mid \ell_i(\vec{x}) \leq 0 \text{ for each } i=1,2,\ldots,m\}$ be a polyhedron in $\mathbb{R}^n$. Then for each s,

$$L(P) \geq 1/2 \log |F_s(P)|.$$

Proof:

The proof of Theorem 3.1 can be found in [21].

This theorem states that $\Omega(\log F_s)$ linear comparisons are necessary to determine if a point lies in a polyhedron composed of F_s s-dimensional faces. As a result, we can determine lower bounds on the computational complexity of any polyhedral decision problem relative to the linear decision tree model of computation by simply examining the facial structure of the polyhedron.

3.4 Applications to Max-Flow

In this section we shall consider a straightforward application of the concepts presented in the previous section to the problem of establishing a non-trivial lower bound on max-flow. We shall first introduce the class of polyhedral decision problems $\{B(P_n) \mid n \geq 2\}$, which is very closely related to the maximum network flow problem. We shall then formalize this relationship by showing that $L(P_n)-n^2-2$ is in fact a lower bound on L_n, where L_n is the linear decision tree complexity of the maximum network flow problem for a complete network on n vertices. Thus any lower bound on $L(P_n)$ will also yield a lower bound on L_n. Finally we shall prove the following three results concerning the facial structure of the polyhedra P_n $(n \geq 2)$:

1) There exists a positive constant c such that $|F_1(P_n)| \geq c \cdot (n-2)!$, for all $n \geq 2$.

2) There exists a positive constant c' such that $|F_1(P_n)| \leq c' \cdot (n-2)!$, for all $n \geq 2$.

3) There exists a positive constant c'' such that $|F_s(P_n)| \leq 2^{(c'' \cdot n^2)}$, for all s and for all $n \geq 2$.

Based on these results, we can then conclude that Theorem 3.1 cannot be directly applied to the class of problems $\{B(P_n) \mid n \geq 2\}$ to obtain a non-trivial lower bound on max-flow.

In order to formally define the class of polyhedral decision problems $\{B(P_n) \mid n \geq 2\}$, we must introduce some new notation. Let G_n denote the complete directed graph on $n \geq 2$ vertices in which the vertex v_1 is specified as s and the vertex v_n is specified as t. We can represent the sets of vertices and edges of G_n as follows:

$$V = \{v_1, v_2, v_3, \ldots, v_n\}$$

and

$$E = \{e_{ij} \mid 1 \leq i, j \leq n\}.$$

Notice that $|V|=n$, $|E|=n^2$ and we are defining e_{ij} to be the edge in G_n directed from vertex v_i to vertex v_j. It should be clear that any capacity function defined on the set E will give rise to a complete flow network on n vertices. Further, notice that the definition of a cut in a network can be directly applied to the graph G_n. We now let $\{X_1, X_2, \ldots, X_{2^{n-2}}\}$ represent the set of all cuts in G_n.

For our purposes, a vector $\vec{y} \in \mathbb{R}^{n^2+1}$ will be represented as $\vec{y} = (y_{11}, \ldots, y_{1n}, \ldots, y_{n1}, \ldots, y_{nn}, y_{n^2+1})$. The polyhedron P_n in $\mathbb{R}^{n^2+1}$ can now be defined as follows:

$$P_n = \{\vec{y} \in \mathbb{R}^{n^2+1} \mid y_{ij} \geq 0 \text{ for all } 1 \leq i, j \leq n,\ y_{n^2+1} \geq 0,\ \ell_k(\vec{y}) \geq 0 \text{ for all } 1 \leq k \leq 2^{n-2}\},$$

where

$$\ell_k(\vec{y}) = \left(\sum \{y_{ij} \mid e_{ij} \in (X_k; \bar{X}_k)\}\right) - y_{n^2+1}.$$

The polyhedral decision problem $B(P_n)$ is to determine whether an input point $\vec{y} \in \mathbb{R}^{n^2+1}$ belongs to the polyhedron P_n. If we think of the set $\{y_{11}, \ldots, y_{nn}\}$ as defining a capacity function on the graph G_n (i.e. $c(e_{ij}) = y_{ij}$), then it should be clear that a point $\vec{y} \in \mathbb{R}^{n^2+1}$ will belong to the polyhedron P_n if and only if the following two relations hold:

1) $y_{ij} \geq 0$, for all $1 \leq i, j \leq n$ (i.e. $c(e_{ij}) = y_{ij}$ defines a legal capacity function on G_n).

2) $0 \leq y_{n^2+1} \leq C(X_{min})$, where X_{min} is any minimum capacity cut on the network defined by G_n and $c(e_{ij}) = y_{ij}$.

By Theorem 1.3, however, we have that $C(X_{min})$ is equal to the value of any maximum flow function f_{max} on the network defined by G_n and $c(e_{ij}) = y_{ij}$.

Therefore, it should also be clear that a point $\vec{y}\in\mathbb{R}^{n^2+1}$ will belong to the polyhedron P_n if and only if $c(e_{ij})=y_{ij}$ defines a legal capacity function on G_n and there exists some legal flow function f on the network defined by G_n and $c(e_{ij})=y_{ij}$, such that $v(f)=y_{n^2+1}$ (i.e. $0\leq y_{n^2+1}\leq v(f_{max})$). The following lemma relates the linear decision tree complexity of $B(P_n)$ to the linear decision tree complexity of the max-flow problem on a complete network of n vertices.

Lemma 3.1:

$$L_n \geq L(P_n)-n^2-2.$$

Before presenting the proof of Lemma 3.1, we should consider the structure of any linear decision tree algorithm which computes a maximum flow function on a complete network on n vertices. Such an algorithm will be a ternary tree operating on input $(y_{11},\ldots,y_{nn})$, where the input defines a capacity function $c(e_{ij})=y_{ij}$ on the graph G_n. Each leaf in the tree will contain a set of n^2 linear functions $\{g_{ij} \mid 1\leq i,j\leq n\}$ defined on the n^2 input variables. For any input, the algorithm will begin at the root and proceed by moving down the tree until a leaf is reached. Once a leaf is reached, a maximum flow function f_{max} on the network defined by the graph G_n and the input $(y_{11},\ldots,y_{nn})$, will be given by $f_{max}(e_{ij}) = g_{ij}(y_{11},\ldots,y_{nn})$. We should note that on every network there will exist a maximum flow function that can be completely defined by a set of linear combinations of the edge capacities on the network.

Proof of Lemma 3.1:

Let T be any optimal linear decision tree algorithm for computing a maximum flow function on a complete network on n vertices. Clearly the height of T must be L_n. Further, we can obtain a linear decision tree T′ for the problem $B(P_n)$ by modifying T as follows. Place the root of T below a new sequence of n^2 distinct tests of the form "Is $y_{ij} \geq 0$?" such that the root of T is reached if and only if all of these new tests produce a "yes" answer. Then replace each leaf in T with a new test of the form "Is $y_{n^2+1} \geq 0$?" followed by a new test of the form "Is $v(f_{max})-y_{n^2+1} \geq 0$?" ($y_{n^2+1}$ being a new input). The new tree T′ should be constructed in such a way that if any of the newly added tests produce a "no" answer, then a leaf containing a "no" decision is reached. Otherwise a leaf containing a "yes" decision is reached. Since the value $v(f_{max})$ is simply a sum of the g_{ij} available at each leaf in T, it should be clear that T′ is in fact a linear decision tree algorithm for solving the problem $B(P_n)$. The height of T′, however, is L_n+n^2+2 and thus we have the relation:

$$L(P_n) \leq L_n+n^2+2$$

$$\Rightarrow \; L_n \geq L(P_n)-n^2-2.$$

□

The problem of establishing a non-trivial lower bound on the linear decision tree complexity of the maximum network flow problem has now been reduced to the problem of establishing a non-trivial lower bound on $L(P_n)$, for each $n \geq 2$. By Theorem 3.1, however, we can establish lower bounds on $L(P_n)$, for each $n \geq 2$, by simply examining the facial structure of the

polyhedra P_n. The remainder of this section will be devoted to proving several lemmas concerning the number of faces composing each of these polyhedra. We will essentially show that each polyhedron is composed of relatively few faces and thus Theorem 3.1 can be of no use in establishing a non-linear lower bound (if one exists) on any $L(P_n)$.

Lemma 3.2:

There exists a positive constant c such that $|F_1(P_n)| \geq c\cdot(n-2)!$, for all $n\geq 2$.

Proof:

Consider any $n\geq 2$ and let p' be an s-t chain in the graph G_n. Now consider the point $\vec{y}'\in\mathbb{R}^{n^2+1}$ with the properties:

1) $y'_{ij}=1$ if $e_{ij}\in p'$, for all $1\leq i,j\leq n$

2) $y'_{ij}=0$ if $e_{ij}\notin p'$, for all $1\leq i,j\leq n$

3) $y'_{n^2+1}=1$.

Clearly the point $\vec{y}'$ belongs to the polyhedron P_n and so there must exist some face of P_n containing $\vec{y}'$ (since every point in a polyhedron lies on some face of that polyhedron). Let $F_{\vec{y}'}(P_n)$ denote the face of P_n which contains the point $\vec{y}'$. Therefore, $F_{\vec{y}'}(P_n)$ is the set of all points $\vec{y}''\in P_n$ which satisfy:

(3.1) $\quad y''_{ij}=0$ iff $y'_{ij}=0$, for all $1\leq i,j\leq n$

(3.2) $\quad y''_{n^2+1}>0$

(3.3) $\quad \ell_k(\vec{y}'')=0$ iff $\ell_k(\vec{y}')=0$, for all $1\leq k\leq 2^{n-2}$.

From (3.1) and (3.2) it should be clear that every point $\vec{y}''\in F_{\vec{y}'}(P_n)$ will be non-zero in exactly the same co-ordinates. Further, applying this

observation to (3.3) and the fact that p' is an s-t chain in G_n, we must have that $y''_{ij}=y''_{n^2+1}$ for each non-zero y''_{ij} $(1\leq i,j\leq n)$. Thus it must be the case that every point in the set $F_{\vec{y}'}(P_n)$ is simply a scalar multiple of the point $\vec{y}'$. It should now be clear that the smallest subspace of $\mathbb{R}^{n^2+1}$ containing the set $F_{\vec{y}'}(P_n)$ has dimension 1. Thus we have that $F_{\vec{y}'}(P_n)\in F_1(P_n)$. Finally, notice that each distinct s-t chain in G_n will give rise to a distinct 1-dimensional face of P_n. Therefore, if Z denotes the number of distinct s-t chains in the graph G_n then we must have:

$$|F_1(P_n)| \geq Z. \tag{3.4}$$

Since G_n is a complete graph on n vertices, however, the value Z can be expressed by the following formula:

$$Z = \sum_{i=1}^{n-1} (n-2)!/(i-1)! \qquad (0! = 1)$$

$$= (n-2)!\cdot\sum_{i=0}^{n-2} 1/i!$$

$$\geq (n-2)!$$

Combining this expression with (3.4) we obtain:

$$|F_1(P_n)| \geq (n-2)!$$

□

Lemma 3.3:

There exists a positive constant c' such that $|F_1(P_n)| \leq c'\cdot(n-2)!$, for all $n\geq 2$.

Proof:

Consider any $n\geq 2$ and let $F'_1(P_n)$ denote the set of all 1-dimensional

faces of the polyhedron P_n which arise from distinct s-t chains in the graph G_n, as outlined in the proof of Lemma 3.2. Now let Q denote the set of all points $\vec{y}' \in P_n$ which satisfy any one of the following three conditions:

1) $\vec{y}' = \vec{0}$

2) $y'_{n^2+1} = 0$, $y'_{1n} = 0$ and $\vec{y}'$ has exactly one non-zero co-ordinate

3) $\vec{y}'$ belongs to a face in the set $F'_1(P_n)$.

First notice that each point in Q which satisfies condition (2) belongs to one of exactly n^2-1 distinct (trivial) 1-dimensional faces of P_n. Further, notice that none of these n^2-1 faces belongs to the set $F'_1(P_n)$. Thus each point in Q, except the point $\vec{0}$, belongs to one of <u>exactly</u> $|F'_1(P_n)|+n^2-1$ distinct 1-dimensional faces of P_n. Next observe that every point in the polyhedron P_n can be expressed as a convex combination of points in Q (consider expressing a network as a sum of distinct s-t chains and isolated edges). Therefore, every point on a 1-dimensional face of P_n can be expressed as a convex combination of points in Q. Thus we must have that the set Q contains at least one point belonging to each 1-dimensional face of P_n and hence:

$$
\begin{aligned}
|F_1(P_n)| &\leq |F'_1(P_n)| + n^2 - 1 \\
&= ((n-2)! \cdot \sum_{i=0}^{n-2} 1/i!) + n^2 - 1 \\
&\leq 3 \cdot (n-2)! + n^2 - 1 \\
&\leq c' \cdot (n-2)! \qquad \text{(for some positive constant } c', \text{ independent of } n)
\end{aligned}
$$

□

Lemma 3.4:

There exists a positive constant c'' such that $|F_s(P_n)| \leq 2^{(c'' \cdot n^2)}$, for all s and for all $n \geq 2$.

Proof:

Consider any $n \geq 2$ and let each vector $\vec{w} \in \mathbb{R}^{2n^2}$ be represented as $\vec{w} = (w_{11}, \ldots, w_{nn}, \bar{w}_{11}, \ldots, \bar{w}_{nn})$. A polyhedron P_n' in $\mathbb{R}^{2n^2}$ can now be defined relative to the graph G_n as follows:

$$P_n' = \{\vec{w} \in \mathbb{R}^{2n^2} \mid w_{ij} \geq 0 \text{ and } \bar{w}_{ij} \geq 0 \text{ and } w_{ij} \geq \bar{w}_{ij} \text{ for all } 1 \leq i, j \leq n,$$
$$\ell_q'(\vec{w}) \geq 0 \text{ and } \ell_q'(\vec{w}) \leq 0 \text{ for all } 2 \leq q \leq n-1,$$
$$\ell_k''(\vec{w}) \geq 0 \text{ for all } 1 \leq k \leq 2^{n-2}\},$$

where

$$\ell_q'(\vec{w}) = \sum \{\bar{w}_{ij} \mid e_{ij} \in In(v_q)\} - \sum \{\bar{w}_{ij} \mid e_{ij} \in Out(v_q)\}$$

and

$$\ell_k''(\vec{w}) = \sum \{w_{ij} \mid e_{ij} \in (X_k; \bar{X}_k)\} - \sum \{\bar{w}_{ij} \mid e_{ij} \in In(t)\}.$$

If we think of the set $\{w_{11}, \ldots, w_{nn}\}$ as defining a capacity function on the graph G_n (i.e. $c(e_{ij}) = w_{ij}$) and the set $\{\bar{w}_{11}, \ldots, \bar{w}_{nn}\}$ as defining a flow function on the network defined by G_n and $c(e_{ij}) = w_{ij}$ (i.e. $f(e_{ij}) = \bar{w}_{ij}$), then it should be clear that a point $\vec{w} \in \mathbb{R}^{2n^2}$ will belong to the polyhedron P_n' if and only if the following two conditions hold:

1) $c(e_{ij}) = w_{ij}$ defines a legal capacity function on the graph G_n (i.e. $c(e) \geq 0$ for each $e \in E$).

2) $f(e_{ij}) = \bar{w}_{ij}$ defines a legal flow function on the network defined by G_n and $c(e_{ij}) = w_{ij}$.

Notice that the constraints $\ell_k''(\vec{w}) \geq 0$ are all redundant since any legal flow function f on a network $\mathcal{N}$ must always satisfy the relation $v(f) \leq C(X_{min})$,

where X_{min} is any minimum capacity cut on $\mathcal{N}$. It should now be clear that we can determine whether a point $\vec{w}\in\mathbb{R}^{2n^2}$ belongs to the polyhedron P_n' by simply testing to see if $\vec{w}$ satisfies each of the first $3n^2+2n-4$ constraints defining P_n'. Thus there exists a straightforward linear decision tree algorithm of height $3n^2+2n-4$ for solving the problem $B(P_n')$ and hence we have the relation:

$$L(P_n') \leq 3n^2+2n-4 < 4n^2.$$

Further, combining this relation with Theorem 3.1 we obtain the result:

$$4n^2 > L(P_n') \geq 1/2 \log |F_s(P_n')|, \text{ for all } s$$

$$\Rightarrow 2^{8n^2} > |F_s(P_n')|, \text{ for all } s. \tag{3.5}$$

If we now let $F(P_n') = \bigcup_s F_s(P_n')$ denote the set of all faces of the polyhedron P_n', then by (3.5) we have:

$$|F(P_n')| \leq \sum_{-1\leq s\leq 2n^2} 2^{8n^2}$$

$$\Rightarrow |F(P_n')| \leq (2n^2+2)\cdot 2^{8n^2}$$

$$\Rightarrow |F(P_n')| \leq 2^{(c''\cdot n^2)} \tag{3.6}$$

(for some positive constant c'', independent of n).

Turning our attention back to the polyhedron P_n, let $F(P_n) = \bigcup_s F_s(P_n)$ denote the set of all faces of the polyhedron P_n. The remainder of this proof will essentially consist of constructing a 1-1 mapping ψ from the elements of $F(P_n)$ into the elements of $F(P_n')$. We will then have that $|F(P_n)| \leq |F(P_n')|$.

Let $F_H(P_n)\in F(P_n)$ be any face of the polyhedron P_n. If $F_H(P_n)=\phi$ (i.e. $F_H(P_n)$ is the empty face) then define $\psi(F_H(P_n))=\phi$. If, however, $F_H(P_n)\neq\phi$ then let $\vec{y}'\in\mathbb{R}^{n^2+1}$ be some point in the set $F_H(P_n)$. Since $\vec{y}'\in F_H(P_n)\subseteq P_n$, we must have that $c(e_{ij})=y'_{ij}$ defines a legal capacity

function on the graph G_n and that there exists some legal flow function f' on the network defined by G_n and $c(e_{ij})=y'_{ij}$, such that $v(f')=y'_{n^2+1}$. Now consider the point $\vec{w}'\in\mathbb{R}^{2n^2}$ such that:

$$w'_{ij}=y'_{ij}, \text{ for all } 1\le i,j\le n$$

and

$$\bar{w}'_{ij}=f'(e_{ij}), \text{ for all } 1\le i,j\le n.$$

Clearly $\vec{w}'\in P'_n$ and thus there must exist some face of P'_n containing $\vec{w}'$. Call this face $F_{\vec{w}'}(P'_n)$ and define $\psi(F_H(P_n))=F_{\vec{w}'}(P'_n)$. It should now be clear that ψ maps each face in $F(P_n)$ into a face in $F(P'_n)$. Thus it remains to be shown only that ψ is a 1-1 mapping. Recall, however, that each face of a polyhedron is uniquely determined by the set of constraints its elements satisfy by equality. Further, notice that the points $\vec{w}'$ and $\vec{y}'$ have the following relationship:

1) $w'_{ij}=0$ iff $y'_{ij}=0$, for all $1\le i,j\le n$

2) $\ell''_k(\vec{w}')=0$ iff $\ell_k(\vec{y}')=0$, for all $1\le k\le 2^{n-2}$

3) $\bar{w}'_{ij}=0$ for all $\{ij \mid e_{ij}\in In(t)\}$ iff $y'_{n^2+1}=0$.

Thus for each distinct face $F_H(P_n)$ of the polyhedron P_n, $\psi(F_H(P_n))$ will be a distinct face of the polyhedron P'_n and so ψ is in fact a 1-1 mapping. We can now conclude that $|F(P_n)| \le |F(P'_n)|$ and thus by (3.6) we have:

$$|F(P_n)| \le 2^{(c''\cdot n^2)}. \tag{3.7}$$

Finally it should be clear that $|F_s(P_n)| \le |F(P_n)|$, for all s, and so by (3.7):

$$|F_s(P_n)| \le 2^{(c''\cdot n^2)}, \text{ for all } s.$$

□

3.5 Conclusions

In this chapter we presented the linear decision tree model of computation [6,18,21,22], the notion of a polyhedron and a polyhedral decision problem [21,22], and the class of polyhedral decision problems $\{B(P_n) \mid n \geq 2\}$ which most naturally arises when considering the maximum network flow problem. We then showed that the problem of establishing a non-linear lower bound on the linear decision tree complexity of max-flow can be reduced to the problem of establishing a non-linear lower bound on $L(P_n)$, for each $n \geq 2$. Next we demonstrated matching upper and lower bounds on the number of faces of dimension 1 composing each of the polyhedra P_n $(n \geq 2)$. Finally, we established a $2^{O(n^2)}$ upper bound on the number of faces, of any dimension, composing each of the polyhedra P_n. Based on our results, we can now conclude that Theorem 3.1 can be of no use in establishing a non-linear lower bound on max-flow, through the class of polyhedral decision problems $\{B(P_n) \mid n \geq 2\}$. It remains an open question, however, whether or not the polyhedral technique can in general be useful for establishing non-linear lower bounds on max-flow. There may, for example, exist some more complex class of polyhedral decision problems that can be reduced to max-flow.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[2] D.M. Avis, "Some Polyhedral Cones Related to Metric Spaces," Ph.D. Thesis, Department of Operations Research, Stanford University, 1977.

[3] A.E. Baratz, "Construction and Analysis of Network Flow Problem which Forces Karzanov Algorithm to $O(n^3)$ Running Time," M.I.T. Laboratory for Computer Science Technical Memo, LCS/TM-83, 1977.

[4] G.B. Dantzig, "Maximization of a Linear Function of Variables Subject to Linear Inequalities," in T.C. Koopmans (ed.), Activity Analysis of Production and Allocation, John Wiley & Sons, 1951, 339-347.

[5] E.A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation," Soviet Math. Dokl., Vol. 11, 1970, 1277-1280.

[6] D.P. Dobkin, R.J. Lipton and S.P. Reiss, "Excursions Into Geometry," Yale University Research Report #71.

[7] J. Edmonds and R.M. Karp, "Theoretical Improvements in Algorithm Efficiency for Network Flow Problems," JACM, Vol. 19, No. 2, 1972, 248-264.

[8] S. Even, "The Max Flow Algorithm of Dinic and Karzanov," M.I.T. Laboratory for Computer Science Technical Memo, LCS/TM-80, 1976.

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

[10] Z. Galil, "A New Algorithm for the Maximal Flow Problem," Proceedings 19th IEEE Symposium on Foundations of Computer Science, 1978, 231-245.

[11] Z. Galil, "On the Theoretical Efficiency of Various Network Flow Algorithms," IBM report, RC7320, 1978.

[12] Z. Galil and A. Naamad, "Network Flow and Generalized Path Compression," The 11th Annual ACM Symposium on Theory of Computing, 1979, 13-26.

[13] T.C. Hu, Integer Programming and Network Flows, Addison-Wesley, 1969.

[14] A.V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows," Soviet Math. Dokl., Vol. 15, 1974, 434-437.

[15] E. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, 1976.

[16] V.M. Malhotra, M. Pramodh Kumar and S.N. Maheshwari, "An $O(V^3)$ Algorithm for Finding Maximum Flows in Networks," Information Processing Letters, Vol. 7, No. 6, 1978, 277-278.

[17] P. McMullen and G.C. Shephard, Convex Polytopes and the Upper Bound Conjecture, Cambridge University Press, 1971.

[18] M.O. Rabin, "Proving Simultaneous Positivity of Linear Forms," JCSS, Vol. 6, 1972, 639-650.

[19] R.P. Stanley, "The Upper Bound Conjecture and Cohen-Macaulay Rings," Studies in Applied Mathematics, Vol. 54, No. 2, 1975.

[20] R. Tarjan, "Testing Graph Connectivity," The 6^{th} Annual ACM Symposium on Theory of Computing, 1974, 185-193.

[21] A.C. Yao, D.M. Avis and R.L. Rivest, "An $\Omega(n^2 \log n)$ Lower Bound to the Shortest Paths Problem," The 9^{th} Annual ACM Symposium on Theory of Computing, 1977, 11-17.

[22] A.C. Yao and R.L. Rivest, "On The Polyhedral Decision Problem," to appear in SIAM Journal on Computing.

[23] N. Zadeh, "Theoretical Efficiency of the Edmonds-Karp Algorithm for Computing Maximal Flows," JACM, Vol. 19, No. 1, 1972, 184-192.