MIT/LCS/TM-83

CONSTRUCTION AND ANALYSIS OF NETWORK FLOW PROBLEM WHICH FORCES KARZANOV ALGORITHM TO $O(n^3)$ RUNNING TIME

Alan E. Baratz

May 1977

CONSTRUCTION AND ANALYSIS OF NETWORK FLOW PROBLEM*
WHICH FORCES KARZANOV ALGORITHM
TO $O(n^3)$ RUNNING TIME

Alan E. Baratz
Department of Electrical Engineering
and Computer Science
MIT
Cambridge, Massachusetts

Abstract: The intent of this paper is to demonstrate the construction of a network flow problem which will force the Karzanov "Preflow" algorithm to run in its theoretic worst case time $O(n^3)$. Once such a "bad case" network has been constructed, an analysis is performed to determine the exact time required by the algorithm to compute the maximum flow through the network.

* This research was partial supported by NSF Grant MCS76-14294

Review of Karzanov Max Network Flow Algorithm

The following introduction to network flow problems and discussion of the Karzanov "Preflow" algorithm was extracted from a paper by Shimon Even entitled, "The Max Flow Algorithm of Dinic and Karzanov":

I. Introduction

A network consists of the following data:

1) A directed finite graph $G(V,E)$. Let n be the number of vertices $(=|V|)$.
2) Two vertices s and t are specified; s is the source and t is the sink.
3) Each edge $e \varepsilon E$ is assigned a positive real number $c(e)$ called the capacity of e.

A legal flow function f is an assignment of a real number $f(e)$ to e which satisfies two conditions:

c1) For every edge $e \varepsilon E$, $0 \leqslant f(e) \leqslant c(e)$
c2) For every vertex $v \varepsilon V$ let $\alpha(v)$ and $\beta(v)$ be the sets of edges incoming to v and outgoing from v respectively. For every vertex $v \neq s,t$

$$\sum_{e \varepsilon \alpha(v)} f(e) = \sum_{e \varepsilon \beta(v)} f(e) \qquad (1)$$

The total flow F of f is defined by

$$F = \sum_{e \varepsilon \alpha(t)} f(e) - \sum_{e \varepsilon \beta(t)} f(e) \qquad (2)$$

II. Maximum Flow Through Maximal Flows in Layered Networks

As in previous algorithms [this] algorithm describes how to improve existing legal flow, and when no improvement is possible the algorithm halts.

If we have a legal present flow f, an edge e, connecting vertices u and v, can be used to transfer flow from u to v in any one of the following two cases:

1) $u \overset{e}{\rightarrow} v$ (e is directed from u to v) and $f(e)<c(e)$.
2) $u \overset{e}{\leftarrow} v$ (e is directed from v to u) and $f(e)>0$.

(Clearly an edge can be used to both transfer flow from u to v and from v to u if $0<f(e)<c(e)$). We say that e is <u>useful from u to v</u> if (1) or (2) hold.

The layered network of G(V,E) with flow f is defined by the following procedure:

1) $V_o=\{s\}$ and set $i=0$
2) Construct $T=\{v \mid v \notin V_j$ for $j\leq i$ and there is a useful edge from a vertex of V_i to $v\}$
3) If T is empty, the existing F is maximum, halt.
4) If T contains t then $\ell=i+1$, $V_\ell=\{t\}$ and halt.
5) Let $V_{i+1}=T$, increment i and return to (2).

For every $0<i\leq\ell$, E_i is the set of edges useful from a vertex of V_{i-1} to a vertex of V_i.

For every edge e in E_j let $\tilde{c}(e)$ be defined as follows:

1) If $u\in V_{j-1}$, $v\in V_j$ and $u \overset{e}{\rightarrow} v$ then $\tilde{c}(e)=c(e)-f(e)$.
2) If $u\in V_{j-1}$, $v\in V_j$ and $u \overset{e}{\leftarrow} v$ then $\tilde{c}(e)=f(e)$.

We now consider all edges of E_j to be directed from V_{j-1} to V_j, even if in G(V,E) they may have the opposite direction (in case (2)). Also, the initial flow in the new network is $\tilde{f}(e)=0$ everywhere. We seek a maximal flow $\tilde{f}$ in

the layered network; by a maximal flow $\tilde{f}$ we mean that $\tilde{f}$ satisfies the condition that for every path

$$s \overset{e_1}{\rightarrow} v_1 \overset{e_2}{\rightarrow} v_2 \rightarrow \ldots v_{\ell-1} \overset{e_\ell}{\rightarrow} t,$$

where $v_j \varepsilon V_j$ and $e_j \varepsilon E_j$ there is at least one edge e_j such that $\tilde{f}(e_j)=\tilde{c}(e_j)$.

In the next section we shall describe how one can find a maximal flow function $\tilde{f}$ efficiently. For now, let us assume that such a flow function has been found and its total value is $\tilde{F}$. The flow f in G(V,E) is changed into f' as follows:

1) If $u \overset{e}{\rightarrow} v$, $u \varepsilon V_{j-1}$ and $v \varepsilon V_j$ then $f'(e)=f(e)+\tilde{f}(e)$.
2) If $u \overset{e}{\leftarrow} v$, $u \varepsilon V_{j-1}$ and $v \varepsilon V_j$ then $f'(e)=f(e)-\tilde{f}(e)$.

It is easy to see that the new flow f' satisfies both c1 (due to the choice of $\tilde{c}$) and c2 (because it is the superposition of two flows which satisfy c2). Clearly $F'=F+\tilde{F}>F$.

Let us call the part of the algorithm which starts with f, finds its layered network, finds a maximal flow $\tilde{f}$ in it and improves the flow in the original network to become f' -- a <u>phase</u>.

III. <u>Construction of a Maximal Flow in a Layered Network</u>

Karazanov's method uses in intermediate steps illegal flow functions which he calls preflow. A <u>preflow</u> function $\tilde{f}$ satisfies c1; that is, for every e in the layered network $0 \leqslant \tilde{f}(e) \leqslant \tilde{c}(e)$. However, it may not satisfy c2. Instead it satisfies the following weaker condition:

(c3) For every v in the layered network let $\tilde{\alpha}(v)$ and $\tilde{\beta}(v)$ be the sets of edges incoming to v and outgoing from v in the layered network, respectively. For every $v \neq s,t$

$$\sum_{e\in\tilde{\alpha}(v)} \tilde{f}(e) \geq \sum_{e\in\tilde{\beta}(v)} \tilde{f}(e). \qquad (5)$$

The algorithm uses two procedures alternately. They are called advance and balance.

The advance procedure will push forward additional preflow. It will start with vertices of some V_j; at that time no forwarding of preflow will be possible for vertices of lower layers. If preflow is pushed into some vertices of V_{j+1}, we shall try to push the preflow further, and the procedure will stop only when all the vertices into which preflow has been pushed have been processed. The vertices will be considered by moving from one layer to the next after all the pertinent vertices in the layer have been processed. The order by which the vertices in one layer are processed is irrelevant. However, for each vertex v the edges in $\tilde{\beta}(v)$ are ordered in some fixed order: $\tilde{\beta}(v)=\{e_{v1}, e_{v2},\ldots,e_{vd}\}$, where d is the outgoing degree of v in the layered network . Initially all the edges of the layered network are open, but as the algorithm proceeds, some edges will be declared closed, and the flow through them will remain unchanged to the end of the algorithm. Once the advance chooses an unbalanced vertex, (one for which (5) is satisfied with inequality), it first pushes flow through e_{v1} if it is open, next through e_{v2} if it is open, etc. It advances through the picked edge e as much as possible. If e becomes saturated (i.e., $\tilde{f}(e)=\tilde{c}(e)$) it moves to the next open edge. If the vertex v becomes balanced, the advance from v ends. Also, if all the edges of $\tilde{\beta}(v)$ are either closed or saturated, the advance from v ends, even though v may still be unbalanced.

For every vertex v there is a stack (push-down store) on which we record the additions of incoming flow into v. Each addition is an ordered pair of an edge $e\in\tilde{\alpha}(v)$ and a positive real number r. It specifies through which edge the additional incoming of r units of flow is produced.

Also, for each vertex we keep in two registers the sum of the incoming flow (which is equal to the sum of the r's stored in the stack) and the sum of the outgoing flow. Clearly, if these two registers contain the same number then the vertex is balanced.

The balancing procedure is the tool through which unbalanced vertices become balanced. It is applied to all vertices on one layer, and this layer is chosen to be the highest which contains unbalanced vertices. We balance a vertex v by canceling most recent additions, and as many of them as necessary so that the incoming flow will equal the outgoing flow. Clearly, the last canceled addition may be only partial, if only a part of the quantity specified by it must be canceled. In this case we restore the corrected addition (same edge, reduced quantity) in the stack. After an unbalanced vertex v is balanced, all edges of $\tilde{\alpha}(v)$ are declared closed.

Algorithm K:

1) Assign zero flow to all edges and all vertex flow registers. Empty the stacks of all the vertices.
2) $i \leftarrow 0$
3) Perform advanced starting from V_i.
4) If there are no unbalanced vertices other than s and t, halt: the present preflow is a maximal flow.
5) Let V_j $(0<j<\ell)$ be the highest layer which contains unbalanced vertices. Perform balancing for the unbalanced vertices in V_j.
6) $i \leftarrow j-1$ and go to (3).

A vertex is called blocked if every directed path from it to t contains at least one saturated edge. Clearly,

s becomes blocked on the first application of Step (3), since all the edges in $\overrightarrow{\beta}(s)$ become saturated.[1]

We shall finally include the modification to the Karzanov "Preflow" algorithm which eliminates from each layered network all dead-end vertices (i.e., vertices $v \neq s,t$ with INDEGREE (v)=0 or OUTDEGREE (v)=0). This modification can be implemented by adding the following steps to the algorithm immediately following the creation of a layered network:

```
begin
    for each vertex v in the layered network do begin
        calculate INDEGREE (v)
        calculate OUTDEGREE (v)
        if ((INDEGREE (v)=0) or (OUTDEGREE (v)=0)) and
            (v≠s,t) do begin
                push v onto stack, s
        end
    end
    while stack, s ≠ empty do begin
        pop vertex v off of stack
        remove vertex v from layered network
        for each vertex v' which is an immediate
            predecessor of v or an immediate successor
            of v do begin
                if v' is an immediate predecessor of v,
                    then adjust OUTDEGREE (v')
                else adjust INDEGREE (v')
                if ((INDEGREE (v')=0) or (OUTDEGREE (v')=0))
                    and (v'≠s,t) do begin
                        push v' on stack, s
                end
        end
    end
end
```

Construction of Network Requiring $0(n^3)$ Running Time

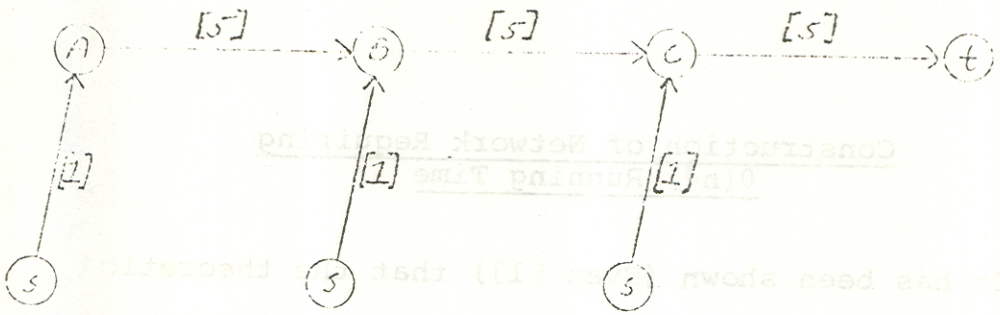
It has been shown (Even [1]) that the theoretic worst case running time for the Karzanov "Preflow" algorithm is $0(n^3)$. This bound, however, results from the fact that any given network flow problem can require the algorithm to iterate through at most $0(n)$ phases with at most $0(n^2)$ edge advances required during each phase to find a maximal flow through the layered network.

The construction of a network which will force the algorithm to properly iterate through $0(n)$ phases is fairly simple. The general technique is to design the flow network with $0(n)$ s-t (source-sink) paths of increasing length as shown in figure 1. The edge capacities for the network are then chosen in such a way that the completion of each phase results in the saturation of the edge (e_1) outgoing from the source in the layered network while leaving the remaining edges unsaturated (see figure 2).

The problem of forcing each of the phases to require $0(n^2)$ edge advances, however, is somewhat more difficult. We must design the flow network in such a way that during each phase a layered network is constructed which has the following two properties:

1) The balance procedure (Step (5) in algorithm K) must be performed on the network $0(n')$ separate

NETWORK FLOW PROBLEM WHICH FORCES 0(n) PHASES

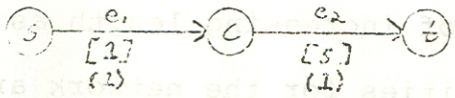


[ ] corresponds to edge capacity

Figure 1

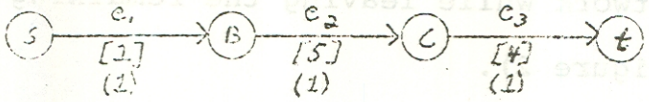
LAYERED NETWORKS CONSTRUCTED DURING SOLUTION PHASES
(Dead-End Vertices Have Been Removed
From Layered Networks)

Phase 1:



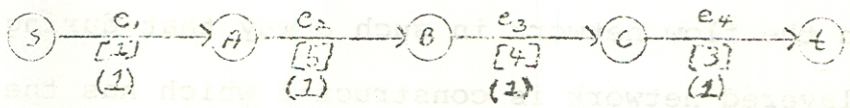
edge e_1 becomes saturated

Phase 2:



edge e_1 becomes saturated

Phase 3:



edge e_1 becomes saturated

( ) corresponds to actual flow through edge

[ ] corresponds to edge capacity

Figure 2

times where n' is the number of vertices in the layered network.

2) Following the execution of each balance procedure, new flow advance must occur along edges emanating from at least $O(n'-b)$ distinct vertices where b is the number of vertices in layered network which lie in layers below the layer containing the most recently balanced vertex.

An example of a general flow network which has been constructed to meet each of the requirements listed above and thus in fact forces the Karzanov "Preflow" algorithm to run in its theoretic worst case time $O(n^3)$ is shown in figure 3. All upper and lower horizontally oriented edges directed from left to right (labeled e_h in figure 3) are assigned capacities equal to nc+nx (upper) or nc (lower) to insure that they never become saturated. The edges directed into the sink (labeled e_t in figure 3), on the other hand, are assigned a capacity of either c or x (with the exception of the one edge assigned a capacity of 2c) in order that during each phase (excluding phase 1), the edges ingoing to the sink become saturated by the flow c+x outgoing from the source. Finally, the remaining vertically oriented edges (labeled e_v in figure 3) are assigned capacities $\frac{c}{n} + x$ so that during each successive phase, flow will initially be pushed along every edge in the new layered network.

GENERAL "BAD" CASE NETWORK FLOW PROBLEM
(All vertices labeled (s) correspond to the source)

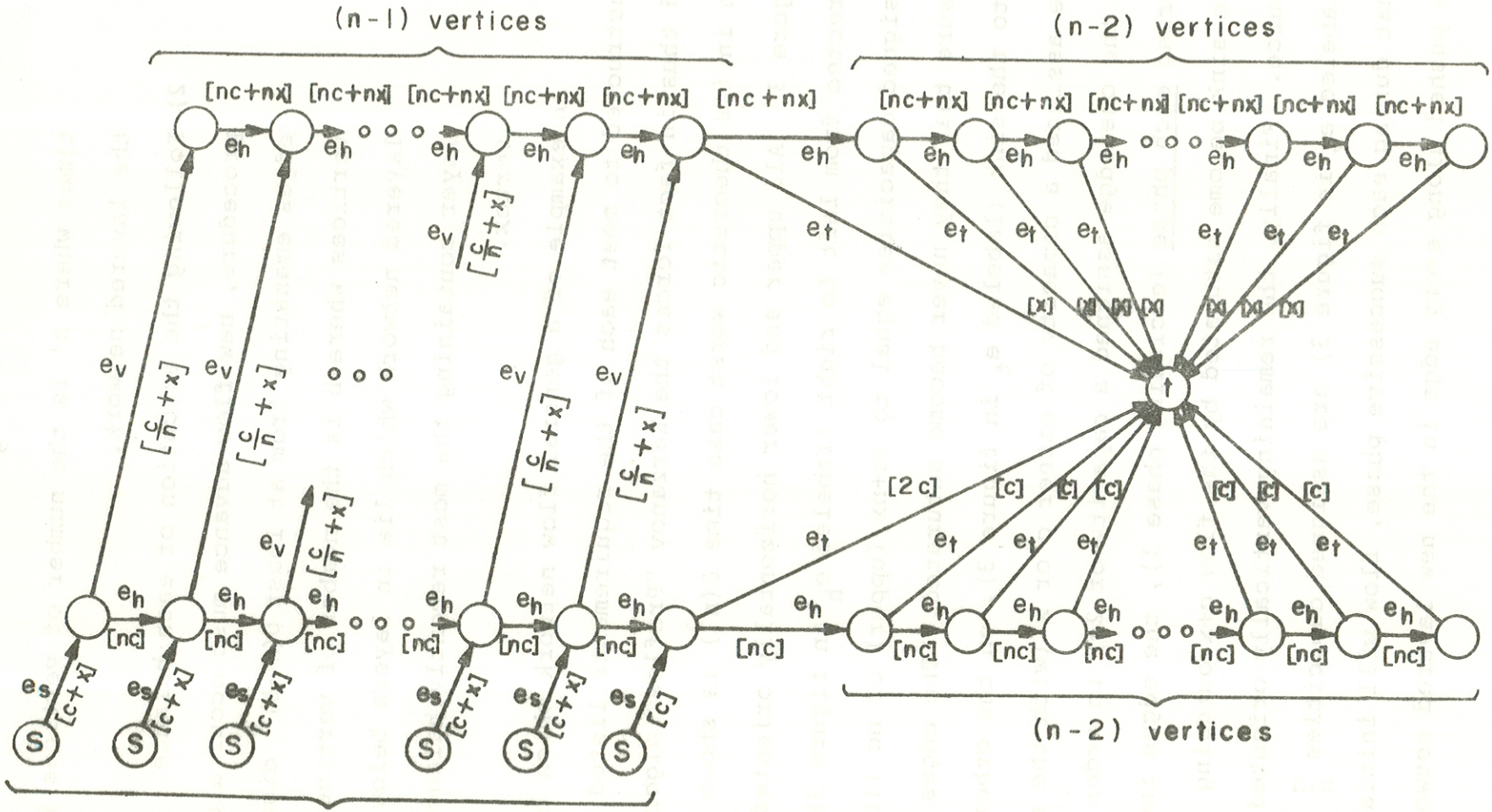


Figure 3

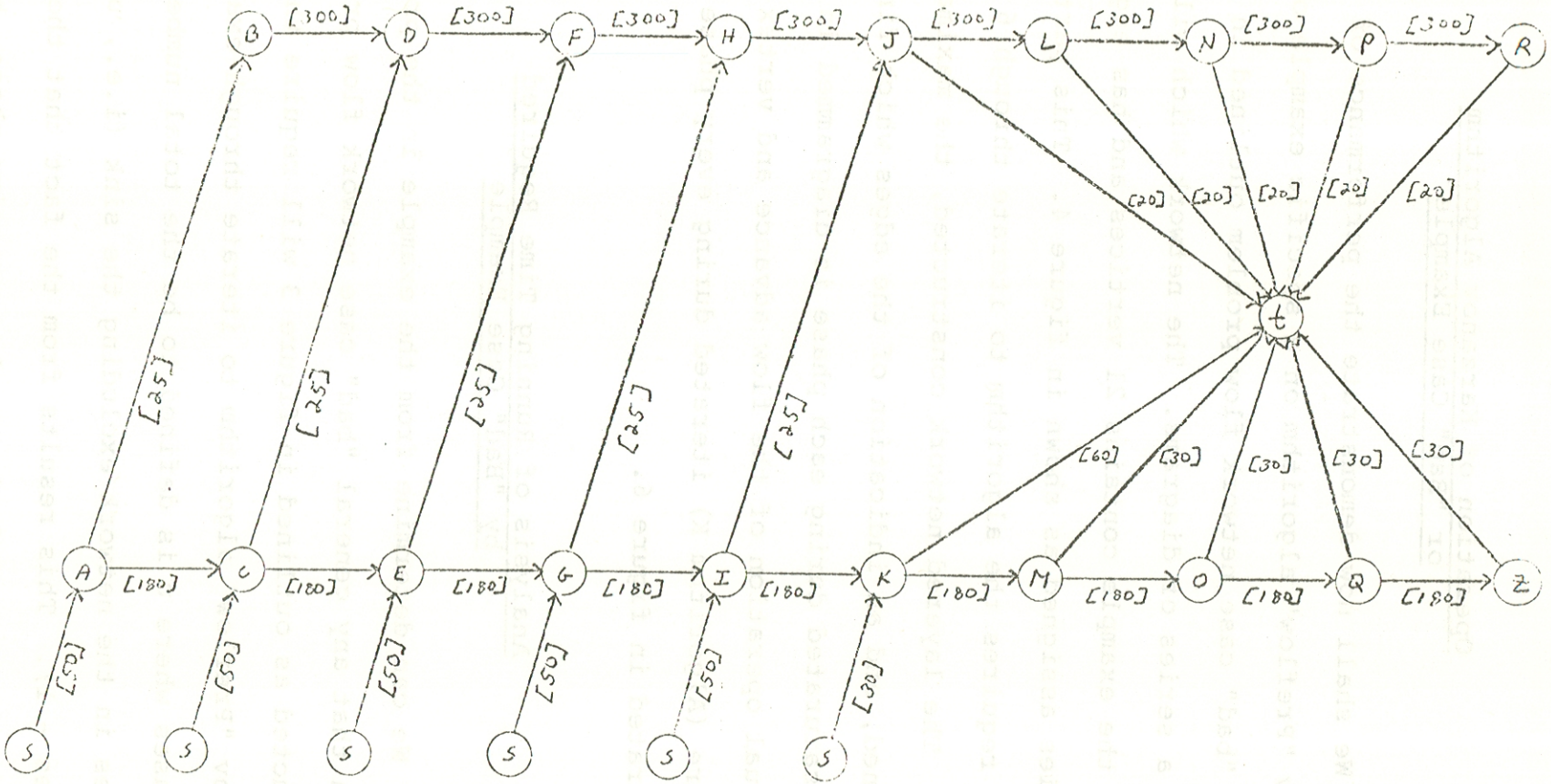
Operation of Karzanov Algorithm or "Bad" Case Example

We shall now demonstrate the performance of the Karzanov "Preflow" algorithm on a specific example of the general "bad" case network flow problem outlined in figure 3, through a series of diagrams. The network which will be used in the example contains 21 vertices and has edge capacities assigned as shown in figure 4. This particular problem requires the algorithm to iterate through 6 seperate phases. The layered network constructed, the maximal flow determined, and an indication of the edges which finally become saturated during each phase is diagrammed in figure 5. The actual operation of the flow advance and vertex balance procedure (Algorithm K) iterated during every phase is then demonstrated in figure 6.

Analysis of Running Time Required by "Bad" Case Example

We can determine from the example in the previous section that any general "bad" case network flow problem constructed as outlined in figure 3 will require the Karzanov "Preflow" algorithm to iterate through exactly $\frac{u+4}{4}$ phases where u is defined to be the total number of vertices in the network excluding the sink (i.e., u=# vertices - 1). This results from the fact that the layered network constructed during each successive phase, beginning

TWENTY-ONE VERTEX "BAD" CASE NETWORK FLOW PROBLEM
(All vertices labeled S correspond to the source)

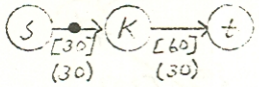


[ ] corresponds to edge capacity

Figure 4

SOLUTION PHASES REQUIRED FOR 21-VERTEX PROBLEM
(Dead-End Vertices Have Been Removed
From Layered Networks)

Phase 1: Balance Procedure - 0 times; Edge Advances - 2

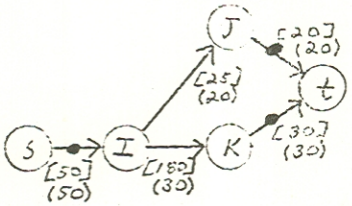


[ ] corresponds to edge capacity

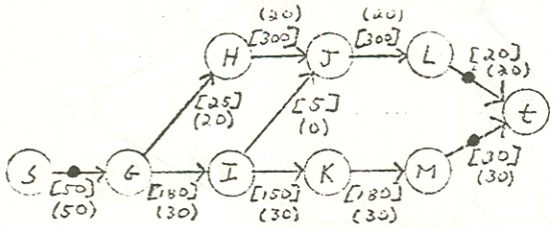
( ) corresponds to actual flow through edge

→ implies saturated edge

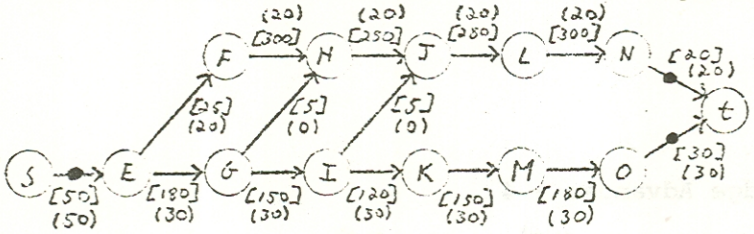
Phase 2: Balance Procedure - 1 time; Edge Advances - 5+2



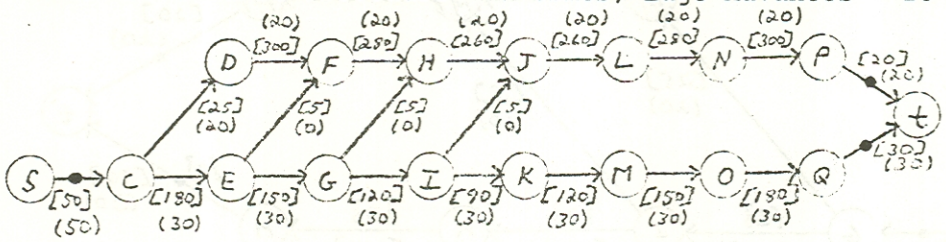
Phase 3: Balance Procedure - 3 times; Edge Advances - 10+3+4



Phase 4: Balance Procedure - 5 times; Edge Advances - 15+4+5+6



Phase 5: Balance Procedure - 7 times; Edge Advances - 20+5+6+7+8



Phase 6: Balance Procedure - 9 times; Edge Advances - 25+6+7+8+9+10

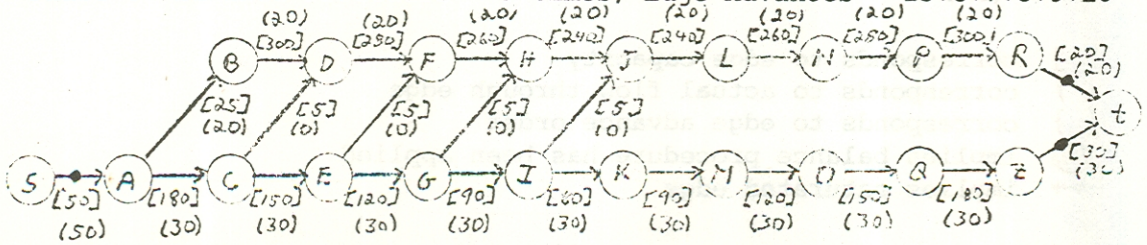
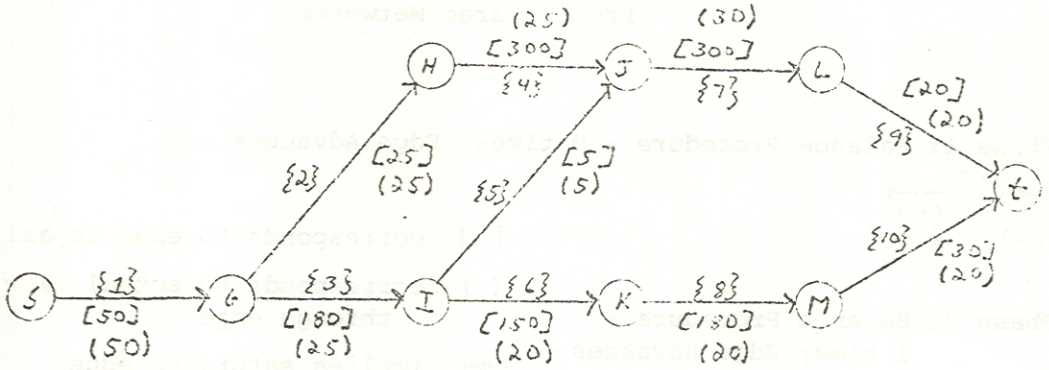


Figure 5

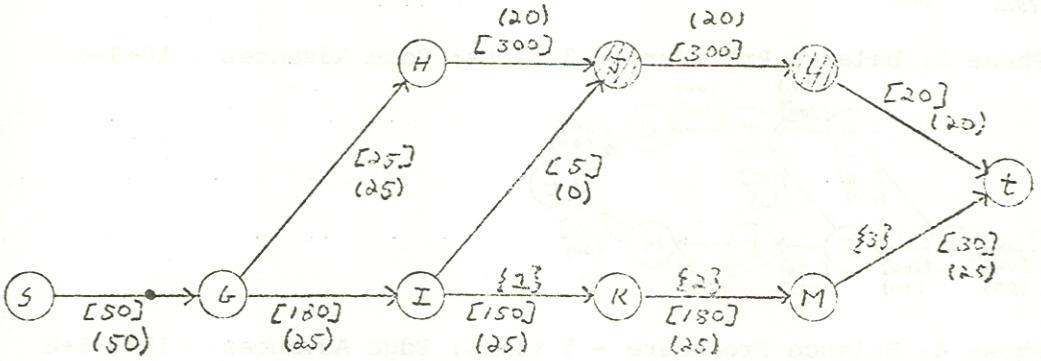
OPERATION OF ALGORITHM K ON 3rd PHASE OF 21-VERTEX PROBLEM

Vertex Balance Order - L,J,K

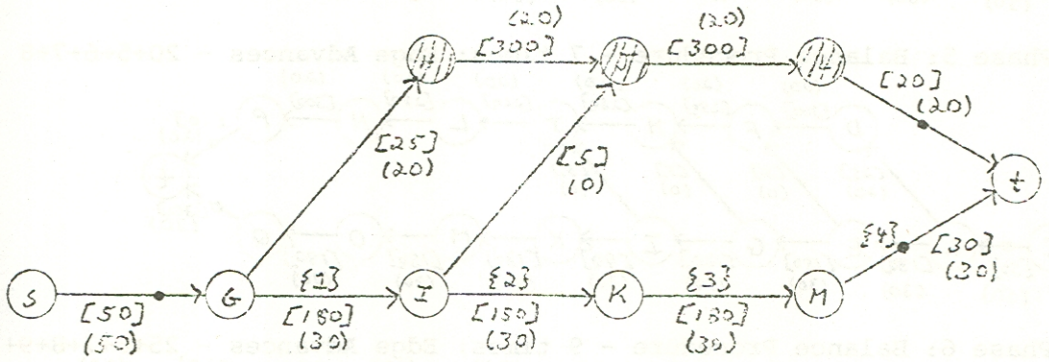


Initial Flow Advance: Edge Advances - 10

Second Flow Advance: Edge Advances - 3



Third Flow Advance: Edge Advances - 4



[ ] corresponds to edge capacity
( ) corresponds to actual flow through edge
{ } corresponds to edge advance order
▨ implies balance procedure has been applied
→● implies saturated edge

Figure 6

with phase 3, is increased in size by exactly four vertices while the increase for phase 2 is only two vertices. We can further determine, however, that the number of edge advances required during each phase ($i=1,\ldots,\frac{u+4}{4}$) can be expressed by the following formula:

$$\text{edge advances/phase} = \begin{cases} 2 & i=1 \\ 5(i-1)+\sum_{j=i}^{2i-2}(j) & i>1 \end{cases}$$

In this expression, the term 5(i-1) corresponds to the number of initial edge advances required during each phase which is simply equal to the total number of edges contained in the layered network. Each element in the summation term, on the other hand, corresponds to the number of additional edge advances required following the execution of each successive balance procedure.

We can now develop the following formula for the total running time required by the Karzanov "Preflow" algorithm to solve a general "bad" case network flow problem as described in the previous section:

$$T(u) = 2 + \sum_{i=1}^{(\frac{u+4}{4})-1}\left(5((i+1)-1) + \sum_{j=i+1}^{2(i+1)-2}(j)\right)$$

$$= 2+\sum_{i=1}^{\frac{u}{4}}\left(5(i) + \sum_{j=i+1}^{2i}(j)\right)$$

$$= \frac{u^3+28u^2+96u+256}{128}$$

Since u is simply the total number of vertices in the network flow problem excluding the sink, however, we have that this running time is $O(n^3)$.

Acknowledgements

I would like to thank Professor Ronald Rivest, Ph.D. of the M.I.T. Laboratory for Computer Science for his suggestion of this problem and great help in realizing its solution.

REFERENCES

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

[2] R. Endre Tarjan, "Testing Graph Connectivity," ACM, Seattle, Washington, April 30 - May 2, 1976.

[3] A. V. Karzanov, "Determing the Maximal Flow in a Network by the Method of Preflows," Soviet Math. Dokl., Vol. 15 (1974):434-37.

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