MULTI-TERMINAL 0-1 FLOW

by

Yossi Shiloach

STAN-CS-78-653
APRIL 1978

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Multi-Terminal 0-1 Flow

Yossi Shiloach */

Computer Science Department
Stanford University
Stanford, California 94305

January, 1978

Abstract. Given an undirected 0-1 flow network with n vertices and m edges, we present an $O(n^2(m+n))$ algorithm which generates all $\binom{n}{2}$ maximal flows between all the pairs of vertices. Since $O(n^2(m+n))$ is also the size of the output, this algorithm is optimal up to a constant factor.

Keywords and phrases: Algorithm, multiterminal flow, 0-1 integer flow.

*/ This research was supported by a Chaim Weizmann Postdoctoral Fellowship and by National Science Foundation grant MCS 75-22870.

1. Introduction.

A 0-1 undirected flow network is essentially an undirected graph $G = (V,E)$ since all the edges have one unit capacity, and the flow assumes only integer values, namely 0 or 1. G is assumed to have n vertices and m edges. The edges will be denoted as two element sets such as $\{u,v\}$.

Given $s,t \in V$, an $s \to t$ 0-1 integer flow ($s \to t$ flow in short) is a function $f: V \times V \to \{0,1\}$ such that:

(a) $f(u, v) = 0$ if $\{u,v\} \notin E$.

(b) $f(u,v) = 0$ or 1 if $\{u,v\} \in E$.

(c) If $f(u,v) = 1$, then $f(v,u) = 0$.

(d) $IN(f,v) = OUT(f,v)$ for all $v \in V-\{s,t\}$, where $IN(f,v) = \sum_{u \in V} f(u,v$

is the total amount of flow entering v and $OUT(f,v) = \sum_{w \in V} f(v,w$

. is the total amount of flow emanating from v .

The value of f denoted by $|f|$ is $OUT(f,s) - IN(f,s)$. An $s \to t$ flow f is maximal if $|f| \geq |f'|$ for any other $s \to t$ flow f'.

The 0-1 integer flow problems are usually associated with finding a maximal number of edge-disjoint or vertex-disjoint paths between two vertices in a graph, Such an individual maximal flow problem can be solved in $O(n^{2/3}(m+n))$ time, as shown in [ET].

In this paper we present an algorithm which generates the maximal flows between all the pairs of vertices within $O(n^2(m+n))$ time which seems to be optimal regarding the output size. Finding all $\binom{n}{2}$ maximal flow values can be done in $O(n^{5/3}(m+n))$ time, if we use Gomory and Hu's algorithm, (see [GH]).

2. The Multiterminal Flow Algorithm (MULTEF).

MULTEF consists of two routines. The first routine computes a cut-tree for G. A cut-tree $T = (V_T, E_T)$ is a weighted tree (i.e., a non-negative weight $w(e)$ is associated with each $e \in E_T$) with the following properties:

(a) $V_T = V$.

(b) For all $s,t \in V$, the value of a maximal $s \to t$ flow equals

$$\min_{e \in P_T(s,t)} w(e) .$$

$P_T(s,t)$ is the unique -path connecting s and t in T. (In the following we will use $d_T(s,t)$ to denote the length of $P_T(s,t)$.) The existence of such a cut-tree is proved in [GH]. They also provide an algorithm which computes the tree by solving only $n-1$ individual max-flow problems.

The second routine is MIN(u,v,w). Given a $u \to v$ flow f_{uv} and a $v \to w$ flow f_{vw}, MIN(u,v,w) computes a $u \to w$ flow f_{uw} such that

$$|f_{uw}| = \min(|f_{uv}|, |f_{vw}|) .$$

The existence of a $u \to w$ flow having this value can be easily proved by using the max-flow = min-cut theorem. MIN(u,v,w) will be described in full in the next section.

MULTEF:

1. Initialization. Compute the cut tree $T = (V_T, E_T)$ of G and $n-1$ maximal $s \to t$ flows for all s, t such that $\{s,t\} \in E_T$.

2. For $d = 2,\ldots,n-1$ do

   Begin

   Use MIN to compute maximal $s \to t$ flows for all u, v such that $d_T(s,t) = d$.

   End.

The validity of MULTEF can be easily derived from the properties. of the cut-tree (using induction on d). The complexity of MULTEF is $O(n^{5/3}(m+n))$ + $O(n^2$ .complexity of MIN) . In Section 3 we will describe a linear time algorithm for MIN which yields an $O(n^2(m+n))$ time bound for MULTEF.

3. MIN(u,v,w) .

Let $u,v,w \in V$. Given a $u \rightarrow v$ flow f_{uv} and a $v \rightarrow w$ flow f_{vw}, MIN(u,v,w) provides a $u \rightarrow w$ flow f_{uw} such that $|f_{uw}| = \min(|f_{uv}|, |f_{uw}|)$. Henceforth we assume that:

$$|f_{uv}| = |f_{vw}| \quad . \tag{3.1}$$

Both f_{uv} and f_{vw} are acyclic flows. (3.2)

If $|f_{uv}| > |f_{vw}|$, we reduce f_{uv} by $|f_{uv}| - |f_{uw}|$ units of flow so that (3.1) holds. The second assumption is justified by a linear time algorithm which eliminates cycles of flow and described in detail in Section 5.

The most straight-forward way to produce a $u \rightarrow w$ flow out of f_{uv} and f_{vw} is to add them up. So let $\phi_{uw} = f_{uv} \oplus f_{vw}$ be defined by:

$$\phi_{uw}(v_1,v_2) = \max(0 , f_{uv}(v_1,v_2) + f_{vw}(v_1,v_2) - f_{uv}(v_2,v_1) - f_{vw}(v_2,v_1)) . \tag{3.3}$$

It is easy to see that ϕ_{uw} is non-negative and if $\phi_{uw}(v_1,v_2) > 0$ then $\phi_{uw}(v_2,v_1) = 0$. Moreover, ϕ_{uw} satisfies the conservation rule, i.e., $IN(\phi_{uw}, z) = OUT(\phi_{uw}, z)$ for all $z \in V-\{u,w\}$. (Equation (3.1) implies the conservation of flow at v too.) However, edges may become overflowed as shown in Figure 1 where $\phi_{uw}(x,y) = 2$.

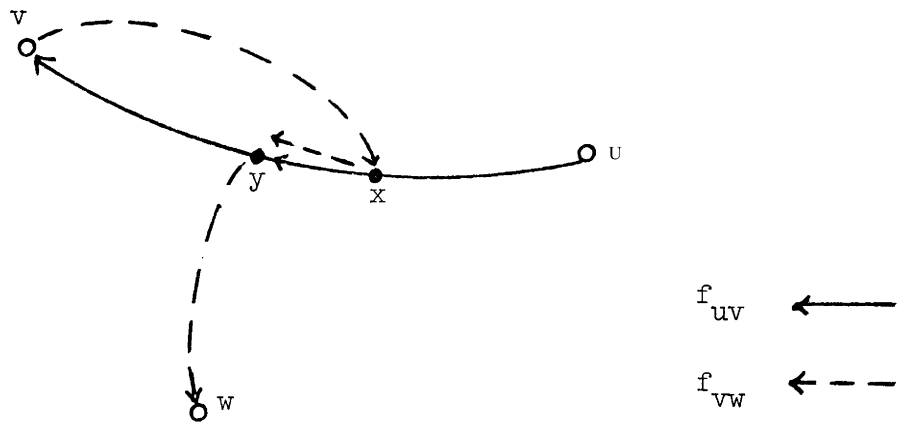


Figure1

The basic idea to resolve that problem (speaking in terms of Figure 1) is to reduce f_{uv} from x to v and reduce f_{vw} from x to v by one unit. The pseudoflow of Figure 1 turns out to be the flow of Figure 2.

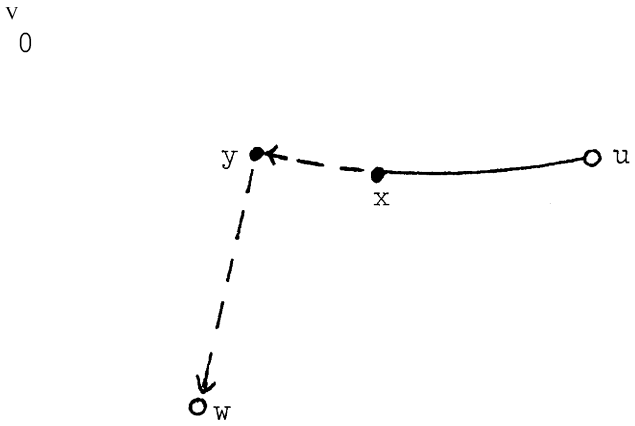
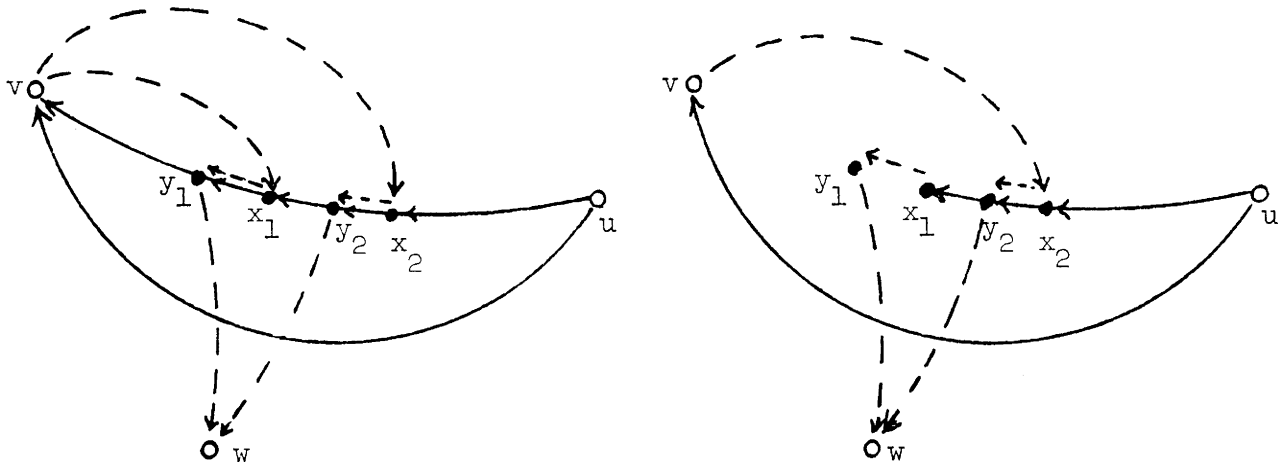


Figure2

The process of reducing f_{uv} from x to v propagates in the same direction as f_{uv} itself and will be denoted as "reducing the flow forward." or "redford" in short. Reducing f_{vw} from x to v has the opposite direction to that of f_{vw} and is called "reducing backward" or "redback". Thus, in principle, we redford f_{uv} and redback f_{vw} towards v .

Trying to implement the redford-redback idea, we might face a problem which is demonstrated in Figure 3. After reducing f_{uv} forward and f_{vw} backward from x_1 to v , we obtain the pseudo-flow of Figure 3-b. Now, we can no longer redford from x_2 . We are going to resolve this difficulty <u>partially</u> by using the acyclic orientation of $f_{\cdot\cdot}$.



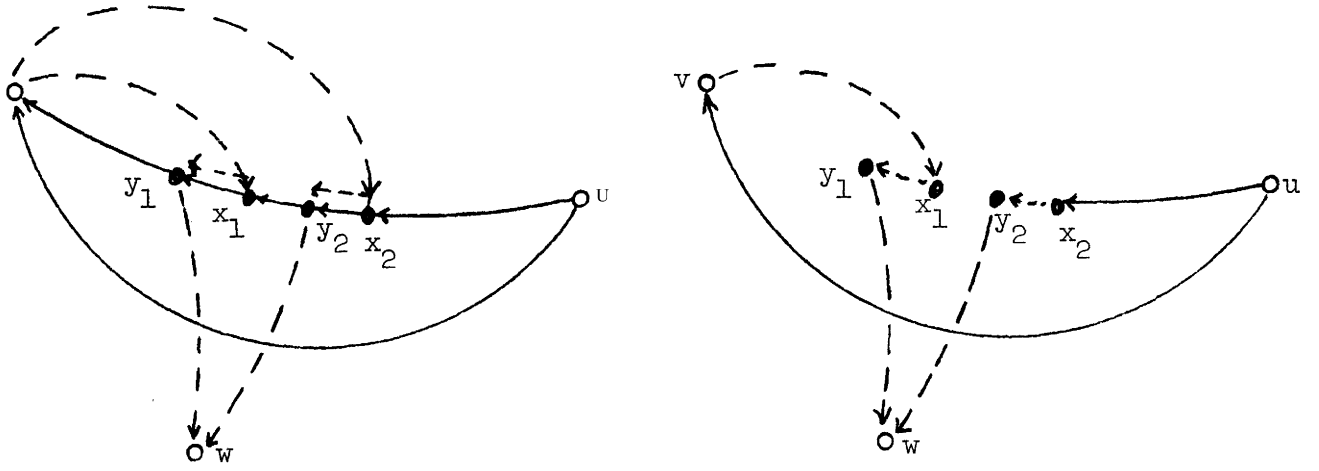
(a) (b)

Figure3

Definition. Given a flow f in an undirected flow network $G = (V,E)$, $G(f)$ is defined by:

$$G(f) = (V, E(f)) \quad \text{where} \quad E(f) = \{(v_1,v_2): f(v_1,v_2) > 0\} .$$

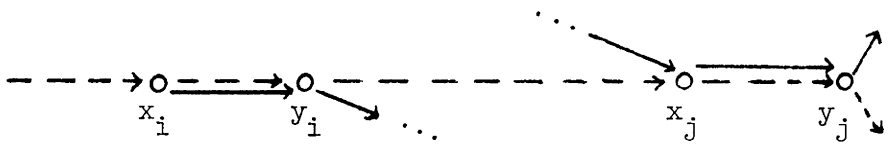
Note that $G(f)$ is a directed graph. Let $E = \{\{x_i,y_i\}: i = 1,\ldots,k\}$ denote the set of overflowed edges, i.e., $\phi_{uw}(x_i, y_i) = 2$ for $i = 1,\ldots,k$. The x_i's will be centers of the redford-redback process. The acyclity of $G(f_{uv})$ can be used to label its vertices such that $\ell(x)$ = length of a longest directed path from x to v in $G(f_{uv})$. This well-known labelling is achieved by labelling all the terminals at a time, deleting them and labelling the new terminals with the previous label + 1. We start with $\ell(v) = 0$. This labelling has the property that if there is a directed path (in $G(f_{uv})$) from x_i to x_j then $\ell(x_i) > \ell(x_j)$. Thus, if we start reducing forward from x_i's with the highest label and then go down to lower labels, we should not face the problem which is sketched in Figure 3. Figure 4 shows what happens if we start to redford-redback from x_2 which has a higher label than x_1 in $G(f_{uv})$. Note that after reducing forward and backward from x_2, no redford-redback is needed at x_1 since $\{x_1,y_1\}$ is not overflowed anymore.



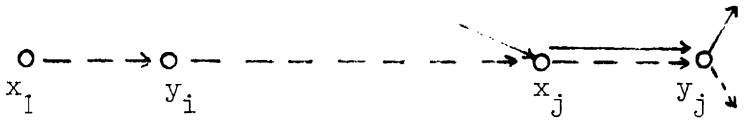
(a) (b)

Figure4

. This is only a partial solution since the x_i's are not necessarily labelled in the same order in $G(f_{vw})$. Since we must redford and redback from the same vertex (otherwise conservation is violated), we might face the same problem in reducing backward. The problem occurs when a redback path enters a vertex x_i from which a previous redback took place and now no f_{vw} flow enters x_i, (see Figure 5).



Initial state.



After a redford-redback at x_i.

If another redford-redback takes place at x_j, the redback path will be stuck at x_1.

Figure5

The solution to this problem can be outlined as follows:

Step 1. We redford along the $u \rightarrow v$ flow until no overflowed edges are left.

We now have to rebalance the vertices in which the redford paths start.

Step 2. We redback starting from the unbalanced vertices. If we get stuck we go to Step 3.

Step 3. We modify the appropriate redford path by increasing the flow along a certain prefix of it.

The algorithm is designed so that Step 3 does not yield to further modifications of the redback paths.

Detailed Implementation.

Step 1. The redford paths, say $P_1,\ldots,P_t$, are a set of edge-disjoint paths in $G(f_{uv})$ which cover all the overflowed edges. Each of the P_i's begins at an overflowed edge and we may assume that P_i begins at (x_i,y_i), $i = 1,\ldots,t$. The P_i's can be easily constructed by using the acyclity of $G(f_{uv})$. As soon as the P_i's are produced, we redford the flow by one unit along each of them.

In the same way we produce a set $\{Q_1,\ldots,Q_s\}$ of edge-disjoint redback paths in $G(f_{vw})$. Each of them starts in an overflowed edge and proceeds "backward" towards v and their union contains $\bar{E}$, The only difference is that we use the Q_j's to redback only if necessary as specified in the implementation of Step 2. The edges at which the current redford paths start, are stored in a stack which contains initially $(x_1,y_1),\ldots,(x_t,y_t)$. Following Steps 2 and 3, one can easily verify that the stack always contains only overflowed edges which are the first in their redford paths.

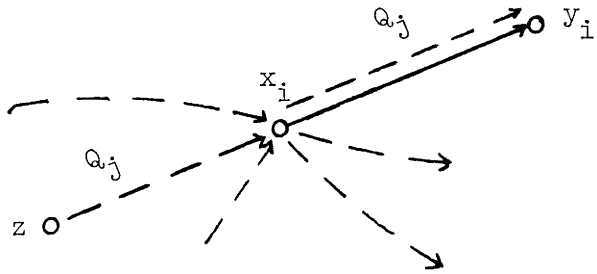
Step 2. Let (x_i,y_i) be the first edge in the stack. We start to redback at x_i, following two rules.

R_1: If $(x_i,y_i) \in Q_j$ then the redback starts at the edge which follows*/ (x_i,y_i) on Q_j, (see Figure 6).

R_2: If we start to redback at an edge which belongs to Q_j, we continue to redback along Q_j until we reach v or get stuck. If we get stuck we go to Step 3. If we reach v, we delete (x_i,y_i) from the stack. If the stack is not empty, we go back to Step 2, otherwise the algorithm terminates.

---

*/ Recall that Q_j proceeds backward.



By R_1, redback starts at (z,x_i) .

Figure 6

Lemma 3.1. A redback trail can get stuck at a vertex z only if $\exists i$ such that: $z = x_i$, (x_i,y_i) is an overflowed edge, the first on its redford path and (x_i,y_i) belong to this trail or a previous one.

Proof. Let us consider two cases.

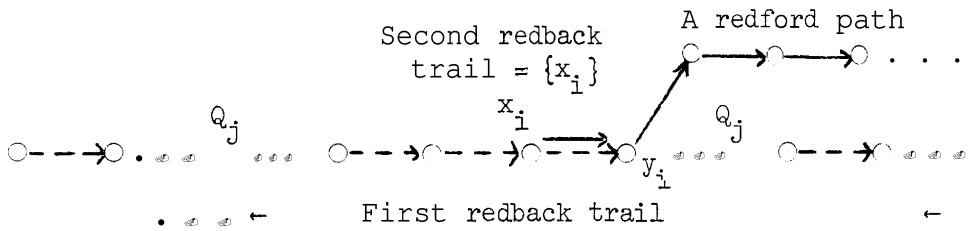
Case 1. The trail starts at z (i.e., the trail consists of a single vertex). Since redback trails start only at unbalanced vertices, $\exists i$ such that $z = x_i$ and an overflowed edge (x_i,y_i) which is the first in its redford path. (In fact, (x_i,y_i) is the top edge in the stack.) Thus (x_i,y_i) belongs to a redback path, say Q_j, and the first edge in our redback trail should have been that which follows (x_i,y_i) on Q_j , (R_1) . Since this edge has already been used in a previous redback trail (otherwise we were not stuck), (x_i,y_i) has also been used in the same trail, (R_2) .

Case 2. The trail did not start at z . The only reason that R_2 cannot be used to continue the trail is that a previous trail started at z before. This again means that $\exists i$ such that $z = x_i$ and (x_i, y_i) is an overflowed

edge, first on its redford path. It also implies that our redback trail enters x_i through $y_{(R_i,)_1}$. □

Both cases are illustrated in Figure 7.

Case 1.



Case 2.

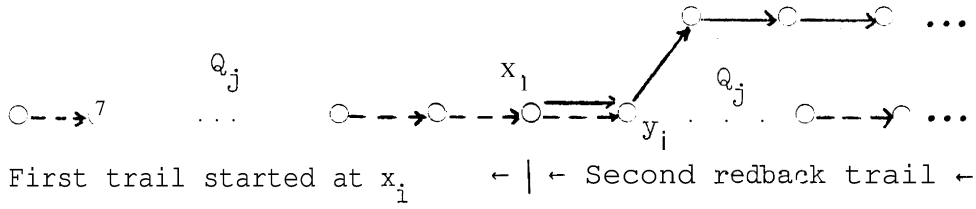


Figure 7

Step 3. Assume that our redback trail got stuck at x_i and (x_i,y_i) is the overflowed edge discussed in Lemma 3.1. Since we cannot redback anymore, we shall balance x_i by increasing the u → v flow forward ("inford") along the old redford path which starts at (x_i,y_i). The idea is that since the flow on (x_i,y_i) has been reduced before (Lemma 3.1), we no longer have to include this edge in our redford program. Thus, we inford along the redford path containing (x_i,y_i) until we reach

v or encounter another overflowed edge (x_j, y_j) in which no redback has taken place so far. In both cases, (x_i, y_i) is deleted from the stack and in the latter, (x_j, y_j) is inserted. If the stack is not empty we go back to Step 2, otherwise, the algorithm terminates.

Remark. Since (x_i, y_i) is the first edge on its redford path (Lemma 3.1), the net effect of the inford is to shorten the redford path to the minimum necessary.

4. Validity, Complexity and a Detailed Example of MIN(u,v,w) .

Validity.

We have to show that:

(1) MIN(u,v,w) terminates.

(2) The output, f_{uw} , of MIN(u,v,w) is a legal $u \to w$ flow and $|f_{uw}| = |f_{uv}| \; (= |f_{vw}|)$.

(1) The termination is quite clear.

Step 1 obviously terminates. Step 2 terminates since reducing back is done only on the $v \to w$ flow and this flow is never increased. Step 3 is finite since the $u \to v$ flow along an edge can be increased at most once.

(2) In order to show that f_{uw} is a legal flow and has the right value, we will show that:

$$\text{(A)} \quad OUT(f_{uw}, z) - IN(f_{uw}, z) = OUT(\phi_{uw}, z) - IN(\phi_{uw}, z \tag{3.4}$$

for all $z \neq v$.

(B) $IN(f_{uw},v) = OUT(f_{uw},v)$.

(C) No edge is overflowed.

(A and B): Equation (3.4) is violated during the execution in two cases. The first occurs when a redford path starts at an edge incident with z , say (z,y) . In this case (z,y) is in the stack and when it reaches its top, z will be rebalanced by the redback routine. The second case occurs when a redback trail gets stuck at z . In this case the inford routine rebalances z .

As far as u and w are concerned, "balanced" means that (3.4) holds.

Since $OUT(\phi_{uw},u) - IN(\phi_{uw},v) = |f_{uv}|$, Equation (3.4) implies that $|f_{uw}| = |f_{uv}|$ as required. Moreover, since every vertex $\neq$ u, w is balanced in ϕ_{uw} , Equation (3.4) implies that every vertex $\neq$ u , w and v is balanced in $f_{\cdot\cdot}$. (The proof of Equation (3.4) does not hold for v .) Equation (3 .1) implies that

$$OUT(\phi_{uw},u) - IN(\phi_{uw},u) = IN(\phi_{uw},w) - OUT(\phi_{uw},w)$$

and (3.4) then implies that

$$OUT(f_{uw},u) - IN(f_{uw},u) = IN(f_{uw},w) - OUT(f_{uw},w) \quad .$$

This combined with the fact that all the other vertices are balanced -- implies the balance of v .

(C) We have taken care of all the overflowed edges at Step 1. We increase the flow again only in Step 3. However, as explained there, this increase does not overflow any edge again.

Complexity.

Producing ϕ_{uw} is obviously linear. An edge of $G(f_{uv})$ is treated at most twice (steps 1 and 3) and an edge of $G(f_{vw})$ is treated at most once (step 2). Thus, MIN(u,v,w) is linear.

A Detailed Example.

In Figure 8 we illustrate the composition of a $u \to v$ flow f_{uv} and a $v \to w$ flow f_{vw}. Both are acyclic and have the same value, 3.

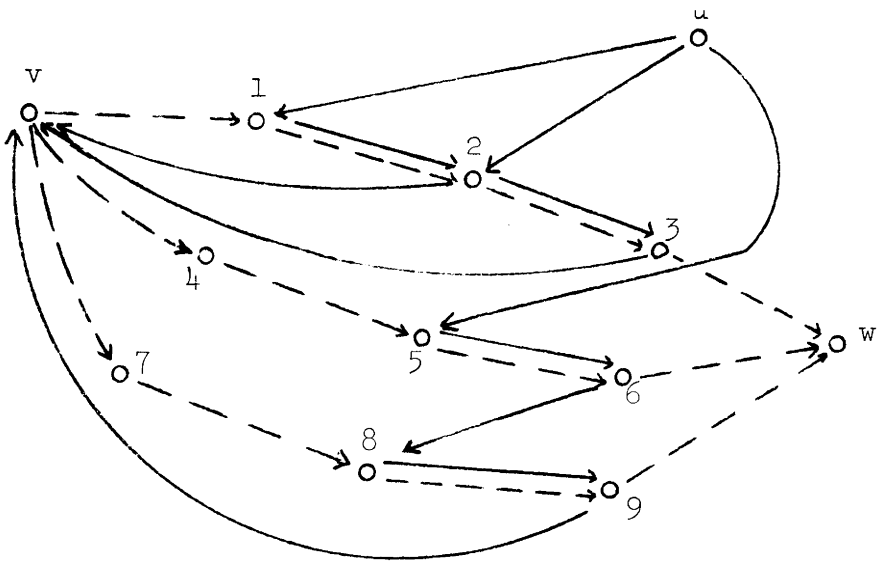


Figure 8

The overflowed edges are (1,2), (2,3), (5,6) and (8,9).

Let the redford paths be:

$$P_1 = (1,2,v), \quad P_2 = (2,3,v), \quad P_3 = (5,6,8,9,v).$$

The redback paths are:

$$Q_1 = (3,2,1,v), \quad Q_2 = (6,5,4,v), \quad Q_3 = (9,8,7,v).$$

The stack contains ((1,2),(2,3),(5,6)).

Figure 9 illustrates the situation after redford has been completed.

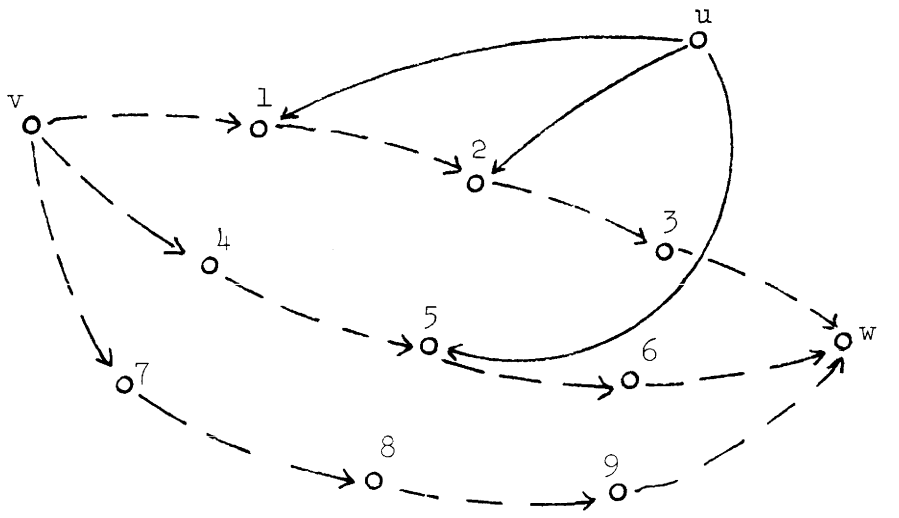


Figure 9

Redback trails should start, at 1 , 2 and 5 (one at each).

First redback trail = (1,v) ; stack = ((2,3),(5,6)) .

Second redback trail = (2,1) , got stuck at 1 .

Figure 10 illustrates the situation at this moment.

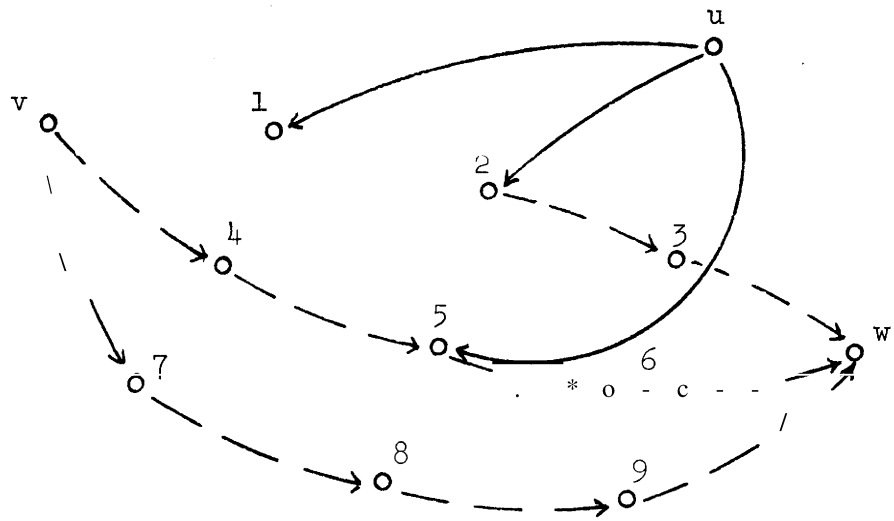


Figure 10

Now inford takes place, starting at (1,2) .

Inford trail = (1,2,v) ; stack = ((5,6)) .

Third redback trail = (5,4,v) ; stack = $\emptyset$.

The final u → w flow is illustrated in Figure 11.

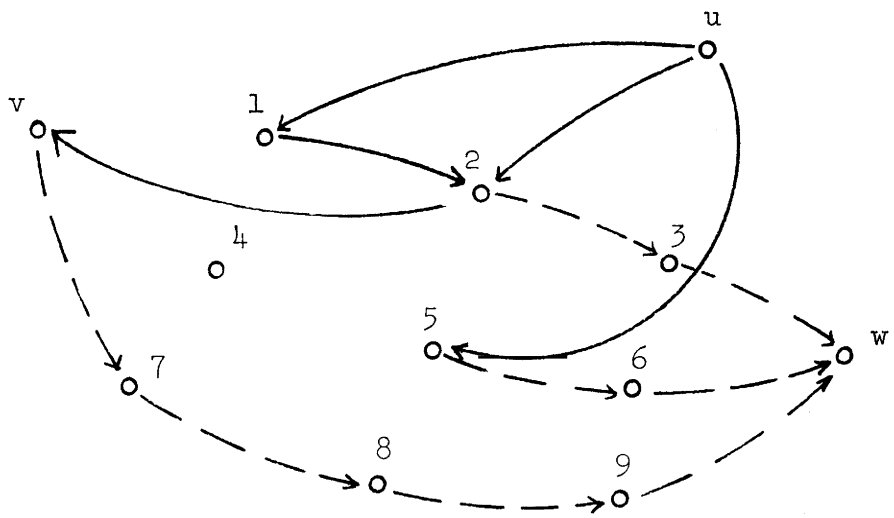


Figure 11

5. ACYC(f) . (R. E. Tarjan)

Given a $u \rightarrow v$ flow f , ACYC(f) produces an acyclic $u \rightarrow v$ flow f' of the same value. We -perform a depth-first-search on G(f) , starting from u . Each vertex z is labelled when we enter it and the label is removed when we backtrack through it. The label consists of the name of the (unique) vertex through which we entered z (the "father" of z). We step forward until one of the two cases occur;

(a) We reach a terminal vertex t .

In this case we backtrack to the father of t removing t and its incident edges from G(f) . (Since no cycle can pass through t). Note that the first terminal vertex which we will encounter is v , however, when v is removed, other terminal vertices might be created.

(b) We reach a labelled vertex z .

This means that we discovered a flow cycle C through z . We backtrack through C and remove its edges from G(f) . The labels of the vertices of C-{z} are removed and the search is continued from z .

G(f') consists of the vertices of G(f) and of the edges of G(f) which were not removed during cycles elimination, (Case b). If G(f) is not connected we delete first those edges which are not on the (weakly) connected component which contains u and v. The validity and linearity of ACYC(f) can be easily proved.

References

[ET] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," SIAM J. on Computing 4, (1975), 507-518.

[GH] R. E. Gomory and T. C. Hu, "Multi-terminal network flows," SIAM J. of Applied Math. 9, (1961), 551-570.