

# Algorithms and Complexity

Herbert S. Wilf  
University of Pennsylvania  
Philadelphia, PA 19104-6395

## Copyright Notice

Copyright 1994 by Herbert S. Wilf. This material may be reproduced for any educational purpose, multiple copies may be made for classes, etc. Charges, if any, for reproduced copies must be just enough to recover reasonable costs of reproduction. Reproduction for commercial purposes is prohibited. This cover page must be included in all distributed copies.

## Internet Edition, Summer, 1994

This edition of Algorithms and Complexity is the file "`pub/wilf/AlgComp.ps.Z`" at the anonymous ftp site "`ftp.cis.upenn.edu`". It may be taken at no charge by all interested persons. Comments and corrections are welcome, and should be sent to `wilf@central.cis.upenn.edu`

## Chapter 3: The Network Flow Problem

### 3.1 Introduction

The network flow problem is an example of a beautiful theoretical subject that has many important applications. It also has generated algorithmic questions that have been in a state of extremely rapid development in the past 20 years. Altogether, the fastest algorithms that are now known for the problem are much faster, and some are much simpler, than the ones that were in use a short time ago, but it is still unclear how close to the ‘ultimate’ algorithm we are.

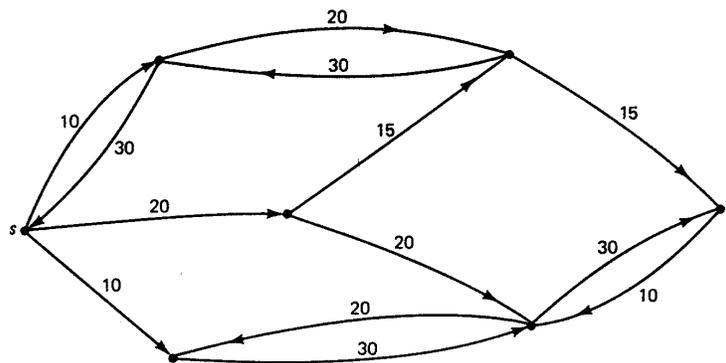
**Definition.** A *network* is an edge-capacitated directed graph, with two distinguished vertices called the *source* and the *sink*.

To repeat that, this time a little more slowly, suppose first that we are given a directed graph (*digraph*)  $G$ . That is, we are given a set of vertices, and a set of *ordered* pairs of these vertices, these pairs being the *edges* of the digraph. It is perfectly OK to have both an edge from  $u$  to  $v$  and an edge from  $v$  to  $u$ , or both, or neither, for all  $u \neq v$ . No edge  $(u, u)$  is permitted. If an edge  $e$  is directed *from* vertex  $v$  to vertex  $w$ , then  $v$  is the *initial* vertex of  $e$  and  $w$  is the *terminal* vertex of  $e$ . We may then write  $v = \text{Init}(e)$  and  $w = \text{Term}(e)$ .

Next, in a network there is associated with each directed edge  $e$  of the digraph a positive real number called its *capacity*, and denoted by  $\text{cap}(e)$ .

Finally, two of the vertices of the digraph are distinguished. One,  $s$ , is the source, and the other,  $t$ , is the sink of the network.

We will let  $\mathbf{X}$  denote the resulting network. It consists of the digraph  $G$ , the given set of edge capacities, the source, and the sink. A network is shown in Fig. 3.1.1.



**Fig. 3.1.1: A network**

Now roughly speaking, we can think of the edges of  $G$  as conduits for a fluid, the capacity of each edge being the carrying-capacity of the edge for that fluid. Imagine that the fluid flows in the network from the source to the sink, in such a way that the amount of fluid in each edge does not exceed the capacity of that edge.

We want to know the maximum net quantity of fluid that could be flowing from source to sink.

That was a rough description of the problem; here it is more precisely.

**Definition.** A *flow* in a network  $\mathbf{X}$  is a function  $f$  that assigns to each edge  $e$  of the network a real number  $f(e)$ , in such a way that

- (1) For each edge  $e$  we have  $0 \leq f(e) \leq \text{cap}(e)$  and
- (2) For each vertex  $v$  other than the source and the sink, it is true that

$$\sum_{\text{Init}(e)=v} f(e) = \sum_{\text{Term}(e)=v} f(e). \tag{3.1.1}$$

The condition (3.1.1) is a flow conservation condition. It states that the outflow from  $v$  (the left side of (3.1.1)) is equal to the inflow to  $v$  (the right side) for all vertices  $v$  other than  $s$  and  $t$ . In the theory of electrical networks such conservation conditions are known as Kirchoff's laws. Flow cannot be manufactured anywhere in the network except at  $s$  or  $t$ . At other vertices, only redistribution or rerouting takes place.

Since the source and the sink are exempt from the conservation conditions there may, and usually will, be a nonzero net flow out of the source, and a nonzero net flow into the sink. Intuitively it must already be clear that these two are equal, and we will prove it below, in section 3.4. If we let  $Q$  be the net outflow from the source, then  $Q$  is also the net inflow to the sink.

The quantity  $Q$  is called the *value of the flow*.

In Fig. 3.1.2 there is shown a flow in the network of Fig. 3.1.1. The amounts of flow in each edge are shown in the square boxes. The other number on each edge is its capacity. The letter inside the small circle next to each vertex is the name of that vertex, for the purposes of the present discussion. The value of the flow in Fig. 3.1.2 is  $Q = 32$ .

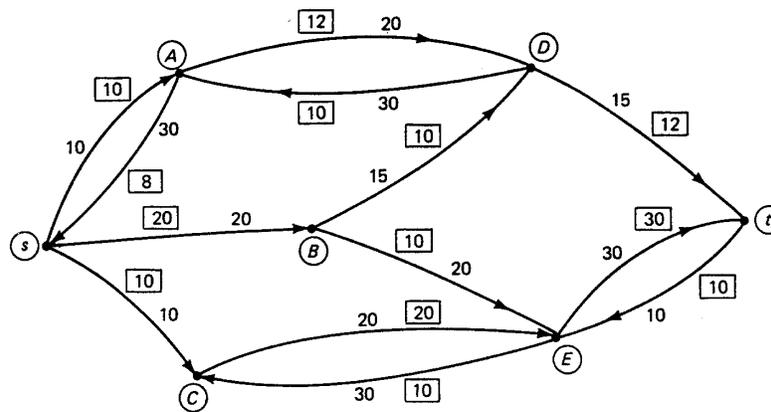


Fig. 3.1.2: A flow in a network

The *network flow problem*, the main subject of this chapter, is: *given a network  $\mathbf{X}$ , find the maximum possible value of a flow in  $\mathbf{X}$ , and find a flow of that value.*

### 3.2 Algorithms for the network flow problem

The first algorithm for the network flow problem was given by Ford and Fulkerson. They used that algorithm not only to solve instances of the problem, but also to prove theorems about network flow, a particularly happy combination. In particular, they used their algorithm to prove the ‘max-flow-min-cut’ theorem, which we state below as theorem 3.4.1, and which occupies a central position in the theory.

The speed of their algorithm, it turns out, depends on the edge capacities in the network as well as on the numbers  $V$  of vertices, and  $E$  of edges, of the network. Indeed, for certain (irrational) values of edge capacities they found that their algorithm might not converge at all (see section 3.5).

In 1969 Edmonds and Karp gave the first algorithm for the problem whose speed is bounded by a polynomial function of  $E$  and  $V$  only. In fact that algorithm runs in time  $O(E^2V)$ . Since then there has been a steady procession of improvements in the algorithms, culminating, at the time of this writing anyway, with an  $O(EV \log V)$  algorithm. The chronology is shown in Table 3.2.1.

The maximum number of edges that a network of  $V$  vertices can have is  $\Theta(V^2)$ . A family of networks might be called *dense* if there is a  $K > 0$  such that  $|E(\mathbf{X})| > K|V(\mathbf{X})|^2$  for all networks in the family. The reader should check that for dense networks, all of the time complexities in Table 3.2.1, beginning with Karzanov’s algorithm, are in the neighborhood of  $O(V^3)$ . On the other hand, for *sparse* networks (networks with relatively few edges), the later algorithms in the table will give significantly better performances than the earlier ones.

<i>Author(s)</i>	<i>Year</i>	<i>Complexity</i>
Ford, Fulkerson	1956	-----
Edmonds, Karp	1969	$O(E^2V)$
Dinic	1970	$O(EV^2)$
Karzanov	1973	$O(V^3)$
Cherkassky	1976	$O(\sqrt{EV}^2)$
Malhotra, <i>et al.</i>	1978	$O(V^3)$
Galil	1978	$O(V^{5/3}E^{2/3})$
Galil and Naamad	1979	$O(EV \log^2 V)$
Sleator and Tarjan	1980	$O(EV \log V)$
Goldberg and Tarjan	1985	$O(EV \log(V^2/E))$

**Table 3.2.1: Progress in network flow algorithms**

**Exercise 3.2.1.** Given  $K > 0$ . Consider the family of all possible networks  $\mathbf{X}$  for which  $|E(\mathbf{X})| = K|V(\mathbf{X})|$ . In this family, evaluate all of the complexity bounds in Table 3.2.1 and find the fastest algorithm for the family.

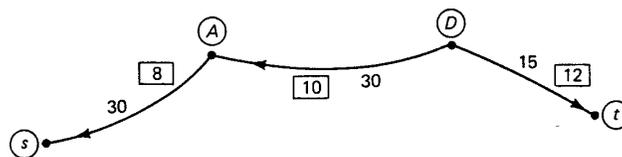
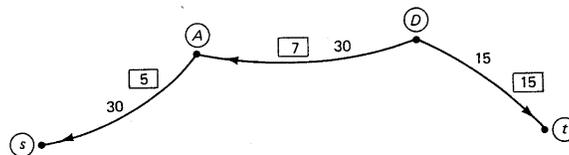
Among the algorithms in Table 3.2.1 we will discuss just two in detail. The first will be the original algorithm of Ford and Fulkerson, because of its importance and its simplicity, if not for its speed.

The second will be the 1978 algorithm of Malhotra, Pramodh-Kumar and Maheshwari (MPM), for three reasons. It uses the idea, introduced by Dinic in 1970 and common to all later algorithms, of *layered networks*, it is fast, and it is extremely simple and elegant in its conception, and so it represents a good choice for those who may wish to program one of these algorithms for themselves.

### 3.3 The algorithm of Ford and Fulkerson

The basic idea of the Ford-Fulkerson algorithm for the network flow problem is this: start with some flow function (initially this might consist of zero flow on every edge). Then look for a *flow augmenting path* in the network. A flow augmenting path is a path from the source to the sink along which we can push some additional flow.

In Fig. 3.3.1 below we show a flow augmenting path for the network of Fig. 3.2.1. The capacities of the edges are shown on each edge, and the values of the flow function are shown in the boxes on the edges.

**Fig. 3.3.1: A flow augmenting path****Fig. 3.3.2: The path above, after augmentation.**

An edge can get elected to a flow augmenting path for two possible reasons. Either

- (a) the direction of the edge is *coherent* with the direction of the path from source to sink and the present value of the flow function on the edge is below the capacity of that edge, or
- (b) the direction of the edge is *opposed* to that of the path from source to sink and the present value of the flow function on the edge is strictly positive.

Indeed, on all edges of a flow augmenting path that are coherently oriented with the path we can increase the flow along the edge, and on all edges that are incoherently oriented with the path we can decrease the flow on the edge, and in either case we will have *increased the value of the flow* (think about that one until it makes sense).

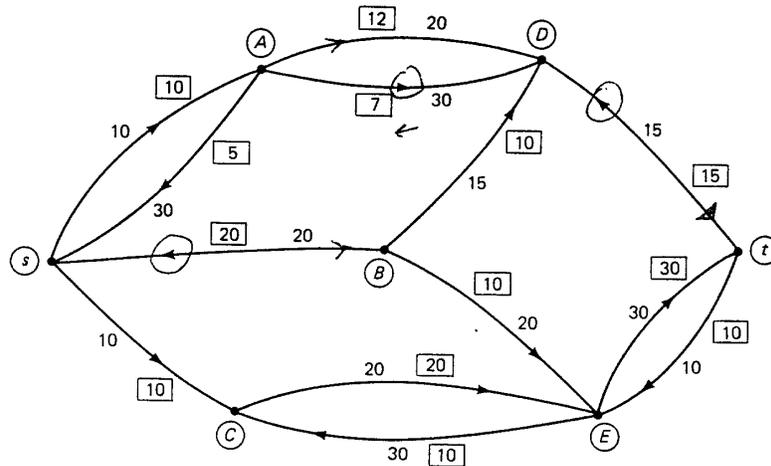
It is, of course, necessary to maintain the conservation of flow, *i.e.*, to respect Kirchhoff's laws. To do this we will augment the flow on every edge of an augmenting path by the same amount. If the conservation conditions were satisfied before the augmentation then they will still be satisfied after such an augmentation.

It may be helpful to remark that an edge is coherently or incoherently oriented only *with respect to a given path* from source to sink. That is, the coherence, or lack of it, is not only a property of the directed edge, but depends on how the edge sits inside a chosen path.

Thus, in Fig. 3.3.1 the first edge is directed towards the source, *i.e.*, incoherently with the path. Hence if we can *decrease* the flow in that edge we will have *increased* the value of the flow function, namely the net flow out of the source. That particular edge can indeed have its flow decreased, by at most 8 units. The next edge carries 10 units of flow towards the source. Therefore if we *decrease* the flow on that edge, by up to 10 units, we will also have *increased* the value of the flow function. Finally, the edge into the sink carries 12 units of flow and is oriented towards the sink. Hence if we *increase* the flow in this edge, by at most 3 units since its capacity is 15, we will have increased the value of the flow in the network.

Since every edge in the path that is shown in Fig. 3.3.1 can have its flow altered in one way or the other so as to increase the flow in the network, the path is indeed a flow augmenting path. The most that we might accomplish with this path would be to push 3 more units of flow through it from source to sink. We couldn't push more than 3 units through because one of the edges (the edge into the sink) will tolerate an augmentation of only 3 flow units before reaching its capacity.

To augment the flow by 3 units we would diminish the flow by 3 units on each of the first two edges and increase it by 3 units on the last edge. The resulting flow in this path is shown in Fig. 3.3.2. The flow in the full network, after this augmentation, is shown in Fig. 3.3.3. Note carefully that if these augmentations are made then flow conservation at each vertex of the network will still hold (check this!).



**Fig. 3.3.3: The network, after augmentation of flow**

After augmenting the flow by 3 units as we have just described, the resulting flow will be the one that is shown in Fig. 3.3.3. The value of the flow in Fig. 3.1.2 was 32 units. After the augmentation, the flow function in Fig. 3.3.3 has a value of 35 units.

We have just described the main idea of the Ford-Fulkerson algorithm. It first finds a flow augmenting path. Then it augments the flow along that path as much as it can. Then it finds another flow augmenting

path, etc. etc. The algorithm terminates when no flow augmenting paths exist. We will prove that when that happens, the flow will be at the maximum possible value, *i.e.*, we will have found the solution of the network flow problem.

We will now describe the steps of the algorithm in more detail.

**Definition.** Let  $f$  be a flow function in a network  $\mathbf{X}$ . We say that an edge  $e$  of  $\mathbf{X}$  is usable from  $v$  to  $w$  if either  $e$  is directed from  $v$  to  $w$  and the flow in  $e$  is less than the capacity of the edge, or  $e$  is directed from  $w$  to  $v$  and the flow in  $e$  is  $> 0$ .

Now, given a network and a flow in that network, how do we find a flow augmenting path from the source to the sink? This is done by a process of labelling and scanning the vertices of the network, beginning with the source and proceeding out to the sink. Initially all vertices are in the conditions ‘unlabeled’ and ‘unscanned.’ As the algorithm proceeds, various vertices will become labeled, and if a vertex is labeled, it may become *scanned*. To scan a vertex  $v$  means, roughly, that we stand at  $v$  and look around at all neighbors  $w$  of  $v$  that haven’t yet been labeled. If  $e$  is some edge that joins  $v$  with a neighbor  $w$ , and if the edge  $e$  is usable from  $v$  to  $w$  as defined above, then we will label  $w$ , because any flow augmenting path that has already reached from the source to  $v$  can be extended another step, to  $w$ .

The label that every vertex  $v$  gets is a triple  $(u, \pm, z)$ , and here is what the three items mean.

The ‘ $u$ ’ part of the label of  $v$  is the name of the vertex that was being scanned when  $v$  was labeled.

The ‘ $\pm$ ’ will be ‘+’ if  $v$  was labeled because the edge  $(u, v)$  was usable from  $u$  to  $v$  (*i.e.*, if the flow from  $u$  to  $v$  was less than the capacity of  $(u, v)$ ) and it will be ‘-’ if  $v$  was labeled because the edge  $(v, u)$  was usable from  $u$  to  $v$  (*i.e.*, if the flow from  $v$  to  $u$  was  $> 0$ ).

Finally, the ‘ $z$ ’ component of the label represents the largest amount of flow that can be pushed from the source to the present vertex  $v$  along any augmenting path that has so far been found. At each step the algorithm will replace the current value of  $z$  by the amount of new flow that could be pushed through to  $z$  along the edge that is now being examined, if that amount is smaller than  $z$ .

So much for the meanings of the various labels. As the algorithm proceeds, the labels that get attached to the different vertices form a record of how much flow can be pushed through the network from the source to the various vertices, and by exactly which routes.

To begin with, the algorithm labels the source with  $(-\infty, +, \infty)$ . The source now has the label-status *labeled* and the scan-status *unscanned*.

Next we will scan the source. Here is the procedure for scanning any vertex  $u$ .

```

procedure scan( $u$ :vertex; $\mathbf{X}$  :network;  $f$ :flow );
for every ‘unlabeled’ vertex  $v$  that is connected
  to  $u$  by an edge in either or both directions, do
    if the flow in  $(u, v)$  is less than  $cap(u, v)$ 
      then
        label  $v$  with  $(u, +, \min\{z(u), cap(u, v) - flow(u, v)\})$ 
      else if the flow in  $(v, u)$  is  $> 0$ 
        then
          label  $v$  with  $(u, -, \min\{z(u), flow(v, u)\})$  and
          change the label-status of  $v$  to ‘labeled’;
    change the scan-status of  $u$  to ‘scanned’
end.{scan}

```

We can use the above procedure to describe the complete scanning and labelling of the vertices of the network, as follows.

```

procedure labelandscan(X :network; f:flow; whyhalt:reason);
give every vertex the scan-status 'unscanned'
    and the label-status 'unlabeled';
u := source;
label source with  $(-\infty, +, \infty)$ ;
label-status of source := 'labeled';
while {there is a 'labeled' and 'unscanned' vertex v
        and sink is 'unlabeled'}
    do scan(v, X, f);
    if sink is unlabeled
        then 'whyhalt' := 'flow is maximum'
        else 'whyhalt' := 'it's time to augment'
    end.{labelandscan}

```

Obviously the labelling and scanning process will halt for one of two reasons: either the sink  $t$  acquires a label, or the sink never gets labeled but no more labels can be given. In the first case we will see that a flow augmenting path from source to sink has been found, and in the second case we will prove that the flow is at its maximum possible value, so the network flow problem has been solved.

Suppose the sink does get a label, for instance the label  $(u, \pm, z)$ . Then we claim that the value of the flow in the network can be augmented by  $z$  units.

To prove this we will construct a flow augmenting path, using the labels on the vertices, and then we will change the flow by  $z$  units on every edge of that path in such a way as to increase the value of the flow function by  $z$  units. This is done as follows.

If the sign part of the label of  $t$  is '+,' then increase the flow function by  $z$  units on the edge  $(u, t)$ , else decrease the flow on edge  $(t, u)$  by  $z$  units.

Then move back one step away from the sink, to vertex  $u$ , and look at its label, which might be  $(w, \pm, z_1)$ . If the sign is '+' then increase the flow on edge  $(w, u)$  by  $z$  units (not by  $z_1$  units!), while if the sign is '-' then decrease the flow on edge  $(u, w)$  by  $z$  units. Next replace  $u$  by  $w$ , etc., until the source  $s$  has been reached.

A little more formally, the flow augmentation algorithm is the following.

```

procedure augmentflow(X :network; f:flow ; amount:real);
{assumes that labelandscan has just been done}
v := sink;
amount := the 'z' part of the label of sink;
repeat
    (previous, sign, z) := label(v);
    if sign = '+'
        then
            increase f(previous, v) by amount
        else
            decrease f(v, previous) by amount;
    v := previous
until v = source
end.{augmentflow}

```

The value of the flow in the network has now been *increased* by  $z$  units. The whole process of labelling and scanning is now repeated, to search for another flow augmenting path. The algorithm halts only when we are unable to label the sink. The complete Ford-Fulkerson algorithm is shown below.

```

procedure fordfulkerson( $\mathbf{X}$  :network;  $f$ : flow;  $maxflowvalue$ :real);
{finds maximum flow in a given network  $\mathbf{X}$  }
set  $f:=0$  on every edge of  $\mathbf{X}$  ;
 $maxflowvalue:=0$ ;
repeat
  labelandscan( $\mathbf{X}$ ,  $f$ , whyhalt);
  if whyhalt='it's time to augment' then
    augmentflow( $\mathbf{X}$ ,  $f$ , amount);
     $maxflowvalue := maxflowvalue + amount$ 
  until whyhalt = 'flow is maximum'
end.{fordfulkerson}

```

Let's look at what happens if we apply the labelling and scanning algorithm to the network and flow shown in Fig. 3.1.2. First vertex  $s$  gets the label  $(-\infty, +, \infty)$ . We then scan  $s$ . Vertex  $A$  gets the label  $(s, -, 8)$ ,  $B$  cannot be labeled, and  $C$  gets labeled with  $(s, +, 10)$ , which completes the scan of  $s$ .

Next we scan vertex  $A$ , during which  $D$  acquires the label  $(A, +, 8)$ . Then  $C$  is scanned, which results in  $E$  getting the label  $(C, -, 10)$ . Finally, the scan of  $D$  results in the label  $(D, +, 3)$  for the sink  $t$ .

From the label of  $t$  we see that there is a flow augmenting path in the network along which we can push 3 more units of flow from  $s$  to  $t$ . We find the path as in procedure *augmentflow* above, following the labels backwards from  $t$  to  $D$ ,  $A$  and  $s$ . The path in question will be seen to be exactly the one shown in Fig. 3.3.1, and further augmentation proceeds as we have discussed above.

### 3.4 The max-flow min-cut theorem

Now we are going to look at the state of affairs that holds when the flow augmentation procedure terminates because it has not been able to label the sink. We want to show that then the flow will have a maximum possible value.

Let  $W \subset V(\mathbf{X})$ , and suppose that  $W$  contains the source and  $W$  does not contain the sink. Let  $\overline{W}$  denote

all other vertices of  $\mathbf{X}$ , *i.e.*,  $\overline{W} = V(\mathbf{X}) - W$ .

**Definition.** By the *cut*  $(W, \overline{W})$  we mean the set of all edges of  $\mathbf{X}$  whose initial vertex is in  $W$  and whose terminal vertex is in  $\overline{W}$ .

For example, one cut in a network consists of all edges whose initial vertex is the source.

Now, every unit of flow that leaves the source and arrives at the sink must at some moment flow from a vertex of  $W$  to a vertex of  $\overline{W}$ , *i.e.*, must flow along some edge of the cut  $(W, \overline{W})$ . If we define the *capacity of a cut* to be the sum of the capacities of all edges in the cut, then it seems clear that the value of a flow can never exceed the capacity of any cut, and therefore that the *maximum* value of a flow cannot exceed the *minimum* capacity of any cut.

The main result of this section is the 'max-flow min-cut' theorem of Ford and Fulkerson, which we state as

**Theorem 3.4.1.** *The maximum possible value of any flow in a network is equal to the minimum capacity of any cut in that network.*

**Proof:** We will first do a little computation to show that the value of a flow can never exceed the capacity of a cut. Second, we will show that when the Ford-Fulkerson algorithm terminates because it has been unable to label the sink, then at that moment there is a cut in the network whose edges are saturated with flow, *i.e.*, such that the flow in each edge of the cut is equal to the capacity of that edge.

Let  $U$  and  $V$  be two (not necessarily disjoint) sets of vertices of the network  $\mathbf{X}$ , and let  $f$  be a flow function for  $\mathbf{X}$ . By  $f(U, V)$  we mean the sum of the values of the flow function along all edges whose initial vertex lies in  $U$  and whose terminal vertex lies in  $V$ . Similarly, by  $cap(U, V)$  we mean the sum of the capacities of all of those edges. Finally, by the *net flow out of*  $U$  we mean  $f(U, \overline{U}) - f(\overline{U}, U)$ .

**Lemma 3.4.1.** Let  $f$  be a flow of value  $Q$  in a network  $\mathbf{X}$ , and let  $(W, \overline{W})$  be a cut in  $\mathbf{X}$ . Then

$$Q = f(W, \overline{W}) - f(\overline{W}, W) \leq \text{cap}(W, \overline{W}). \quad (3.4.1)$$

**Proof of lemma:** The net flow out of  $s$  is  $Q$ . The net flow out of any other vertex  $w \in W$  is 0. Hence, if  $V(\mathbf{X})$  denotes the vertex set of the network  $\mathbf{X}$ , we obtain

$$\begin{aligned} Q &= \sum_{w \in W} \{f(w, V(\mathbf{X})) - f(V(\mathbf{X}), w)\} \\ &= f(W, V(\mathbf{X})) - f(V(\mathbf{X}), W) \\ &= f(W, W \cup \overline{W}) - f(W \cup \overline{W}, W) \\ &= f(W, W) + f(W, \overline{W}) - f(W, W) - f(\overline{W}, W) \\ &= f(W, \overline{W}) - f(\overline{W}, W). \end{aligned}$$

This proves the ‘=’ part of (3.4.1), and the ‘ $\leq$ ’ part is obvious, completing the proof of lemma 3.4.1. ■

We now know that the maximum value of the flow in a network cannot exceed the minimum of the capacities of the cuts in the network.

To complete the proof of the theorem we will show that a flow of maximum value, which surely exists, must saturate the edges of some cut.

Hence, let  $f$  be a flow in  $X$  of maximum value, and call procedure  $\text{labelandscan}(\mathbf{X}, f, \text{whyhalt})$ . Let  $W$  be the set of vertices of  $\mathbf{X}$  that have been labeled when the algorithm terminates. Clearly  $s \in W$ . Equally clearly,  $t \notin W$ , for suppose the contrary. Then we would have termination with ‘whyhalt’ = ‘it’s time to augment,’ and if we were then to call procedure  $\text{augmentflow}$  we would find a flow of higher value, contradicting the assumed maximality of  $f$ .

Since  $s \in W$  and  $t \notin W$ , the set  $W$  defines a cut  $(W, \overline{W})$ .

We claim that every edge of the cut  $(W, \overline{W})$  is saturated. Indeed, if  $(x, y)$  is in the cut,  $x \in W$ ,  $y \notin W$ , then edge  $(x, y)$  is saturated, else  $y$  would have been labeled when we were scanning  $x$  and we would have  $y \in W$ , a contradiction. Similarly, if  $(y, x)$  is an edge where  $y \in \overline{W}$  and  $x \in W$ , then the flow  $f(y, x) = 0$ , else again  $y$  would have been labeled when we were scanning  $x$ , another contradiction.

Therefore, every edge from  $W$  to  $\overline{W}$  is carrying as much flow as its capacity permits, and every edge from  $\overline{W}$  to  $W$  is carrying no flow at all. Hence the sign of equality holds in (3.4.1), the value of the flow is equal to the capacity of the cut  $(W, \overline{W})$ , and the proof of theorem 3.4.1 is finished. ■

### 3.5 The complexity of the Ford-Fulkerson algorithm

The algorithm of Ford and Fulkerson terminates if and when it arrives at a stage where the sink is not labeled but no more vertices can be labeled. If at that time we let  $W$  be the set of vertices that have been labeled, then we have seen that  $(W, \overline{W})$  is a minimum cut of the network, and the present value of the flow is the desired maximum for the network.

The question now is, how long does it take to arrive at that stage, and indeed, is it guaranteed that we will *ever* get there? We are asking if the algorithm is *finite*, surely the most primitive complexity question imaginable.

First consider the case where every edge of the given network  $\mathbf{X}$  has *integer* capacity. Then during the labelling and flow augmentation algorithms, various additions and subtractions are done, but there is no way that any nonintegral flows can be produced.

It follows that the augmented flow is still integral. The *value* of the flow therefore increases by an integer amount during each augmentation. On the other hand if, say,  $C^*$  denotes the combined capacity of all edges that are outbound from the source, then it is eminently clear that the value of the flow can never exceed  $C^*$ . Since the value of the flow increases by at least 1 unit per augmentation, we see that no more than  $C^*$  flow augmentations will be needed before a maximum flow is reached. This yields

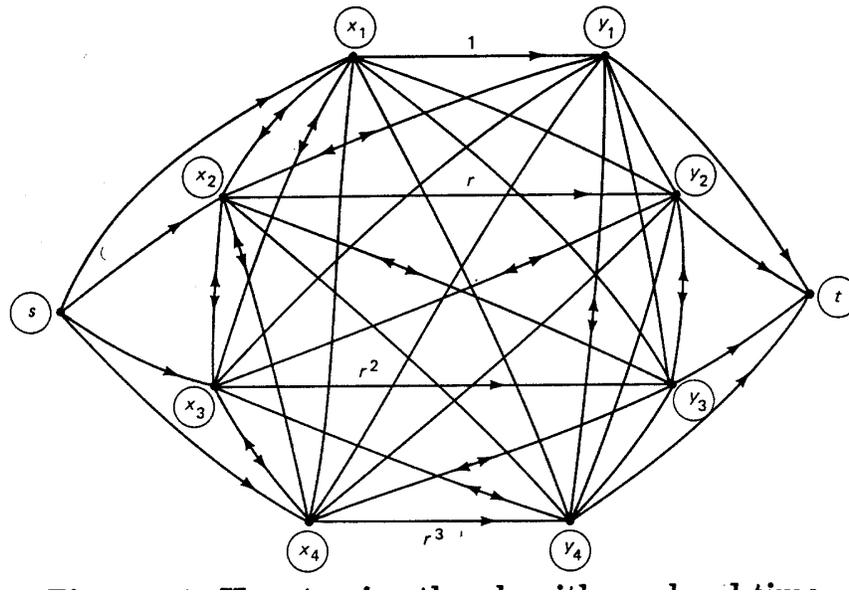
**Theorem 3.5.1.** *In a network with integer capacities on all edges, the Ford-Fulkerson algorithm terminates after a finite number of steps with a flow of maximum value.*

This is good news and bad news. The good news is that the algorithm is finite. The bad news is that the complexity estimate that we have proved depends not only on the numbers of edges and vertices in  $\mathbf{X}$ , but on the edge capacities. If the bound  $C^*$  represents the true behavior of the algorithm, rather than some weakness in our analysis of the algorithm, then even on very small networks it will be possible to assign edge capacities so that the algorithm takes a very long time to run.

And it *is* possible to do that.

We will show below an example due to Ford and Fulkerson in which the situation is even worse than the one envisaged above: not only will the algorithm take a very long time to run; it won't converge at all!

Consider the network  $\mathbf{X}$  that is shown in Fig. 3.5.1. It has 10 vertices  $s, t, x_1, \dots, x_4, y_1, \dots, y_4$ . There are directed edges  $(x_i, x_j) \forall i \neq j, (x_i, y_j) \forall i, j, (y_i, y_j) \forall i \neq j, (y_i, x_j) \forall i, j, (s, x_i) \forall i$ , and  $(y_j, t) \forall j$ .



**Fig. 3.5.1:** How to give the algorithm a hard time

In this network, the four edges  $A_i = (x_i, y_i)$  ( $i = 1, 4$ ) will be called the *special edges*. Next we will give the capacities of the edges of  $\mathbf{X}$ . Write  $r = (-1 + \sqrt{5})/2$ , and let

$$S = (3 + \sqrt{5})/2 = \sum_{n=0}^{\infty} r^n.$$

Then to every edge of  $\mathbf{X}$  except the four special edges we assign the capacity  $S$ . The special edges  $A_1, A_2, A_3, A_4$  are given capacities  $1, r, r^2, r^2$ , respectively (you can see that this is going to be interesting).

Suppose, for our first augmentation step, we find the flow augmenting path  $s \rightarrow x_1 \rightarrow y_1 \rightarrow t$ , and that we augment the flow by 1 unit along that path. The four special edges will then have *residual capacities* (excesses of capacity over flow) of  $0, r, r^2, r^2$ , respectively.

Inductively, suppose we have arrived at a stage of the algorithm where the four special edges, taken in some rearrangement  $A'_1, A'_2, A'_3, A'_4$ , have residual capacities  $0, r^n, r^{n+1}, r^{n+1}$ . We will now show that the algorithm might next do two flow augmentation steps the net result of which would be that the inductive state of affairs would again hold, with  $n$  replaced by  $n + 1$ .

Indeed, choose the flow augmenting path

$$s \rightarrow x'_2 \rightarrow y'_2 \rightarrow x'_3 \rightarrow y'_3 \rightarrow t.$$

The only special edges that are on this path are  $A'_2$  and  $A'_3$ . Augment the flow along this path by  $r^{n+1}$  units (the maximum possible amount).

Next, choose the flow augmenting path

$$s \rightarrow x'_2 \rightarrow y'_2 \rightarrow y'_1 \rightarrow x'_1 \rightarrow y'_3 \rightarrow x'_3 \rightarrow y'_4 \rightarrow t.$$

Notice that with respect to this path the special edges  $A'_1$  and  $A'_3$  are incoherently directed. Augment the flow along this path by  $r^{n+2}$  units, once more the largest possible amount.

The reader may now verify that the residual capacities of the four special edges are  $r^{n+2}$ , 0,  $r^{n+2}$ ,  $r^{n+1}$ . In the course of doing this verification it will be handy to use the fact that

$$r^{n+2} = r^n - r^{n+1} \quad (\forall n \geq 0).$$

These two augmentation steps together have increased the flow value by  $r^{n+1} + r^{n+2} = r^n$  units. Hence the flow in an edge will never exceed  $S$  units.

The algorithm converges to a flow of value  $S$ . Now comes the bad news: the maximum flow in this network has the value  $4S$  (find it!).

Hence, for this network

- (a) the algorithm does not halt after finitely many steps even though the edge capacities are finite and
- (b) the sequence of flow values converges to a number that is not the maximum flow in the network.

The irrational capacities on the edges may at first seem to make this example seem ‘cooked up.’ But the implication is that even with a network whose edge capacities are all integers, the algorithm might take a very long time to run.

Motivated by the importance and beauty of the theory of network flows, and by the unsatisfactory time complexity of the original algorithm, many researchers have attacked the question of finding an algorithm whose success is guaranteed within a time bound that is independent of the edge capacities, and depends only on the size of the network.

We turn now to the consideration of one of the main ideas on which further progress has depended, that of *layering* a network with respect to a flow function. This idea has triggered a whole series of improved algorithms. Following the discussion of layering we will give a description of one of the algorithms, the MPM algorithm, that uses layered networks and guarantees fast operation.

### 3.6 Layered networks

Layering a network is a technique that has the effect of replacing a single max-flow problem by several problems, each a good deal easier than the original. More precisely, in a network with  $V$  vertices we will find that we can solve a max-flow problem by solving at most  $V$  slightly different problems, each on a layered network. We will then discuss an  $O(V^2)$  method for solving each such problem on a layered network, and the result will be an  $O(V^3)$  algorithm for the original network flow problem.

Now we will discuss how to *layer a network with respect to a given flow function*. The purpose of the italics is to emphasize the fact that one does not just ‘layer a network.’ Instead, there is given a network  $\mathbf{X}$  and a flow function  $f$  for that network, and together they induce a layered network  $\mathbf{Y} = \mathbf{Y}(\mathbf{X}, f)$ , as follows.

First let us say that an edge  $e$  of  $\mathbf{X}$  is *helpful from  $u$  to  $v$*  if either  $e$  is directed from  $u$  to  $v$  and  $f(e)$  is below capacity or  $e$  is directed from  $v$  to  $u$  and the flow  $f(e)$  is positive.

Next we will describe the layered network  $\mathbf{Y}$ . Recall that in order to describe a network one must describe the vertices of the network, the directed edges, give the capacities of those edges, and designate the source and the sink. The network  $\mathbf{Y}$  will be constructed one layer at a time from the vertices of  $\mathbf{X}$ , using the flow  $f$  as a guide. For each layer, we will say which vertices of  $\mathbf{X}$  go into that layer, then we will say which vertices of the previous layer are connected to each vertex of the new layer. All of these edges will be directed from the earlier layer to the later one. Finally we will give the capacities of each of these new edges.

The  $0^{\text{th}}$  layer of  $\mathbf{Y}$  consists only of the source  $s$ . The vertices that comprise layer 1 of  $\mathbf{Y}$  will be every vertex  $v$  of  $\mathbf{X}$  such that in  $\mathbf{X}$  there is a helpful edge from  $s$  to  $v$ . We then draw an edge in  $\mathbf{Y}$  directed from  $s$  to  $v$  for each such vertex  $v$ . We assign to that edge in  $\mathbf{Y}$  a capacity  $\text{cap}(s, v) - f(s, v) + f(v, s)$ .

The set of all such  $v$  will be called layer 1 of  $\mathbf{Y}$ . Next we construct layer 2 of  $\mathbf{Y}$ . The vertex set of layer 2 consists of all vertices  $w$  that do not yet belong to any layer, and such that there is a helpful edge in  $\mathbf{X}$  from some vertex  $v$  of layer 1 to  $w$ .

Next we draw the edges from layer 1 to layer 2: for each vertex  $v$  in layer 1 we draw a single edge in  $\mathbf{Y}$  directed from  $v$  to every vertex  $w$  in layer 2 for which there is a helpful edge in  $\mathbf{X}$  from  $v$  to  $w$ .

Note that the edge always goes *from*  $v$  to  $w$  regardless of the direction of the helpful edge in  $\mathbf{X}$ . Note also that in contrast to the Ford-Fulkerson algorithm, even after an edge has been drawn from  $v$  to  $w$  in  $\mathbf{Y}$ , additional edges may be drawn to the same  $w$  from other vertices  $v'$ ,  $v''$  in layer 1.

Assign capacities to the edges from layer 1 to layer 2 in the same way as described above, that is, the capacity in  $\mathbf{Y}$  of the edge from  $v$  to  $w$  is  $cap(v, w) - f(v, w) + f(w, v)$ . This latter quantity is, of course, the total residual (unused) flow-carrying capacity of the edges in both directions between  $v$  and  $w$ .

The layering continues until we reach a layer  $L$  such that there is a helpful edge from some vertex of layer  $L$  to the sink  $t$ , or else until no additional layers can be created (to say that no more layers can be created is to say that among the vertices that haven't yet been included in the layered network that we are building, there aren't any that are adjacent to a vertex that is in the layered network, by a helpful edge).

In the former case, we then create a layer  $L + 1$  that consists solely of the sink  $t$ , we connect  $t$  by edges directed from the appropriate vertices of layer  $L$ , assign capacities to those edges, and the layering process is complete. Observe that not all vertices of  $\mathbf{X}$  need appear in  $\mathbf{Y}$ .

In the latter case, where no additional layers can be created but the sink hasn't been reached, the present flow function  $f$  in  $\mathbf{X}$  is maximum, and the network flow problem in  $\mathbf{X}$  has been solved.

Here is a formal statement of the procedure for layering a given network  $\mathbf{X}$  with respect to a given flow function  $f$  in  $\mathbf{X}$ . Input are the network  $\mathbf{X}$  and the present flow function  $f$  in that network. Output are the layered network  $\mathbf{Y}$ , and a logical variable *maxflow* that will be *True*, on output, if the flow is at a maximum value, *False* otherwise.

```

procedure layer ( $\mathbf{X}$ ,  $f$ ,  $\mathbf{Y}$ , maxflow);
{forms the layered network  $Y$  with respect to the flow  $f$  in  $\mathbf{X}$  }
{maxflow will be 'True' if the input flow  $f$  already has the
  maximum possible value for the network, else it will be 'False'}
 $L := 0$ ;  $layer(L) := \{source\}$ ; maxflow := false;
repeat
   $layer(L + 1) := \emptyset$ ;
  for each vertex  $u$  in  $layer(L)$  do
    for each vertex  $v$  such that  $\{layer(v) = L + 1$  or  $v$  is
      not in any layer $\}$  do
       $q := cap(u, v) - f(u, v) + f(v, u)$ ;
      if  $q > 0$  then do
        draw edge  $u \rightarrow v$  in  $\mathbf{Y}$ ;
        assign capacity  $q$  to that edge;
        assign vertex  $v$  to  $layer(L + 1)$ ;
   $L := L + 1$ 
  if  $layer(L)$  is empty then exit with maxflow := true;
until sink is in  $layer(L)$ ;
delete from  $layer(L)$  of  $Y$  all vertices other than sink,
  and remove their incident edges from  $Y$ 
end. $\{layer\}$ 

```

In Fig. 3.6.1 we show the typical appearance of a layered network. In contrast to a general network, in a layered network every path from the source to some fixed vertex  $v$  has the same number of edges in it (the number of the layer of  $v$ ), and all edges on such a path are directed the same way, from the source towards

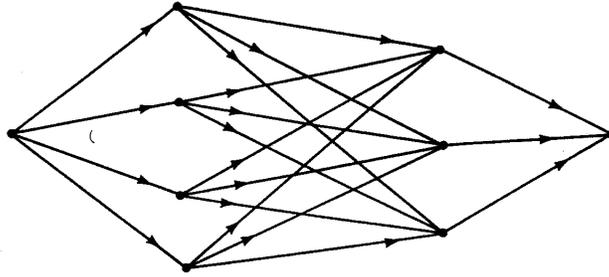


Fig. 3.6.1: A general layered network

*v.* These properties of layered networks are very friendly indeed, and make them much easier to deal with than general networks.

In Fig. 3.6.2 we show specifically the layered network that results from the network of Fig. 3.1.2 with the flow shown therein.

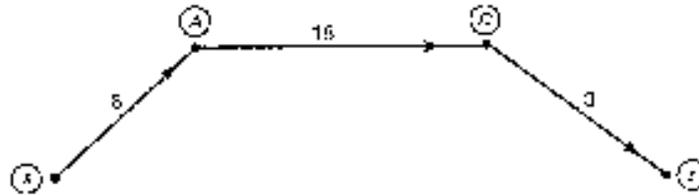


Fig. 3.6.2: A layering of the network in Fig. 3.1.2

The next question is this: exactly what problem would we like to solve on the layered network  $\mathbf{Y}$ , and what is the relationship of that problem to the original network flow problem in the original network  $\mathbf{X}$ ?

The answer is that in the layered network  $\mathbf{Y}$  we are looking for a *blocking flow*  $g$ . By a blocking flow we mean a flow function  $g$  in  $\mathbf{Y}$  such that *every path from source to sink in  $\mathbf{Y}$  has at least one saturated edge*.

This immediately raises two questions: (a) what can we do if we find a blocking flow in  $\mathbf{Y}$ ? (b) how can we find a blocking flow in  $\mathbf{Y}$ ? The remainder of this section will be devoted to answering (a). In the next section we will give an elegant answer to (b).

Suppose that we have somehow found a blocking flow function,  $g$ , in  $\mathbf{Y}$ . What we do with it is that we use it to augment the flow function  $f$  in  $\mathbf{X}$ , as follows.

```

procedure augment( $f$ ,  $\mathbf{X}$ ;  $g$ ,  $\mathbf{Y}$ );
{augment flow  $f$  in  $\mathbf{X}$  by using a blocking flow  $g$ 
  in the corresponding layered network  $\mathbf{Y}$ }
  for every edge  $e : u \rightarrow v$  of the layered network  $\mathbf{Y}$ ,
do
  increase the flow  $f$  in the edge  $u \rightarrow v$  of the
  network  $\mathbf{X}$  by the amount
     $\min\{g(e), \text{cap}(u \rightarrow v) - f(u \rightarrow v)\}$ ;
  if not all of  $g(e)$  has been used
    then decrease the flow in edge  $v \rightarrow u$  by
      the unused portion of  $g(e)$ 
end.{augment}

```

After augmenting the flow in the original network  $\mathbf{X}$ , what then? We construct a new layered network, from  $\mathbf{X}$  and the newly augmented flow function  $f$  on  $\mathbf{X}$ .

The various activities that are now being described may sound like some kind of thinly disguised repackaging of the Ford-Fulkerson algorithm, but they aren't just that, because here is what can be proved to happen:

First, if we start with zero flow in  $\mathbf{X}$ , make the layered network  $\mathbf{Y}$ , find a blocking flow in  $\mathbf{Y}$ , augment the flow in  $\mathbf{X}$ , make a new layered network  $\mathbf{Y}$ , find a blocking flow, etc. etc., then *after at most  $V$  phases* ('phase' = layer + block + augment) we will have found the maximum flow in  $\mathbf{X}$  and the process will halt.

Second, each phase can be done very rapidly. The MPM algorithm, to be discussed in section 3.7, finds a blocking flow in a layered network in time  $O(V^2)$ .

By the *height* of a layered network  $\mathbf{Y}$  we will mean the number of edges in any path from source to sink. The network of Fig. 3.6.1 has height 3. Let's now show

**Theorem 3.6.1.** *The heights of the layered networks that occur in the consecutive phases of the solution of a network flow problem form a strictly increasing sequence of positive integers. Hence, for a network  $X$  with  $V$  vertices, there can be at most  $V$  phases before a maximum flow is found.*

Let  $\mathbf{Y}(p)$  denote the layered network that is constructed at the  $p^{\text{th}}$  phase of the computation and let  $H(p)$  denote the height of  $\mathbf{Y}(p)$ . We will first prove

**Lemma 3.6.1.** *If*

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_m \quad (v_0 = \text{source})$$

*is a path in  $\mathbf{Y}(p+1)$ , and if every vertex  $v_i$  ( $i = 1, m$ ) of that path also appears in  $\mathbf{Y}(p)$ , then for every  $a = 0, m$  it is true that if vertex  $v_a$  was in layer  $b$  of  $\mathbf{Y}(p)$  then  $a \geq b$ .*

**Proof of lemma:** The result is clearly true for  $a = 0$ . Suppose it is true for  $v_0, v_1, \dots, v_a$ , and suppose  $v_{a+1}$  was in layer  $c$  of network  $\mathbf{Y}(p)$ . We will show that  $a + 1 \geq c$ . Indeed, if not then  $c > a + 1$ . Since  $v_a$ , by induction, was in a layer  $\leq a$ , it follows that the edge

$$e^* : v_a \rightarrow v_{a+1}$$

was not present in network  $\mathbf{Y}(p)$  since its two endpoints were not in two consecutive layers. Hence the flow in  $\mathbf{Y}$  between  $v_a$  and  $v_{a+1}$  could not have been affected by the augmentation procedure of phase  $p$ . But edge  $e^*$  is in  $\mathbf{Y}(p+1)$ . Therefore it represented an edge of  $\mathbf{Y}$  that was helpful from  $v_a$  to  $v_{a+1}$  at the beginning of phase  $p+1$ , was unaffected by phase  $p$ , but was not helpful at the beginning of phase  $p$ . This contradiction establishes the lemma. ■

Now we will prove the theorem. Let

$$s \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{H(p+1)-1} \rightarrow t$$

be a path from source to sink in  $\mathbf{Y}(p+1)$ .

Consider first the case where every vertex of the path also lies in  $\mathbf{Y}(p)$ , and apply the lemma to  $v_m = t$  ( $m = H(p+1)$ ),  $a = m$ . We conclude at once that  $H(p+1) \geq H(p)$ . Now we want to exclude the '=' sign. If  $H(p+1) = H(p)$  then the entire path is in  $\mathbf{Y}(p)$  and in  $\mathbf{Y}(p+1)$ , and so all of the edges in  $\mathbf{Y}$  that the edges of the path represent were helpful both before and after the augmentation step of phase  $p$ , contradicting the fact that the blocking flow that was used for the augmentation saturated some edge of the chosen path. The theorem is now proved for the case where the path had all of its vertices in  $\mathbf{Y}(p)$  also.

Now suppose that this was not the case. Let  $e^* : v_a \rightarrow v_{a+1}$  be the first edge of the path whose terminal vertex  $v_{a+1}$  was not in  $\mathbf{Y}(p)$ . Then the corresponding edge(s) of  $\mathbf{Y}$  was unaffected by the augmentation in phase  $p$ . It was helpful from  $v_a$  to  $v_{a+1}$  at the beginning of phase  $p+1$  because  $e^* \in \mathbf{Y}(p+1)$  and it was unaffected by phase  $p$ , yet  $e^* \notin \mathbf{Y}(p)$ . The only possibility is that vertex  $v_{a+1}$  would have entered into  $\mathbf{Y}(p)$

in the layer  $H(p)$  that contains the sink, but that layer is special, and contains only  $t$ . Hence, if  $v_a$  was in layer  $b$  of  $\mathbf{Y}(p)$ , then  $b+1 = H(p)$ . By the lemma once more,  $a \geq b$ , so  $a+1 \geq b+1 = H(p)$ , and therefore  $H(p+1) > H(p)$ , completing the proof of theorem 3.6.1. ■

To summarize, if we want to find a maximum flow in a given network  $\mathbf{Y}$  by the method of layered networks, we carry out

```

procedure maxflow ( $\mathbf{X}, \mathbf{Y}, f$ );
set the flow function  $f$  to zero on all edges of  $\mathbf{Y}$ ;
repeat
    (i) construct the layered network  $\mathbf{Y} = \mathbf{Y}(\mathbf{X}, f)$ 
        if possible, else exit with flow at maximum
        value;
    (ii) find a blocking flow  $g$  in  $\mathbf{Y}$ ;
    (iii) augment the flow  $f$  in  $\mathbf{Y}$  with the blocking
        flow  $g$ , by calling procedure augment above
until exit occurs in (i) above;
end.{maxflow}

```

According to theorem 3.6.1, the procedure will repeat steps (i), (ii), (iii) at most  $V$  times because the height of the layered network increases each time around, and it certainly can never exceed  $V$ . The labor involved in step (i) is certainly  $O(E)$ , and so is the labor in step (iii). Hence if BFL denotes the labor involved in some method for finding a blocking flow in a layered network, then the whole network flow problem can be done in time  $O(V \cdot (E + \text{BFL}))$ .

The idea of layering networks is due to Dinic. Since his work was done, all efforts have been directed at the problem of reducing BFL as much as possible.

### 3.7 The MPM algorithm

Now we suppose that we are given a *layered* network  $\mathbf{Y}$  and we want to find a blocking flow in  $\mathbf{Y}$ . The following ingenious suggestion is due to Malhotra, Pramodh-Kumar and Maheshwari.

Let  $V$  be some vertex of  $\mathbf{Y}$ . The *in-potential* of  $v$  is the sum of the capacities of all edges directed into  $v$ , and the *outpotential* of  $v$  is the total capacity of all edges directed out from  $v$ . The *potential* of  $v$  is the smaller of these two.

(A) Find a vertex  $v$  of smallest potential, say  $P^*$ . Now we will push  $P^*$  more units of flow from source to sink, as follows.

(B) (Pushout) Take the edges that are outbound from  $v$  in some order, and saturate each one with flow, unless and until saturating one more would lift the total flow used over  $P^*$ . Then assign all remaining flow to the next outbound edge (not necessarily saturating it), so the total outflow from  $v$  becomes exactly  $P^*$ .

(C) Follow the flow to the next higher layer of  $\mathbf{Y}$ . That is, for each vertex  $v'$  of the next layer, let  $h(v')$  be the flow into  $v'$ . Now saturate all except possibly one outbound edge of  $v'$ , to pass through  $v'$  the  $h(v')$  units of flow. When all vertices  $v'$  in that layer have been done, repeat for the next layer, etc. We never find a vertex with insufficient capacity, in or out, to handle the flow that is thrust upon it, because we began by choosing a vertex of minimum potential.

(D) (Pullback) When all layers 'above'  $v$  have been done, then follow the flow to the next layer 'below'  $v$ . For each vertex  $v'$  of that layer, let  $h(v')$  be the flow out of  $v'$  to  $v$ . Then saturate all except possibly one *incoming* edge of  $v'$ , to pass through  $v'$  the  $h(v')$  units of flow. When all  $v'$  in that layer have been done, proceed to the next layer below  $v$ , etc.

(E) (Update capacities) The flow function that has just been created in the layered network must be stored somewhere. A convenient way to keep it is to carry out the augmentation procedure back in the network  $\mathbf{X}$  at this time, thereby, in effect 'storing' the contributions to the blocking flow in  $\mathbf{Y}$  in the flow array for  $\mathbf{X}$ . This can be done concurrently with the MPM algorithm as follows: Every time we increase the flow in some edge  $u \rightarrow v$  of  $\mathbf{Y}$  we do it by augmenting the flow from  $u$  to  $v$  in  $\mathbf{X}$ , and then decreasing the capacity of edge  $u \rightarrow v$  in  $\mathbf{Y}$  by the same amount. In that way the capacities of the edges in  $\mathbf{Y}$  will always be the updated *residual* capacities, and the flow function  $f$  in  $\mathbf{X}$  will always reflect the latest augmentation of the flow in  $\mathbf{Y}$ .

(F) (Prune) We have now pushed the original  $h(v)$  units of flow through the whole layered network. We intend to repeat the operation on some other vertex  $v$  of minimum potential, but first we can prune off of the network some vertices and edges that are guaranteed never to be needed again.

The vertex  $v$  itself has either all incoming edges or all outgoing edges, or both, at zero residual capacities. Hence no more flow will ever be pushed through  $v$ . Therefore we can delete  $v$  from the network  $\mathbf{Y}$  together with all of its incident edges, incoming or outgoing. Further, we can delete from  $\mathbf{Y}$  all of the edges that were saturated by the flow pushing process just completed, *i.e.*, all edges that now have zero residual capacity.

Next, we may now find that some vertex  $w$  has had all of its incoming or all of its outgoing edges deleted. That vertex will never be used again, so delete it and any other incident edges it may still have. Continue the pruning process until only vertices remain that have nonzero potential. If the source and the sink are still connected by some path, then repeat from (A) above.

Else the algorithm halts. The blocking flow function  $g$  that we have just found is the following: if  $e$  is an edge of the input layered network  $\mathbf{Y}$ , then  $g(e)$  is the sum of all of the flows that were pushed through edge  $e$  at all stages of the above algorithm.

It is obviously a blocking flow: since no path between  $s$  and  $t$  remains, every path must have had at least one of its edges saturated at some step of the algorithm. What is the complexity of this algorithm? Certainly we delete at least one vertex from the network at every pruning stage, because the vertex  $v$  that had minimum potential will surely have had either all of its incoming or all of its outgoing edges (or both) saturated.

It follows that steps (A)–(E) can be executed at most  $V$  times before we halt with a blocking flow. The cost of saturating all edges that get saturated, since every edge has but one saturation to give to its network, is  $O(E)$ . The number of partial edge-saturation operations is at most two per vertex visited. For each minimal-potential vertex  $v$  we visit at most  $V$  other vertices, so we use at most  $V$  minimal-potential vertices altogether. So the partial edge saturation operations cost  $O(V^2)$  and the total edge saturations cost  $O(E)$ .

The operation of finding a vertex of minimum potential is ‘free,’ in the following sense. Initially we compute and store the in- and out- potentials of every vertex. Thereafter, each time the flow in some edge is increased, the outpotential of its initial vertex and the inpotential of its terminal vertex are reduced by the same amount. It follows that the cost of maintaining these arrays is linear in the number of vertices,  $V$ . Hence it affects only the constants implied by the ‘big oh’ symbols above, but not the orders of magnitude.

The total cost is therefore  $O(V^2)$  for the complete MPM algorithm that finds a blocking flow in a layered network. Hence a maximum flow in a network can be found in  $O(V^3)$  time, since at most  $V$  layered networks need to be looked at in order to find a maximum flow in the original network.

In contrast to the nasty example network of section 3.5, with its irrational edge capacities, that made the Ford-Fulkerson algorithm into an infinite process that converged to the wrong answer, the time bound  $O(V^3)$  that we have just proved for the layered-network MPM algorithm is totally independent of the edge capacities.

### 3.8 Applications of network flow

We conclude this chapter by mentioning some applications of the network flow problem and algorithm.

Certainly, among these, one most often mentions first the problem of maximum matching in a bipartite graph. Consider a set of  $P$  people and a set of  $J$  jobs, such that not all of the people are capable of doing all of the jobs.

We construct a graph of  $P + J$  vertices to represent this situation, as follows. Take  $P$  vertices to represent the people,  $J$  vertices to represent the jobs, and connect vertex  $p$  to vertex  $j$  by an undirected edge if person  $p$  can do job  $j$ . Such a graph is called *bipartite*. In general a graph  $G$  is bipartite if its vertices can be partitioned into two classes in such a way that no edge runs between two vertices of the same class (see section 1.6).

In Fig. 3.8.1 below we show a graph that might result from a certain group of 8 people and 9 jobs.

The maximum matching problem is just this: assuming that each person can handle at most one of the jobs, and that each job needs only one person, assign people to the jobs in such a way that the largest possible number of people are employed. In terms of the bipartite graph  $G$ , we want to find a *maximum number of edges, no two incident with the same vertex*.

To solve this problem by the method of network flows we construct a network  $\mathbf{Y}$ . First we adjoin two new vertices  $s, t$  to the bipartite graph  $G$ . If we let  $P, J$  denote the two classes of vertices in the graph  $G$ , then we draw an edge from  $s$  to each  $p \in P$  and an edge from each  $j \in J$  to  $t$ . Each edge in the network is

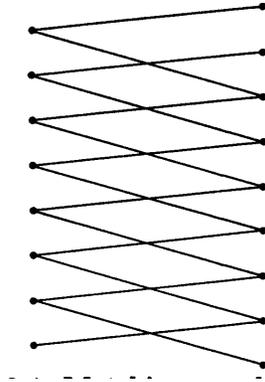


Fig. 3.8.1: Matching people to jobs

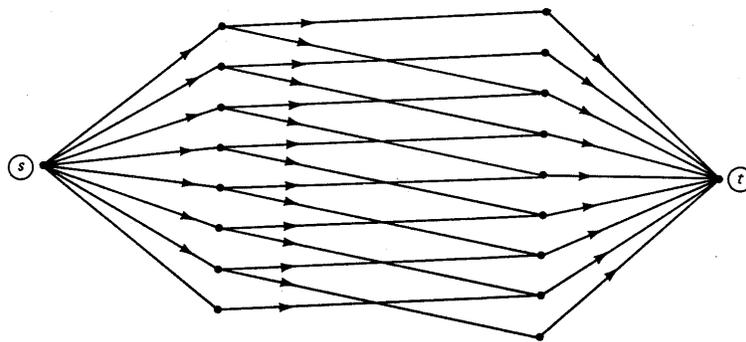


Fig. 3.8.2: The network for the matching problem

given capacity 1. The result for the graph of Fig. 3.8.1 is shown in Fig. 3.8.2.

Consider a maximum integer-valued flow in this network, of value  $Q$ . Since each edge has capacity 1,  $Q$  edges of the type  $(s, p)$  each contain a unit of flow. Out of each vertex  $p$  that receives some of this flow there will come one unit of flow (since inflow equals outflow at such vertices), which will then cross to a vertex  $j$  of  $J$ . No such  $j$  will receive more than one unit because at most one unit can leave it for the sink  $t$ . Hence the flow defines a matching of  $Q$  edges of the graph  $G$ . Conversely, any matching in  $G$  defines a flow, hence a maximum flow corresponds to a maximum matching. In Fig. 3.8.3 we show a maximum flow in the network of Fig. 3.8.2 and therefore a maximum matching in the graph of Fig. 3.8.1.

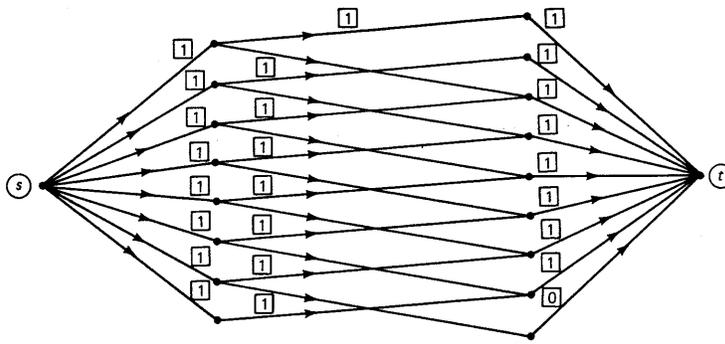


Fig. 3.8.3: A maximum flow

For a second application of network flow methods, consider an undirected graph  $G$ . The *edge-connectivity* of  $G$  is defined as the smallest number of edges whose removal would disconnect  $G$ . Certainly, for instance,

if we remove all of the edges incident to a single vertex  $v$ , we will disconnect the graph. Hence the edge connectivity cannot exceed the minimum degree of vertices in the graph. However the edge connectivity could be a lot smaller than the minimum degree as the graph of Fig. 3.8.4 shows, in which the minimum is large, but the removal of just one edge will disconnect the graph.

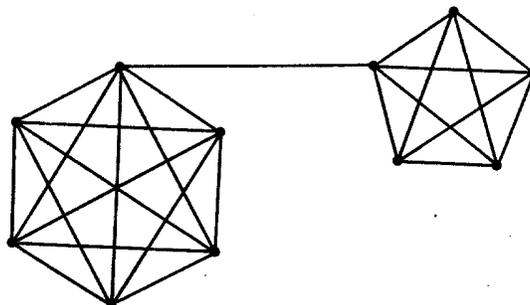


Fig. 3.8.4: Big degree, low connectivity

Finding the edge connectivity is quite an important combinatorial problem, and it is by no means obvious that network flow methods can be used on it, but they can, and here is how.

Given  $G$ , a graph of  $V$  vertices. We solve not just one, but  $V - 1$  network flow problems, one for each vertex  $j = 2, \dots, V$ .

Fix such a vertex  $j$ . Then consider vertex 1 of  $G$  to be the source and vertex  $j$  to be the sink of a network  $\mathbf{X}_j$ . Replace each edge of  $G$  by two edges of  $\mathbf{X}_j$ , one in each direction, each with capacity 1. Now solve the network flow problem in  $\mathbf{X}_j$  obtaining a maximum flow  $Q(j)$ . Then the smallest of the numbers  $Q(j)$ , for  $j = 2, \dots, V$  is the edge connectivity of  $G$ . We will not prove this here.\*

As a final application of network flow we discuss the beautiful question of determining whether or not there is a matrix of 0's and 1's that has given row and column sums. For instance, is there a  $6 \times 8$  matrix whose row sums are respectively (5, 5, 4, 3, 5, 6) and whose column sums are (3, 4, 4, 4, 3, 3, 4, 3)? Of course the phrase 'row sums' means the same thing as 'number of 1's in each row' since we have said that the entries are only 0 or 1.

Hence in general, let there be given a row sum vector  $(r_1, \dots, r_m)$  and a column sum vector  $(s_1, \dots, s_n)$ . We ask if there exists an  $m \times n$  matrix  $A$  of 0's and 1's that has exactly  $r_i$  1's in the  $i$ th row and exactly  $s_j$  1's in the  $j$ th column, for each  $i = 1, \dots, m$ ,  $j = 1, \dots, n$ . The reader will no doubt have noticed that for such a matrix to exist it must surely be true that

$$r_1 + \dots + r_m = s_1 + \dots + s_n \quad (3.8.1)$$

since each side counts the total number of 1's in the matrix. Hence we will suppose that (3.8.1) is true.

Now we will construct a network  $\mathbf{Y}$  of  $m + n + 2$  vertices named  $s, x_1, \dots, x_m, y_1, \dots, y_n$ , and  $t$ . There is an edge of capacity  $r_i$  drawn from the source  $s$  to vertex  $x_i$ , for each  $i = 1, \dots, m$ , and an edge of capacity  $s_j$  drawn from vertex  $y_j$  to the sink  $t$ , for each  $j = 1, \dots, n$ . Finally, there are  $mn$  edges of capacity 1 drawn from each edge  $x_i$  to each vertex  $y_j$ .

Next find a maximum flow in this network. Then there is a 0-1 matrix with the given row and column sum vectors if and only if a maximum flow saturates every edge outbound from the source, that is, if and only if a maximum flow has value equal to the right or left side of equation (3.8.1). If such a flow exists then a matrix  $A$  of the desired kind is constructed by putting  $a_{i,j}$  equal to the flow in the edge from  $x_i$  to  $y_j$ .

---

\* S. Even and R. E. Tarjan, Network flow and testing graph connectivity, SIAM J. Computing 4 (1975), 507-518.

### Exercises for section 3.8

1. Apply the max-flow min-cut theorem to the network that is constructed in order to solve the bipartite matching problem. Precisely what does a cut correspond to in this network? What does the theorem tell you about the matching problem?
2. Same as question 1 above, but applied to the question of discovering whether or not there is a 0-1 matrix with a certain given set of row and column sums.

### Bibliography

The standard reference for the network flow problem and its variants is

L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1974.

The algorithm, the example of irrational capacities and lack of convergence to maximum flow, and many applications are discussed there. The chronology of accelerated algorithms is based on the following papers. The first algorithms with a time bound independent of the edge capacities are in

J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *JACM* **19**, 2 (1972), 248-264.

E. A. Dinic, Algorithm for solution of a problem of maximal flow in a network with power estimation, *Soviet Math. Dokl.*, **11** (1970), 1277-1280.

The paper of Dinic, above, also originated the idea of a layered network. Further accelerations of the network flow algorithms are found in the following.

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

B. V. Cherkassky, Algorithm of construction of maximal flow in networks with complexity of  $O(V^2E)$  operations, *Akad. Nauk. USSR, Mathematical methods for the solution of economical problems* **7** (1977), 117-126.

The MPM algorithm, discussed in the text, is due to

V. M. Malhotra, M. Pramodh-Kumar and S. N. Maheshwari, An  $O(V^3)$  algorithm for finding maximum flows in networks, *Information processing Letters* **7**, (1978), 277-278.

Later algorithms depend on refined data structures that save fragments of partially constructed augmenting paths. These developments were initiated in

Z. Galil, A new algorithm for the maximal flow problem, *Proc. 19th IEEE Symposium on the Foundations of Computer Science*, Ann Arbor, October 1978, 231-245.

Andrew V. Goldberg and Robert E. Tarjan, A new approach to the maximum flow problem, 1985.

A number of examples that show that the theoretical complexity estimates for the various algorithms cannot be improved are contained in

Z. Galil, On the theoretical efficiency of various network flow algorithms, IBM report RC7320, September 1978.

The proof given in the text, of theorem 3.6.1, leans heavily on the one in

Shimon Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

If edge capacities are all 0's and 1's, as in matching problems, then still faster algorithms can be given, as in

S. Even and R. E. Tarjan, Network flow and testing graph connectivity, *SIAM J. Computing* **4**, (1975), 507-518.

If every pair of vertices is to act, in turn, as source and sink, then considerable economies can be realized, as in

R. E. Gomory and T. C. Hu, Multiterminal network flows, *SIAM Journal*, **9** (1961), 551-570.

Matching in general graphs is much harder than in bipartite graphs. The pioneering work is due to J. Edmonds, Path, trees, and flowers, *Canadian J. Math.* **17** (1965), 449-467.