CHAPTER

ELEVEN

Network Flows

11.1 Basic Notation and Terminology

The basic setup for a network flow problem begins with an oriented graph G, called a *network* in which we have two special vertices called respectively the *source* and the *sink*. We use the letter S to denote the source, while the letter T is used to denote the sink (terminus). All edges incident with the source are oriented away from the source, while all edges incident with the sink are oriented with the sink. Futhermore, on each edge, we have a non-negative *capacity*. The capacity of the edge $e = (x, y)$ is denoted $c(e)$ or by $c(x, y)$. In a computer program, the nodes of a network may be identified with integer keys, but in our course, we will typically use letters in labeling the nodes of a network. This helps to distinguish nodes from capacities in diagrams of networks. We illustrate a network in Figure 11.1.

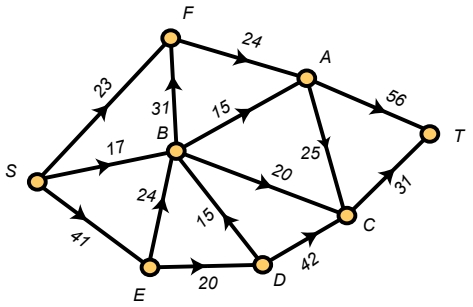

FIGURE 11.1: A NETWORK

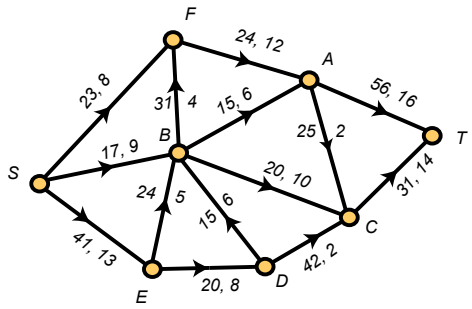

FIGURE 11.2: A NETWORK FLOW

A *flow* ϕ in a network is a function which assigns to each directed edge $e = (x, y)$ a non-negative value $\phi(e) = \phi(x, y)$ so that the following "conservation" laws hold:

1. $\sum_x \phi(S, x) = \sum_x \phi(x, T)$, i.e., the amount leaving the source is equal to the amount arriving at the sink. This quantity is called the *value* of the flow ϕ.

2. For every vertex y which is neither the source nor the sink, $\sum_x \phi(x, y) = \sum_x \phi(y, x)$, i.e., the amount leaving y is equal to the amount entering y.

We illustrate a flow in a network in Figure 11.2. The value of the flow is 30.

Remark 11.1. Given a network, it is very easy to find a flow. We simply assign $\phi(e) = 0$ for every edge e. It is very easy to *underestimate* the importance of this observation—as in general, it may be very difficult to find a feasible solution to a linear programming problem. And conceptually, finding a feasible solution—*any* solution—is just as hard as finding an *optimal* solution.

11.2 Flows and Cuts

Given a network, it is natural to ask what is the maximum flow, i.e., what is the largest number v_0 so that there exists a flow ϕ of value v_0. Of course, we not only want to find the maximum value v_0, but we also want to find a flow ϕ having this value. Although it may seem a bit surprising, we will develop an efficient algorithm which (a) finds a flow of maximum value, and (b) finds a certificate verifying the claim of optimality. This certificate makes use of the following important concept.

A partition $V = L \cup U$ of the vertex set V of a network into two non-empty subsets with $S \in L$ and $T \in U$ is called a *cut*. The *capacity* of a cut $V = L \cup U$, denoted $c(L, U)$,

is defined by

$$c(L,U) = \sum_{x\in L, y\in U} c(x,y)$$

Note that in computing the capacity of the cut $V = L \cup U$, we only add the capacities of the edges from L to U. We don't include the edges from U to L in this sum.

The relationship between flows and cuts rests on the following fundamentally important theorem.

Theorem 11.2. *Let $G = (V, E)$ be a network, let ϕ be a flow in G and let $V = L \cup U$ be a cut. Then the value of the flow is at most as large as the capacity of the cut.*

Proof. Let ϕ be a flow of value v_0 and let $V = L \cup U$ be a cut. Also, let c_0 denote the capacity of this cut. Then

$$\begin{aligned}
v_0 &= \sum_x \phi(S,x) \\
&= \sum_x \phi(S,x) + \sum_{y\neq S,T} \big(\sum_x \phi(x,y) - \phi(x,y)\big) \\
&= \sum_{x\in L, y\in U} \phi(x,y) - \phi(y,x) \\
&\leq \sum_{x\in L, y\in U} \phi(x,y) \\
&\leq \sum_{x\in L, y\in U} c(x,y) \\
&= c_0
\end{aligned}$$

□

So the surprise of this chapter is that the inequality in the preceding theorem is tight.

Theorem 11.3. The Max Flow–Min Cut Theorem *Let $G = (V, E)$ be a network. Then let v_0 be the maximum value of a flow, and let c_0 be the minimum capacity c_0 of a cut. Then $v_0 = c_0$.*

11.3 Augmenting Paths and the Labeling Algorithm

In this section, we develop the classic labeling algorithm of Ford and Fulkerson which starts with any flow in a network and proceeds to modify the flow—always increasing (or at least never decreasing) the value of the flow—until reaching a step where no further improvements are possible. At this stage, the algorithm will identify a cut whose capacity is the the same as the value of the current flow.

Our presentation of the labeling algorithm of the labeling algorithm makes use of some natural and quite descriptive terminology. Suppose we have a network $G = (V, E)$ with a flow ϕ of value v. We call ϕ the *current* flow and look for ways to *augment* ϕ by making a relatively small number of changes. An edge $e = (x, y)$ with $\phi(x, y) > 0$ is said to be *used*, and when $\phi(x, y) = c(x, y) > 0$, we say the edge is *full*. When $\phi(x, y) < c(x, y)$, we say the edge $e = (x, y)$ has *spare capacity*, and when $0 = \phi(x, y) < c(x, y)$, we say the edge $e = (x, y)$ is *empty*. Note that we simply ignore edges with zero capacity.

An augmenting path is then a sequence $P = (x_0, x_1, \ldots, x_m)$ of distinct vertices in the network such that $x_0 = S$, $x_m = t$, and for each $i = 1, 2, \ldots, m$, either (a) (x_{i-1}, x_i) has spare capacity or (b) (x_i, x_{i-1}) is used. When condition (a) holds, it is customary to refer to the edge (x_{i-1}, x_i) as a *forward* edge of the augmenting path P. Similarly, if condition (b) holds, then the edge (x_{i-1}, x_i) is called a *backward* edge.

If we are able—by any method whatsoever—to find an augmenting path, then it is relatively easy to modify the current flow into one with value $v + \delta$ where $\delta > 0$. Here's how. First, let δ_1 be the positive number defined by:

$$\delta_1 = \min\{c(x_{i-1}, x_i) - \phi(x_{i-1}, x_i) : (x_{i-1}, x_i) \text{ a foward edge of } P\}$$

Note that the edges (x_0, x_1) and (x_{m-1}, x_m) are always forward edges. So the positive quantity δ_1 is defined for every augmenting path.

When the augmenting path P has no backward edges, we set $\delta = \delta_1$. But when P has one or more backward edges, we pause to set

$$\delta_2 = \min\{\phi(x_i, x_{i-1}) : (x_{i-1}, x_i) \text{ a backward edge of } P\}$$

We then set $\delta = \min\{\delta_1, \delta_2\}$.

In either case, we now have a positive number δ and we make the following elementary observation.

Proposition 11.4. *Suppose we have an augmenting path $P = (x_0, x_1, \ldots, x_m)$ with $\delta > 0$ calculated as above. Modify the flow ϕ by changing the values along the edges of the path P by an amount which is either $+\delta$ or $-\delta$ according to the following rule:*

1. *Increase the flow along the edges of P which are forwards, and*
2. *Decrease the flow along the edges of P which are backwards.*

Then the resulting function $\hat{\phi}$ is a flow and it has value $v + \delta$.

Example 11.5. The network flow shown in Figure FIX ME!! has many augmenting paths. Here are some of them:

1. $P_1 = (S, F, A, T)$ with $\delta = 12$. All edges are forward.
2. $P_2 = (S, B, A, T)$ with $\delta = 8$. All edges are forward.

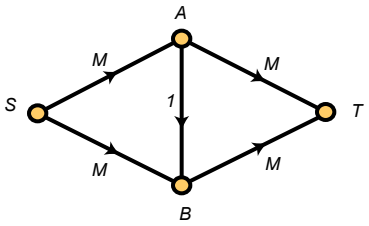

FIGURE 11.3: A SMALL NETWORK

3. $P_3 = (S, E, D, C, B, A, T)$ with $\delta = 9$. All edges are forward, except (C, B) which is backward.

4. $P_4 = (S, B, E, D, C, A, T)$ with $\delta = 2$. All edges are forward, except (B, E) and (C, A) which are backward.

11.3.1 Caution on Augmenting Paths

Suppose a kind friend promises to tell you about the existence of an augmenting path—provided there is one. Should you be grateful? The surprising answer is: Not necessarily.

The reason is that it is possible to identify augmenting paths which make incremental progress but in some sense don't make enough of a difference to matter. In making this statement, it should be stressed that what is enough to matter is all in the eyes of the beholder, since it may be very difficult to quantify what is a "good" increase. But doesn't it make sense to say that we would like to have an algorithm whose performance depends primarily on the size of the network and not on the values of the capacities—apart from the inherent difficulty of reading large numbers?

Here is an elementary example that you can find in almost any textbook on network flows or optimization in general.

Example 11.6. Let M be a large positive integer. Then consider the following network on four vertices:

Clearly the maximum value of a flow in this network is $2M$. This value is achieved by the flow with $\phi(S, A) = M$, $\phi(A, T) = M$, $\phi(S, B) = M$, $\phi(B, T) = M$ and $\phi(A, B) = 0$. But let's see how we would find this obvious answer using augmenting flows.

Set $i = 0$ and start with the zero flow, i.e., set $\phi_0(e) = 0$ for every edge e. Then for the value $i = 0$, we have a flow ϕ_{2i} of value $2i$ with $\phi_{2i}(S, A) = i$, $\phi_{2i}(S, B) = i$, $\phi_{2i}(A, T) = i$ and $\phi_{2i}(B, T) = i$. Also, $\phi_{2i}(A, B) = 0$. It follows that $P = (S, A, B, T)$ is an augmenting path with all edges forward, resulting in $\delta = 0$. Increasing the flow by 1 along all edges of the path results in a flow ϕ_{2i+1} of value $2i + 1$ in which $\phi_{21+1}(A, B) = 2i + 1$,

$\phi_{2i+1}(A,T) = 2i$, $\phi_{2i+1}(S,B) = 2i$, $\phi_{2i+1}(B,T) = 2i+1$ and $\phi_{2i+1}(A,B) = 1$. Now the path $P = (S,B,A,T)$ is an augmenting path with edge (B,A) being backward. We increase the flow by 1 along the foward edges (S,B) and (A,T) while decreasing the flow along the backward edge (B,A) to obtain the flow ϕ_{2i+2}. Clearly the process halts in $2M$ steps, i.e., the number of augmentation may be arbitrarily large and depends on the capacities of the edges *regardless* of the size of the network.

This elementary example is actually quite revealing. It suggests that shorter augmenting paths may be preferable to longer ones, and indeed that is the case!

11.4 The Ford-Fulkerson Labeling Algorithm

In this section, we outline the classic Ford-Fulkerson labeling algorithm for finding a maximum flow in a network. The algorithm begins with a linear order on the vertex set which establishes a notion of *precedence*. Typically, the first vertex in this linear order is the source while the second is the sink. After that, the vertices can be listed in any order. In our course, we will use the following convention. The vertices will be labeled with capital letters of the English alphabet, and the linear order will be $(S,T,A,B,C,D,E,F,G,\dots)$, and we will refer to this as the *pseudo-alphabetic* order. Of course, this convention only makes sense for networks with at most 26 vertices, but this limitation will be not cramp our style. For real world problems, we take comfort in the fact that computers can deal quite easily with integer keys of just about any size.

In carrying out the labeling algorithm, vertices will be classified as either *labeled* or *unlabeled*. At first, we will start with only the source being labeled while all other vertices will be unlabeled. By criteria yet to be spelled out, we will systematically consider unlabeled vertices and determine which should be labeled. If we ever label the sink, then we will have discovered an augmenting path, and the flow will be suitably updated.

After updating the flow, we start over again with just the sink being labeled.

This process is repeated until (and we will see that this always occurs) we reach a point where the labeling halts with some vertices labeled (one of these is the source) and some vertices unlabeled (one of these is the sink). We will then note that the partition $V = L \cup U$ into labeled and unlabeled vertices is a cut whose capacity is exactly equal to the value of the current flow. This is the certificate we seek, as it justifies the optimality of the flow we now have in hand.

Labeling the Vertices. Vertices will be labeled with ordered triples of symbols. Each time we start the labeling process, we begin by labeling the source with the triple $(*,+,\infty)$. The rules by which we label vertices will be explicit.

Potential on a Labeled Vertex. Let u be a labeled vertex. The third coordinate of the label given to u will be positive real number—although it may be infinite. We call this quantity the *potential* on u and denote it by $p(u)$. Note that the potential on the source is

First Labeled, First Scanned. The labeling algorithm involves a scan from a vertex u which is always a labeled vertex. As the vertices are labeled, they determine another linear order. The source will always be the first vertex in this order. After that, the order in which vertices are labeled will change with time. But the important rule is that we scan vertices in the order that they are labeled—until we label the sink. If for example, the initial scan—always done from the source—results in labels being applied to vertices D, G and M, then we next scan from vertex D. If that scan results in vertices B, F, G and Q being labeled, then we next scan from G, as it was labeled before B, even though B precedes G in the pseudo-alphabetic order. This aspect of the algorithm results in a *breadth-first* search of the vertices looking for ways to label previously unlabeled vertices.

Never Relabel a Vertex. Once a vertex is labeled, we do not change its label. We are content to label previously unlabeled vertices—up until the time where we label the sink. Then, after updating the flow and increasing the value, all labels, except of course the special label on the source, are discarded and we start all over again.

Labeling Vertices Using Forward Edges Suppose we are scanning from a labeled vertex u with potential $p(u) > 0$. From u, we consider the unlabeled neighbors of u in pseudo-alphabetic order. Now suppose that we are looking at a neighbor v of u with the edge (u, v) belonging to the network. This means that the edge is directed from u to v. If $e = (u, v)$ is not full, then we label the vertex v with the triple $(u, +, a)$ where $a = \min\{p(u), c(e) - \phi(e)\}$. Note that the potential $a = p(v)$ is positive since a is the minimum of two positive numbers.

Labeling Vertices Using Backward Edges Now suppose that we are looking at a neighbor v of u with the edge (v, u) belonging to the network. This means that the edge is directed from v to u. If $e = (v, u)$ is used, then we label the vertex v with the triple $(u, -, a)$ where $a = \min\{p(u), \phi(e)\}$. Again, note that the potential $a = p(v)$ is positive since a is the minimum of two positive numbers.

What Happens When the Sink is Labeled? The labeling algorithm halts if the sink is ever labeled. Note that we are always trying our best to label the sink since in each scan, the sink is the very first vertex to be considered. Now suppose that the sink is labeled with the triple $(u, +, a)$. Note that the second coordinate on the label must be $+$ since all edges incident with the sink are oriented towards the sink.

We claim that we can find an augmenting path P which results in an increased flow with $\delta = a$, the potential on the sink. To see this, we merely back-track. The source T got its label from $u = u_1$, u_1 got its label from u_2, and so forth. Eventually, we discover a vertex u_m which got its label from the source. The augmenting path is then $P = (S = u_m, u_{m-1}, \ldots, u_1, T)$. Clearly, the value of δ for this path is the potential a on the sink.

And if the Sink is Not Labeled? On the other hand, suppose we have scanned from every labeled vertex and there are still unlabeled vertices remaining, one of which is the

sink. Now we claim victory. To see that we have won, we simply observe that if L is the set of labeled vertices, and U is the set of unlabeled vertices, the every edge $e = (x, y)$ with $x \in L$ and $y \in U$ is full, i.e., $\phi(e) = c(e)$. If this were not the case, then y would qualify for a label with x as the first coordinate. Also, note that $\phi(y, x) = 0$ for every edge e with $x \in L$ and $y \in U$. Regardless, we see that the capacity of the cut $V = L \cup U$ is exactly equal to the value of the current cut, so we have both a maximum flow and minimum cut providing a certificate of optimality. Sweet!

11.5 A Concrete Example

Let's apply the Labeling Algorithm to the network flow shown in Figure FIX ME. Then we start with the source:

$$S: \quad (*, +, \infty)$$

and since the source S is the first vertex labeled, it is also the first one scanned. So we look at the neighbors of S using the pseudo-alphabetic order on the vertices. So the first one to be considered is vertex B and since the edge (S, B) is not full, we label B:

$$B: \quad (S, +, 8)$$

We then consider vertex E and label it as

$$E: \quad (S, +, 28)$$

Next is vertex F which is labeled as

$$F: \quad (S, +, 15)$$

At this point, the scan from S is complete. The first vertex after S to be labeled was B so we now scan from B. The (unlabeled) neighbors of B to be considered, in order, are A, C, and D. This results in the following labels:

$$\begin{aligned} A: &\quad (B, +, 8) \\ C: &\quad (B, +, 8) \\ D: &\quad (B, -, 6) \end{aligned}$$

The next vertex to be scanned is E, but E has no unlabeled neighbors. So we then move on the F, which again has no unlabeled neighbors. Finally, we scan from A and using the pseudo-alphabetic order, we first consider the sink T (which in this case is the only remaining unlabeled vertex. This results in the following label for T.

$$T: \quad (A, +, 8)$$

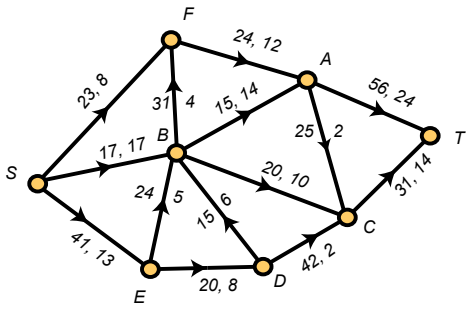

FIGURE 11.4: AN UPDATED NETWORK FLOW

Now that the sink is labeled, we know there is an augmenting path. We discover this path by backtracking. The sink T got its label from A; vertex A got its label from B, and B got its label from S. Therefore the augmenting path is $P = (S, B, A, T)$ with $\delta = 8$. All edges on this path are forward. The flow is then updated by increasing the flow on the edges of P by 8. This results in the flow shown in Figure 11.4. The value of this flow is 38.

Here is the sequence of labels that will be found when the labeling algorithm is applied to this updated flow (Note that in the scan from S, the vertex B will not be labeled, since now the edge (S, B) is full.

$$
\begin{aligned}
E: &\quad (S, +, 28)\\
F: &\quad (S, +, 15)\\
B: &\quad (E, +, 19)\\
D: &\quad (E, +, 12)\\
A: &\quad (F, +, 12)\\
C: &\quad (B, +, 10)\\
T: &\quad (A, +, 12)
\end{aligned}
$$

This labeling results in the augmenting path $P = (S, F, A, T)$ with $\delta = 12$.

After this update, the value of the flow has been increased and is now $50 = 38 + 12$. We start the labeling process over again and repeat until we reach a stage where some vertices (including the source) are labeled and some vertices (including the sink) are unlabeled.

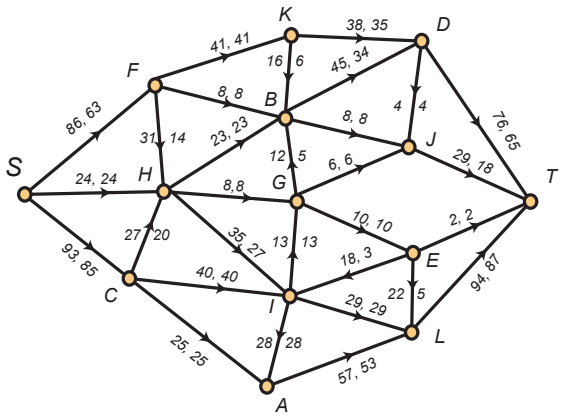

FIGURE 11.5: ANOTHER NETWORK FLOW

11.5.1 How the Labeling Algorithm Halts

Consider the following network flow:

The value of the flow is 172. Applying the labeling algorithm using the pseudo-alphabetic order results in the following labels.

$$
\begin{array}{ll}
S & (*, +, \infty) \\
C & (S, +, 8) \\
F & (S, +, 23) \\
H & (C, +, 7) \\
I & (H, +, 7) \\
E & (I, -, 3) \\
G & (E, -, 3) \\
L & (E, +, 3) \\
B & (G, +, 3) \\
T & (L, +, 3)
\end{array}
$$

These labels result in the augmenting path $P = (S, C, H, I, E, L, T)$ with $\delta = 3$.

However, after updating the flow and increasing its value to 175, the labeling algorithm halts with

$$\begin{array}{ll} S & (*, +, \infty) \\ C & (S, +, 5) \\ F & (S, +, 23) \\ H & (C, +, 4) \\ I & (H, +, 4) \end{array}$$

Now we observe that the labeled and unlabeled vertices are $L = \{S, C, F, H, I\}$ and $U = \{T, A, B, D, E, G, J, K\}$. Furthermore, the capacity of the cut $V = L \cup U$ is

$$41 + 8 + 23 + 8 + 13 + 29 + 28 + 25 = 175$$

This shows that we have found a cut whose capacity is exactly equal to the value of the current flow. In turn, this shows that the flow is optimal.

11.6 Integer Solutions of Linear Programming Problems

A linear programming problem is an optimization problem that can be stated in the following form: Find the maximum value of a linear function

$$c_1x_1 + c_2x_2 + c_3x_3 + \cdots + c_nx_n$$

subject to m constraints $C_1, C_2, \ldots, C_m$, where each constraint C_i is a linear equation of the form:

$$C_i: \quad a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \cdots + a_{in}x_n = b_i$$

where all coefficients and constants are real numbers.

While the general subject of linear programming is far too broad for this course, we would be remiss if we didn't point out that:

1. Linear programming problems are a *very* important class of optimization problems and they are have many applications in engineering, science, and industrial settings.

2. There are relatively efficient algorithms for finding solutions to linear programming problems.

3. A linear programming problem posed with rational coefficients and constants has an optimal solution with rational values—if it has an optimal solution at all.

4. A linear programming problem posed with integer coefficients and constants need not have an optimal solution with integer values—even when it has an optimal solution with rational values.

5. A very important theme in operations research is to determine when a linear programming problem posed in integers has has an optimal solution with integer values. This is a subtle and often very difficult problem.

6. The problem of finding a maximum flow in a network is a special case of a linear programming problem.

7. A network flow problem in which all capacities are integers has a maximum flow in which the flow on every edge is an integer. The Ford-Fulkerson labeling algorithm guarantees this!

8. In general, linear programming algorithms are not used on networks. Instead, special purpose algorithms, such as Ford-Fulkerson, have proven to be more efficient in practice.