Lecture 13

Network Flow

Supplemental reading in CLRS: Sections 26.1 and 26.2

When we concerned ourselves with shortest paths and minimum spanning trees, we interpreted the edge weights of an undirected graph as distances. In this lecture, we will ask a question of a different sort. We start with a *directed* weighted graph G with two distinguished vertices s (the source) and t (the sink). We interpret the edges as unidirectional water pipes, with an edge's capacity indicated by its weight. The *maximum flow problem* then asks, how can one route as much water as possible from s to t?

To formulate the problem precisely, let's make some definitions.

Definition. A **flow network** is a directed graph $G = (V, E)$ with distinguished vertices s (the source) and t (the sink), in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v)$. We require that E never contain both (u, v) and (v, u) for any pair of vertices u, v (so in particular, there are no loops). Also, if $u, v \in V$ with $(u, v) \notin E$, then we define $c(u, v)$ to be zero. (See Figure 13.1).

In these notes, we will always assume that our flow networks are finite. Otherwise, it would be quite difficult to run computer algorithms on them.

Definition. Given a flow network $G = (V, E)$, a **flow** in G is a function $f : V \times V \to \mathbb{R}$ satisfying

1. Capacity constraint: $0 \le f(u, v) \le c(u, v)$ for each $u, v \in V$

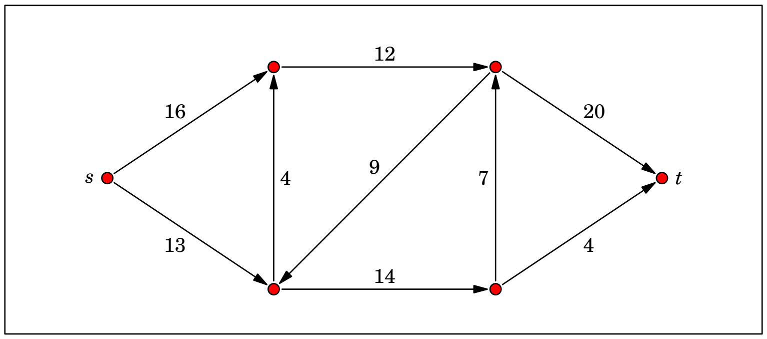

Figure 13.1. A flow network.

2. Flow conservation: for each $u \in V \setminus \{s,t\}$, we have[1]

$$\underbrace{\sum_{v \in V} f(v,u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u,v)}_{\text{flow out of } u}.$$

In the case that flow conservation is satisfied, one can prove (and it's easy to believe) that the net flow out of s equals the net flow into t. This quantity is called the **flow value**, or simply the magnitude, of f. We write

$$\underbrace{|f|}_{\text{flow value}} = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) = \sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v).$$

Note that the definition of a flow makes sense even when G is allowed to contain both an edge and its reversal (and therefore is not truly a flow network). This will be important in §13.1.1 when we discuss augmenting paths.

13.1 The Ford–Fulkerson Algorithm

The **Ford–Fulkerson algorithm** is an elegant solution to the maximum flow problem. Fundamentally, it works like this:

```
1 while there is a path from $s$ to $t$ that can hold more water do
2     Push more water through that path
```

Two notes about this algorithm:

- The notion of "a path from s to t that can hold more water" is made precise by the notion of an *augmenting path*, which we define in §13.1.1.
- The Ford–Fulkerson algorithm is essentially a greedy algorithm. If there are multiple possible augmenting paths, the decision of which path to use in line 2 is completely arbitrary.[2] Thus, like any terminating greedy algorithm, the Ford–Fulkerson algorithm will find a locally optimal solution; it remains to show that the local optimum is also a global optimum. This is done in §13.2.

13.1.1 Residual Networks and Augmenting Paths

The Ford–Fulkerson algorithm begins with a flow f (initially the zero flow) and successively improves f by pushing more water along some path p from s to t. Thus, given the current flow f, we need

[1] In order for a flow of water to be sustainable for long periods of time, there cannot exist an accumulation of excess water anywhere in the pipe network. Likewise, the amount of water flowing into each node must at least be sufficient to supply all the outgoing connections promised by that node. Thus, the amount of water entering each node must equal the amount of water flowing out. In other words, the net flow into each vertex (other than the source and the sink) must be zero.

[2] There are countless different versions of the Ford–Fulkerson algorithm, which differ from each other in the heuristic for choosing which augmenting path to use. Different situations (in which we have some prior information about the nature of G) may call for different heuristics.

a way to tell how much more water a given path p can carry. To start, note that a chain is only as strong as its weakest link: if $p = \langle v_0, \dots, v_n \rangle$, then

$$\begin{pmatrix}\text{amount of additional water}\\ \text{that can flow through } p\end{pmatrix} = \min_{1 \le i \le n} \begin{pmatrix}\text{amount of additional water that}\\ \text{can flow directly from } v_{i-1} \text{ to } v_i\end{pmatrix}.$$

All we have to know now is how much additional water can flow directly between a given pair of vertices u,v. If $(u,v) \in E$, then clearly the flow from u to v can be increased by up to $c(u,v) - f(u,v)$. Next, if $(v,u) \in E$ (and therefore $(u,v) \notin E$, since G is a flow network), then we can simulate an increased flow from u to v by decreasing the throughput of the edge (v,u) by as much as $f(v,u)$. Finally, if neither (u,v) nor (v,u) is in E, then no water can flow directly from u to v. Thus, we define the **residual capacity** between u and v (with respect to f) to be

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{13.1}$$

When drawing flows in flow networks, it is customary to label an edge (u,v) with both the capacity $c(u,v)$ and the throughput $f(u,v)$, as in Figure 13.2.

Next, we construct a directed graph G_f, called the **residual network** of f, which has the same vertices as G, and has an edge from u to v if and only if $c_f(u,v)$ is positive. (See Figure 13.2.) The weight of such an edge (u,v) is $c_f(u,v)$. Keep in mind that $c_f(u,v)$ and $c_f(v,u)$ may both be positive for some pairs of vertices u,v. Thus, the residual network of f is in general not a flow network.

Equipped with the notion of a residual network, we define an **augmenting path** to be a path from s to t in G_f. If p is such a path, then by virtue of our above discussion, we can perturb the flow f at the edges of p so as to increase the flow value by $c_f(p)$, where

$$c_f(p) = \min_{(u,v) \in p} c_f(u,v). \tag{13.2}$$

The way to do this is as follows. Given a path p, we might as well assume that p is a simple path.[3] In particular, p will never contain a given edge more than once, and will never contain both an edge and its reversal. We can then define a new flow f' in the residual network (even though the residual network is not a flow network) by setting

$$f'(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \in p \\ 0 & \text{otherwise.} \end{cases}$$

Exercise 13.1. *Show that f' is a flow in G_f, and show that its magnitude is $c_f(p)$.*

Finally, we can "**augment**" f by f', obtaining a new flow $f \uparrow f'$ whose magnitude is $|f| + |f'| =$

[3] Recall that a *simple path* is a path which does not contain any cycles. If p is not simple, we can always pare p down to a simple path by deleting some of its edges (see Exercise B.4-2 of CLRS, although the claim I just made is a bit stronger). Doing so will never decrease the residual capacity of p (just look at (13.2)).

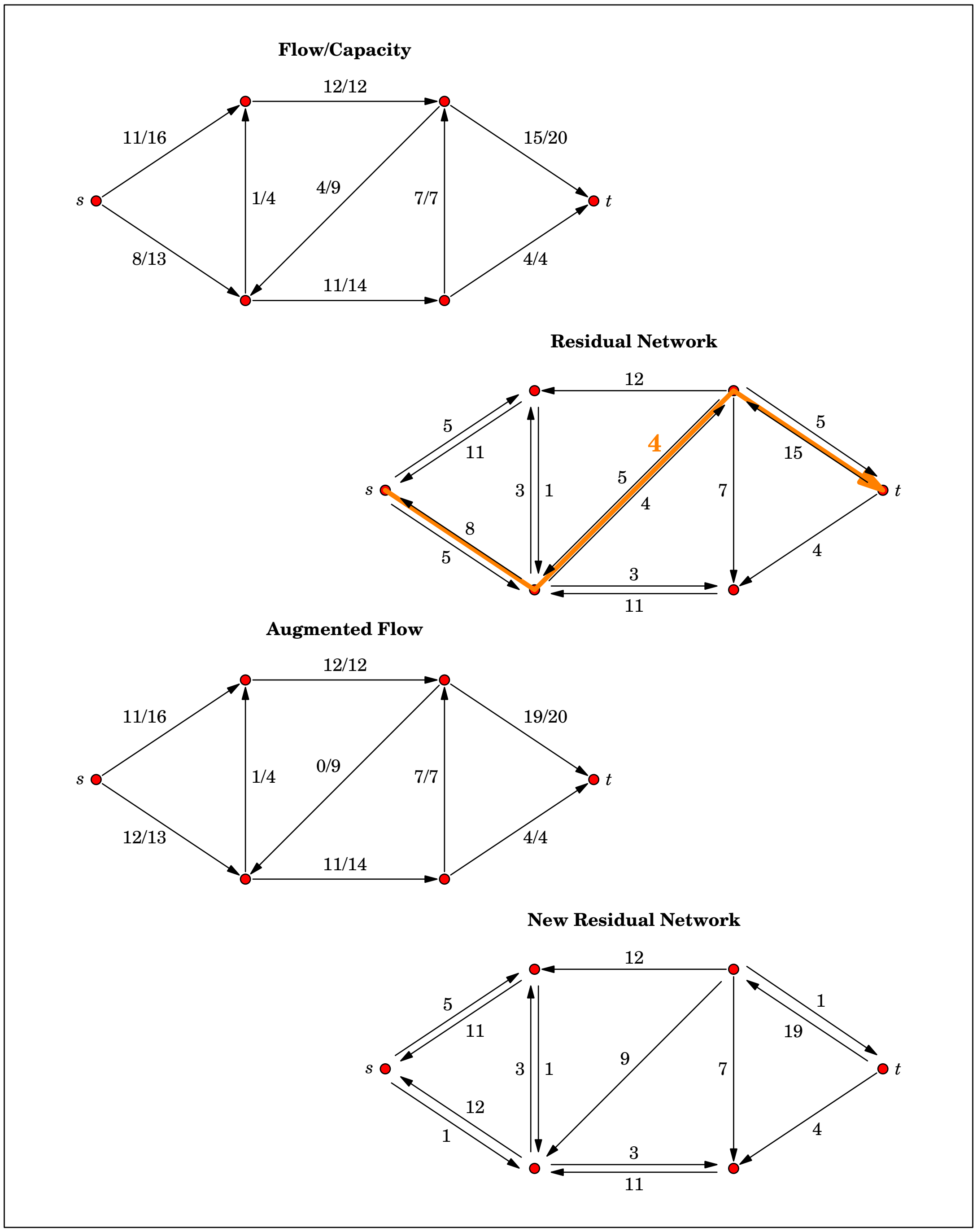

Figure 13.2. We begin with a flow network G and a flow f: the label of an edge (u,v) is "a/b," where $a = f(u,v)$ is the flow through the edge and $b = c(u,v)$ is the capacity of the edge. Next, we highlight an augmenting path p of capacity 4 in the residual network G_f. Next, we augment f by the augmenting path p. Finally, we obtain a new residual network in which there happen to be no more augmenting paths. Thus, our new flow is a maximum flow.

$|f| + c_f(p)$. It is defined by[4]

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + c_f(p) & \text{if } (u,v) \in p \text{ and } (u,v) \in E \\ f(u,v) - c_f(p) & \text{if } (v,u) \in p \text{ and } (u,v) \in E \\ f(u,v) & \text{otherwise.} \end{cases}$$

Lemma 13.1 (CLRS Lemma 26.1)**.** *Let f be a flow in the flow network $G = (V,E)$ and let f' be a flow in the residual network G_f. Let $f \uparrow f'$ be the augmentation of f by f', as described in* (13.3)*. Then*

$$|f \uparrow f'| = |f| + |f'|.$$

Proof sketch. First, we show that $f \uparrow f'$ obeys the capacity constraint for each edge in E and obeys flow conservation for each vertex in $V \setminus \{s,t\}$. Thus, $f \uparrow f'$ is truly a flow in G. Next, we obtain the identity $|f \uparrow f'| = |f| + |f'|$ by simply expanding the left-hand side and rearranging terms in the summation. □

13.1.2 Pseudocode Implementation of the Ford–Fulkerson Algorithm

Now that we have laid out the necessary conceptual machinery, let's give more detailed pseudocode for the Ford–Fulkerson algorithm.

```
Algorithm: FORD–FULKERSON($G$)
▷ Initialize flow $f$ to zero
for each edge $(u,v) \in E$ do
    $(u,v).f \leftarrow 0$
▷ The following line runs a graph search algorithm (such as BFS or DFS)* to find a
  path from $s$ to $t$ in $G_f$
while there exists a path $p : s \rightsquigarrow t$ in $G_f$ do
    $c_f(p) \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$
    for each edge $(u,v) \in p$ do
        ▷ Because $(u,v) \in G_f$, it must be the case that either $(u,v) \in E$ or $(v,u) \in E$.
        ▷ And since $G$ is a flow network, the "or" is exclusive: $(u,v) \in E$ xor $(v,u) \in E$.
        if $(u,v) \in E$ then
            $(u,v).f \leftarrow (u,v).f + c_f(p)$
        else
            $(v,u).f \leftarrow (v,u).f - c_f(p)$
```

* For more information about breath-first and depth-first searches, see Sections 22.2 and 22.3 of CLRS.

Here, we use the notation $(u,v).f$ synonymously with $f(u,v)$; though, the notation $(u,v).f$ suggests a convenient implementation decision in which we attach the value of $f(u,v)$ as satellite data to the

[4] In a more general version of augmentation, we don't require p to be a simple path; we just require that f' be some flow in the residual network G_f. Then we define

$$f'(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{13.3}$$

edge (u,v) itself rather than storing all of f in one place. Also note that, because we often need to consider both $f(u,v)$ and $f(v,u)$ at the same time, it is important that we equip each edge $(u,v) \in E$ with a pointer to its reversal (v,u). This way, we may pass from an edge (u,v) to its reversal (v,u) without performing a costly search to find (v,u) in memory.

We defer the proof of correctness to §13.2. We do show, though, that the Ford–Fulkerson algorithm halts if the edge capacities are integers.

Proposition 13.2. *If the edge capacities of G are integers, then the Ford–Fulkerson algorithm terminates in time $O\left(E \cdot |f^*|\right)$, where $|f^*|$ is the magnitude of any maximum flow for G.*

Proof. Each time we choose an augmenting path p, the right-hand side of (13.2) is a positive integer. Therefore, each time we augment f, the value of $|f|$ increases by at least 1. Since $|f|$ cannot ever exceed $|f^*|$, it follows that lines 5–13 are repeated at most $|f^*|$ times. Each iteration of lines 5–13 takes $O(E)$ time if we use a breadth-first or depth-first search in line 5, so the total running time of FORD–FULKERSON is $O\left(E \cdot |f^*|\right)$. □

Exercise 13.2. *Show that, if the edge capacities of G are rational numbers, then the Ford–Fulkerson algorithm eventually terminates. What sort of bound can you give on its running time?*

Proposition 13.3. *Let G be a flow network. If all edges in G have integer capacities, then there exists a maximum flow in G in which the throughput of each edge is an integer. One such flow is given by running the Ford–Fulkerson algorithm on G.*

Proof. Run the Ford–Fulkerson algorithm on G. The residual capacity of each augmenting path p in line 5 is an integer (technically, induction is required to prove this), so the throughput of each edge is only ever incremented by an integer. The conclusion follows if we assume that the Ford–Fulkerson algorithm is correct. The algorithm is in fact correct, by Corollary 13.8 below. □

Flows in which the throughput of each edge is an integer occur frequently enough to deserve a name. We'll call them **integer flows**.

Perhaps surprisingly, Exercise 13.2 is not true when the edge capacities of G are allowed to be arbitrary real numbers. This is not such bad news, however: it simply says that there *exists* a sufficiently foolish way of choosing augmenting paths so that FORD–FULKERSON never terminates. If we use a reasonably good heuristic (such as the shortest-path heuristic used in the Edmonds–Karp algorithm of §13.1.3), termination is guaranteed, and the running time needn't depend on $|f^*|$.

13.1.3 The Edmonds–Karp Algorithm

The **Edmonds–Karp algorithm** is an implementation of the Ford–Fulkerson algorithm in which the the augmenting path p is chosen to have *minimal length* among all possible augmenting paths (where each edge is assigned length 1, regardless of its capacity). Thus the Edmonds–Karp algorithm can be implemented by using a breadth-first search in line 5 of the pseudocode for FORD–FULKERSON.

Proposition 13.4 (CLRS Theorem 26.8)**.** *In the Edmonds–Karp algorithm, the total number of augmentations is $O(VE)$. Thus total running time is $O\left(VE^2\right)$.*

Proof sketch.

- First one can show that the lengths of the paths p found by breadth-first search in line 5 of FORD–FULKERSON are monotonically nondecreasing (this is Lemma 26.7 of CLRS).

- Next, one can show that each edge $e \in E$ can only be the bottleneck for p at most $O(V)$ times. (By "bottleneck," we mean that e is the (or, an) edge of smallest capacity in p, so that $c_f(p) = c_f(e)$.)
- Finally, because only $O(E)$ pairs of vertices can ever be edges in G_f and because each edge can only be the bottleneck $O(V)$ times, it follows that the number of augmenting paths p used in the Edmonds–Karp algorithm is at most $O(VE)$.
- Again, since each iteration of lines 5–13 of FORD–FULKERSON (including the breadth-first search) takes time $O(E)$, the total running time for the Edmonds–Karp algorithm is $O\left(VE^2\right)$.

□

The shortest-path heuristic of the Edmonds–Karp algorithm is just one possibility. Another interesting heuristic is *relabel-to-front*, which gives a running time of $O\left(V^3\right)$. We won't expect you to know the details of relabel-to-front for 6.046, but you might find it interesting to research other heuristics on your own.

13.2 The Max Flow–Min Cut Equivalence

Definition. A **cut** $(S,\ T = V \setminus S)$ of a flow network G is just like a cut (S,T) of the graph G in the sense of §3.3, except that we require $s \in S$ and $t \in T$. Thus, any path from s to t must cross the cut (S,T). Given a flow f in G, the **net flow** $f(S,T)$ across the cut (S,T) is defined as

$$f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u). \tag{13.4}$$

One way to picture this is to think of the cut (S,T) as an oriented dam in which we count water flowing from S to T as positive and water flowing from T to S as negative. The **capacity** of the cut (S,T) is defined as

$$c(S,T) = \sum_{u\in S}\sum_{v\in T} c(u,v). \tag{13.5}$$

The motivation for this definition is that $c(S,T)$ should represent the maximum amount of water that could ever possibly flow across the cut (S,T). This is explained further in Proposition 13.6.

Lemma 13.5 (CLRS Lemma 26.4)**.** *Given a flow f and a cut (S,T), we have*

$$f(S,T) = |f|.$$

We omit the proof, which can be found in CLRS. Intuitively, this lemma is an easy consequence of flow conservation. The water leaving s cannot build up at any of the vertices in S, so it must cross over the cut (S,T) and eventually pour out into t.

Proposition 13.6. *Given a flow f and a cut (S,T), we have*

$$f(S,T) \le c(S,T).$$

Thus, applying Lemma 13.5, we find that for any flow f and any cut (S,T), we have

$$|f| \le c(S,T).$$

Proof. In (13.4), $f(v,u)$ is always nonnegative. Moreover, we always have $f(u,v) \le c(u,v)$ by the capacity constraint. The conclusion follows. $\square$

Proposition 13.6 tells us that the magnitude of a maximum flow is at most equal to the capacity of a minimum cut (i.e., a cut with minimum capacity). In fact, this bound is tight:

Theorem 13.7 (Max Flow–Min Cut Equivalence)**.** *Given a flow network G and a flow f, the following are equivalent:*

(i) *f is a maximum flow in G.*

(ii) *The residual network G_f contains no augmenting paths.*

(iii) *$|f| = c(S,T)$ for some cut (S,T) of G.*

If one (and therefore all) of the above conditions hold, then (S,T) is a minimum cut.

Proof. Obviously (i) implies (ii), since an augmenting path in G_f would give us a way to increase the magnitude of f. Also, (iii) implies (i) because no flow can have magnitude greater than $c(S,T)$ by Proposition 13.6.

Finally, suppose (ii). Let S be the set of vertices u such that there exists a path $s \rightsquigarrow u$ in G_f. Since there are no augmenting paths, S does not contain t. Thus (S,T) is a cut of G, where $T = V \setminus S$. Moreover, for any $u \in S$ and any $v \in T$, the residual capacity $c_f(u,v)$ must be zero (otherwise the path $s \rightsquigarrow u$ in G_f could be extended to a path $s \rightsquigarrow u \to v$ in G_f). Thus, glancing back at (13.1), we find that whenever $u \in S$ and $v \in T$, we have

$$f(u,v) = \begin{cases} c(u,v) & \text{if } (u,v) \in E \\ 0 & \text{if } (v,u) \in E \\ 0 \text{ (but who cares)} & \text{otherwise.} \end{cases}$$

Thus we have

$$f(S,T) = \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{u\in S}\sum_{v\in T} 0 = c(S,T) - 0 = c(S,T);$$

so (iii) holds. Because the magnitude of any flow is at most the capacity of any cut, f must be a maximum flow and (S,T) must be a minimum cut. $\square$

Corollary 13.8. *The Ford–Fulkerson algorithm is correct.*

Proof. When FORD–FULKERSON terminates, there are no augmenting paths in the residual network G_f. $\square$

13.3 Generalizations

The definition of a flow network that we laid out may seem insufficient for handling the types of flow problems that come up in practice. For example, we may want to find the maximum flow in a directed graph which sometimes contains both an edge (u,v) and its reversal (v,u). Or, we might want to find the maximum flow in a directed graph with multiple sources and sinks. It turns out that both of these generalizations can easily be reduced to the original problem by performing clever graph transformations.

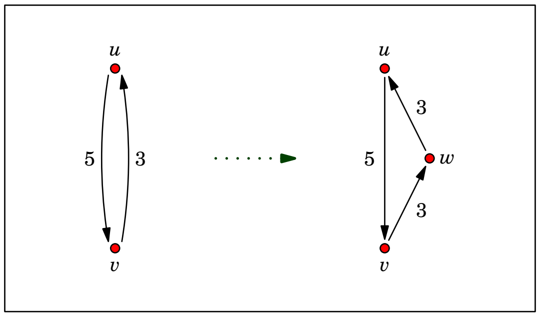

Figure 13.3. We can resolve the issue of E containing both an edge and its reversal by creating a new vertex and rerouting one of the old edges through that vertex.

13.3.1 Allowing both an edge and its reversal

Suppose we have a directed weighted graph $G = (V, E)$ with distinguished vertices s and t. We would like to use the Ford–Fulkerson algorithm to solve the flow problem on G, but G might not be a flow network, as E might contain both (u, v) and (v, u) for some pair of vertices u, v. The trick is to construct a new graph $G' = (V', E')$ from G in the following way: Start with $(V', E') = (V, E)$. For every unordered pair of vertices $\{u, v\} \subseteq V$ such that both (u, v) and (v, u) are in E, add a dummy vertex w to V'. In E', replace the edge (u, v) with edges (u, w) and (w, v), each with capacity $c(u, v)$ (see Figure 13.3). It is easy to see that solving the flow problem on G' is equivalent to solving the flow problem on G. But G' is a flow network, so we can use FORD–FULKERSON to solve the flow problem on G'.

13.3.2 Allowing multiple sources and sinks

Next suppose we have a directed graph $G = (V, E)$ in which there are multiple sources $s_1, \ldots, s_k$ and multiple sinks $t_1, \ldots, t_\ell$. Again, it makes sense to talk about the flow problem in G, but the Ford–Fulkerson algorithm does not immediately give us a way to solve the flow problem in G. The trick this time is to add new vertices s and t to V. Then, join s to each of $s_1, \ldots, s_k$ with a directed edge of capacity ∞,[5] and join each of $t_1, \ldots, t_\ell$ to t with a directed edge of capacity ∞ (see Figure 13.4). Again, it is easy to see that solving the flow problem in this new graph is equivalent to solving the flow problem in G.

13.3.3 Multi-commodity flow

Even more generally, we might want to transport multiple types of commodities through our network simultaneously. For example, perhaps G is a road map of New Orleans and the commodities are emergency relief supplies (food, clothing, flashlights, gasoline. . .) in the wake of Hurricane Katrina. In the **multi-commodity flow** problem, there are commodities $1, \ldots, k$, sources $s_1, \ldots, s_k$, sinks $t_1, \ldots, t_k$, and quotas (i.e., positive numbers) $d_1, \ldots, d_k$. Each source s_i needs to send d_i units of commodity i to sink t_i. (See Figure 13.5.) The problem is to determine whether there is a way to do so while still obeying flow conservation and the capacity constraint, and if so, what that way is.

[5] The symbol ∞ plays the role of a *sentinel value* representing infinity (such that $\infty > x$ for every real number x). Depending on your programming language (and on whether you cast the edge capacities as integers or floating-point numbers), the symbol ∞ may or may not be supported natively. If it is not supported natively, you will have to either implement it or add extra code to the implementation of FORD–FULKERSON so that operations on edge capacities support

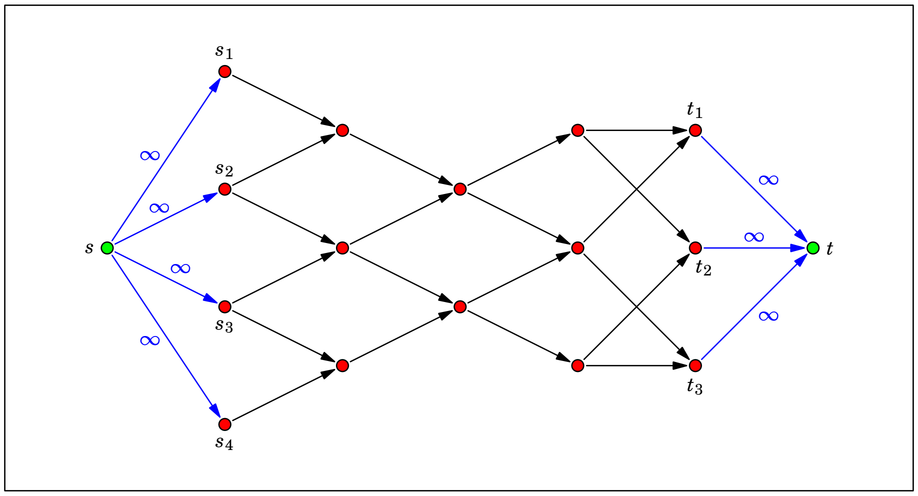

Figure 13.4. We can allow multiple sources and sinks by adding a "master" source that connects to each other source via an edge of capacity ∞, and similarly for the sinks.

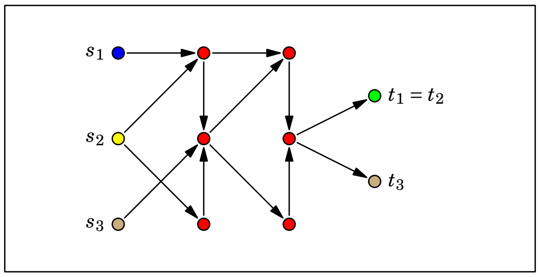

Figure 13.5. An instance of the multi-commodity flow problem with three commodities.

It turns out that the multi-commodity flow problem is NP-complete, even when $k = 2$ and all the edge capacities are 1. Thus, most computer scientists currently believe that there is no way to solve the multi-commodity flow problem in polynomial time, though it has not been definitively proved yet.

both numbers and the symbol ∞.