CS 577: Introduction to Algorithms	Fall 2015

Network Flow

Instructor: Dieter van Melkebeek — Scribe: Andrew Morgan

DRAFT

When we covered greedy algorithms, we thought of them as making a sequence of irrevocable decisions. By making the right decisions, we get a simple, correct algorithm. However, we don't always want to forced to make our decisions irrevocable; we want to be able to change our mind after considering new information about the input instance. In general, this can lead to a tremendous blow-up in the running time, since most algorithms along these lines end up considering an exponential number of possible solutions. When we covered dynamic programming, we saw how to mitigate this: the idea is to give an algorithm that organizes the solutions concisely, so that the algorithm ultimately only considers a small number of possible solutions.

In these notes, we present a different approach to extending greedy algorithms, that of *network flow*. The basic framework here will be a bit different than in previous sections, because network flow is itself a single computational problem. The way we will apply "network flow" to other computational problems will be by *reducing* them to network flow.

For instance, in Section 2, we'll define the maximum bipartite matching problem. The way we'll solve this problem is by turning instances of it into instances of the network flow problem, solving the network flow problem, and then turning the result of the network flow problem into a solution to the original matching problem. The main idea here is that it's much easier to reduce matching to network flow than it is to produce from scratch an algorithm for matching. We present a few more examples demonstrating the power and flexibility of this idea of reducing to network flow, as well as to give an intuitive idea of when a problem can nicely reduce to network flow.

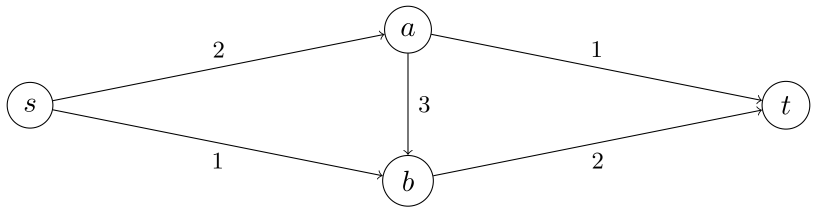
Another motivation for studying network flow is its duality theory. As we'll see, network flow is a maximization problem, and it has as a dual problem, "minimum cut", which is a minimization problem. The duality theory for network flow has as its main theorem that these two problems are very closely related. Other than being a neat result in its own right, this duality itself has a number of mathematical consequences—the well-known Hall's Marriage Theorem and Menger's Theorem are two we will specifically cover. It will also expand the range of problems which reduce simply to network flow, such as the project selection problem in Section 5, where the most natural reduction is to the minimum cut problem.

1 Network Flow

A *network* is a directed graph $G = (V, E)$ with two distinct, distinguished vertices, $s, t \in V$, called the *source* and *sink* respectively. These vertices have the property that the in-degree of s is zero, and that the out-degree of t is zero. Each network additionally has a *capacity function* on the edges, $c : E \to [0, +\infty)$. Figure 1 sketches an example of a network.

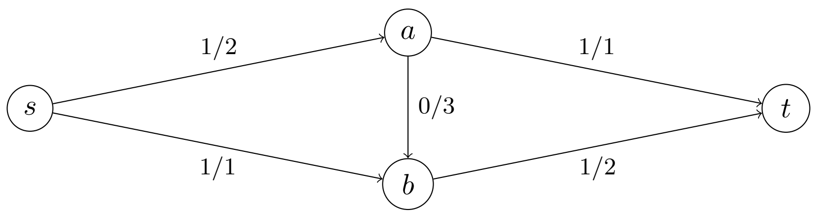
A *flow* through a network G is a function on the edges, $f : E \to [0, +\infty)$ that satisfies two properties:

Figure 1: A network



A depiction of a network on four vertices. The capacities of each edge are written along the corresponding edge.

Figure 2: A flow through a network



A depiction of a flow through the network from Figure 1. The flows and capacities are written as fractions—the expression a/b next to an edge e means $f(e) = a$ and $c(e) = b$. Note that this satisfies the conservation and capacity constraints.

- **(Conservation)** For every vertex besides the source and sink, the total flow *entering* the vertex is equal to the total flow *leaving* the vertex. Symbolically,

$$f_{\text{in}}(v) = f_{\text{out}}(v)$$

- **(Capacity)** For every edge $e \in E$, the flow *across* the edge e is at most the capacity of e. Symbolically,

$$f(e) \leq c(e)$$

The notation $f_{\text{in}}(v)$ denotes the sum over all edges *entering into* v of the flow across that edge. The notation $f_{\text{out}}(v)$ denotes the sum over all edges *exiting out of* v of the flow across that edge. Symbolically:

$$f_{\text{in}}(v) \doteq \sum_{e \in E \,:\, e=(u,v)} f(e)$$

and

$$f_{\text{out}}(v) \doteq \sum_{e \in E \,:\, e=(v,w)} f(e)$$

Figure 2 sketches a representation of a flow through a network.

At an intuitive level, a network can be thought of as a series of tubes through which something, say oil, might flow. Each tube has a capacity, indicating the maximum amount of oil that can flow through that tube. The source represents an oil supplier, who is trying to transport oil to the sink. The source can only do so by pushing the oil through the tubes. The ways of pushing oil through the tubes correspond to flows. The conservation constraint simply says that the intermediate nodes neither add nor remove oil, and the capacity constraints ensure that no more oil is being sent through a single edge than is possible.

A natural question to ask in this setting is, how much oil can the source push to the sink? Or more abstractly, how much flow can be sent from the source node to the sink node? To measure this, we need a way to quantify the total amount of flow being sent from the source to the sink. We call this the *value* of the flow, and denote the value of a flow f by $\mu(f)$.

One way to define the value of a flow is to just add up the total amount of flow leaving the source vertex, *i.e.*, $\mu(f) \doteq f_{\text{out}}(s)$. But what's so special about s? Why not define the value of a flow to be the total flow coming into the sink vertex? Intuitively, these quantities should actually be the same. More generally, we should expect that, whenever we cut the network into two pieces, S and T, so that the source is in S and the sink is in T, then the value of the flow should just be the total amount of flow crossing from S to T, and that this really should not depend on the choice of cut.

Indeed, this does turn out to be the case. Formally, we define an s-t cut to be a cut (S, T) of G (the graph underlying the network) which has the property that s is in S and t is in T. The value of the flow f *with respect to the s-t cut* (S, T) is the *net* flow crossing the cut from S to T. *i.e.*, , the value of the flow is the total flow crossing the cut *from S to T* minus the total flow crossing the cut *from T to S*. We denote this quantity by $\mu(f, (S, T))$. Symbolically,

$$\mu(f, (S,T)) \doteq \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e) \tag{1}$$

(The notation $E(S, T)$ means the edges $(u, v) \in E$ for which u is in S and v is in T.) Our first step toward understanding network flow is in proving that $\mu(f, (S, T))$ does not depend on the s-t cut chosen. Formally, we have Proposition 1.

Proposition 1. *For every s-t cut (S, T), we have $\mu(f) = \mu(f, (S, T))$.*

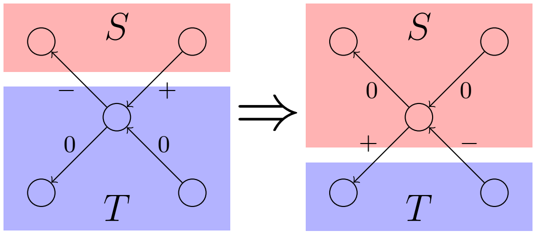
Proof. The proof essentially boils down to the conservation property of flows and the fact that any s-t cut has s on one side and t on the other. We give two arguments. Figure 3 gives a visual aide to understanding.

The first argument is inductive; we sketch the main ideas. The claim is trivial when $S = \{s\}$: $\mu(f)$ is exactly the first sum in Equation (1), and since the in-degree of s is zero, the second sum is zero. For cuts (S, T) with larger S, we can think of starting with the cut $(\{s\}, V \setminus \{s\})$ and then moving vertices v from the 't-side' of the cut to the 's-side' of the cut until we have (S, T). When we move a single vertex across, the sums in (1) change. In particular, the whole expression decreases by $f_{\text{in}}(v)$, as terms either disappear from the left sum in (1) or appear in the right sum in (1). Similarly, it increases by $f_{\text{out}}(v)$, as terms either disappear from the left sum or appear in the right sum. However, when v isn't the source or sink, we know that $f_{\text{in}}(v) = f_{\text{out}}(v)$, which means the overall change to the sum is zero.

The second argument is a single-step, purely algebraic proof, we can add the following set of equations:

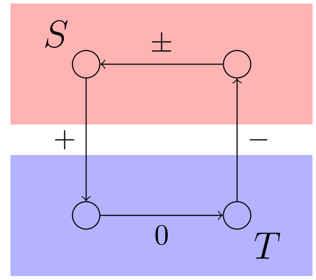
$$\begin{array}{ccccccc} & \mu(v) & \doteq & f_{\text{out}}(s) & - & f_{\text{in}}(s) & \\ + & 0 & = & f_{\text{out}}(v) & - & f_{\text{in}}(v) & \forall v \in S \setminus \{s\} \\ \hline & \mu(v) & = & \sum_{v \in S} f_{\text{out}}(v) & - & \sum_{v \in S} f_{\text{in}}(v) & \end{array}$$

Figure 3: Proof of Proposition 1



(a) The inductive step in the first proof

The + and − signs indicate the sign by which the edge is counted; the symbol 0 means the edge is not counted. Note that as the center node crosses from the blue region to the red region, the signs of edges leaving the center node increase by one, and the signs of edges entering the center node decrease by one.



(b) The cancellation in the second proof

The + and − signs indicate the sign by which the edge is counted. The symbol 0 indicates the edge is not counted in the sum. The symbols ± indicates that the edge is counted with both signs, and hence cancels.

and observe that we can rewrite the sums on the right-hand-side as

$$\sum_{e \in E(S,V)} f(e) - \sum_{e \in E(V,S)} f(e)$$

By canceling the common terms in the sums (corresponding to edges in $E(S,S)$) to get

$$\mu(f) = \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e) \doteq \mu(f,(S,T))$$

$\square$

Proposition 1 tells us that there is a meaningful way to value a flow, and that this meaning matches our intuition. Our objective in the network flow problem will be to find a flow through a network with *maximum* value. Formally the network flow problem meets the following description:

Input: A network $G = (V, E)$ with capacity function $c : E \to [0, +\infty)$.

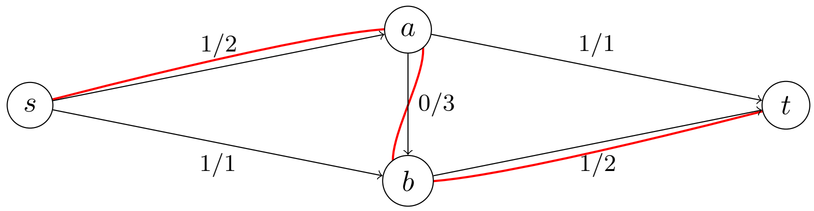
Output: A flow f through G whose value $\mu(f)$ is maximum.

1.1 Weak Duality

Consider the flow given in Figure 4. It has value two. There is a flow with larger value: simply 'push' one unit of flow from s to a to b to t (as indicated by the red path). This yields a new flow with value three. Can we do better than three? The answer in this case is no: every flow has to satisfy the capacity constraints, so every flow f in the example network must have $f_{\text{out}}(s) \leq 3$, and thus $\mu(f) \leq 3$.

In general, every flow must have value at most the total capacity of edges leaving s. Similarly, every flow must have value at most the total capacity of edges entering t. Even more generally, we can consider arbitrary s-t cuts and define a notion of capacity for the whole cut, with the property that the value of the flow is at most the capacity of the cut.

Figure 4: A non-maximal flow



The flow given by the fractions has value two, and is not maximal. Pushing one additional unit of flow along the red curve yields a flow with value three. The new flow is maximal, because the s-t cut with $S = \{s\}, T = \{a, b, t\}$ has capacity three.

Fix an s-t cut (S, T) arbitrarily. We want to define the *capacity of the cut* to be a quantity which easily upper bounds the value of any flow. An easy way to do this is to simply bound the value of the flow relative to the cut (S, T), and then appeal to Proposition 1. We can do this by using two facts:

- For flow crossing from S to T, the total flow has to be at most the sum of the capacities of the edges from S to T. This is because, by the capacity constraints, the flow across any given edge crossing from S to T has to be at most the capacity of that edge.
- For flow crossing from T to S, the total flow has to be nonnegative; this simply follows from the fact that the flow along any given edge must be nonnegative.

Thus the *net flow* from S to T has to be at most the total capacity of the edges from S to T (minus zero, for the edges from T to S). We denote it by $c(S, T)$, and define it formulaically as

$$c(S,T) = \sum_{e \in E(S,T)} c(e)$$

It's easy to see then that the value of every flow has to be at most the capacity of every cut: for any flow f and s-t cut (S, T), we have

$$\begin{aligned} \mu(f) &= \mu(f, (S,T)) \\ &= \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e) \\ &\leq \sum_{e \in E(S,T)} c(e) - \sum_{e \in E(T,S)} 0 \\ &\doteq c(S,T) \end{aligned}$$

This fact is referred to as *weak duality*. We can formulate it more traditionally as follows:

$$\max_{\text{flows } f} \mu(f) \leq \min_{s\text{-}t \text{ cuts } (S,T)} c(S,T)$$

or even more succinctly as

$$\textbf{Max-Flow} \ \leq \ \textbf{Min-Cut}$$

1.2 Strong Duality

A natural question is to ask for the conditions under which equality holds in the weak duality statement. The answer, perhaps surprisingly, is *always*.

We will prove this by showing partial correctness of an algorithm which computes the maximum flow in a network. But to introduce the algorithm, we need a couple more concepts.

The first is that of an *augmenting path*. This is simply a formalization of the red path in Figure 4. In the example, we could 'push' an extra unit of flow along the red path, and this increased the value of the flow. In general, let G be a network with capacity function c, and let f be a flow through G. Let P be a path from s to t through G with the property that, for every edge e in P, the flow across e is strictly less than the capacity of e, *i.e.*, $f(e) < c(e)$. Let $\alpha > 0$ be such that $f(e) + \alpha \leq c(e)$ for all edges e in the path P. Then we can *augment* the flow f along the path P by adding α units of flow to each edge in P. In other words, we get a new flow f' defined to be

$$f'(e) = \begin{cases} f(e) + \alpha & : \quad e \in P \\ f(e) & : \quad e \notin P \end{cases}$$

It's easy to see that f' satisfies the conservation and capacity constraints, so it is also a flow. Furthermore, the value of the flow has increased by α.

Augmenting paths will be the essential tool for computing a maximum flow. To demonstrate that, let's fix a flow f. Consider the set S of vertices to which s can send positive flow. Either t is in this set of vertices or it is not. We have the following alternative:

- If t is in S, then there is an augmenting path from s to t, and so we can improve our flow f.
- If t is not in S, then we get an s-t cut, namely $(S, V \setminus S)$.

It would be really nice if, in the second case, the s-t cut has capacity exactly equal to value of f. In particular, this would imply that f is a maximum flow by weak duality. The first case of the alternative even tells us how to compute it: simply repeatedly find augmenting paths until none exist.

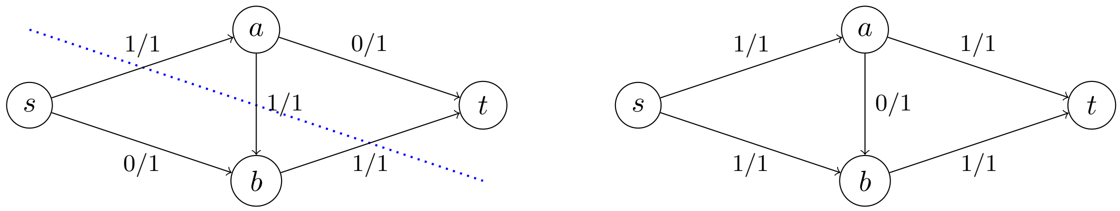
This plan almost works: suppose we have a flow f in the second case above, and consider any edge (u, v) in $E(S, V \setminus S)$. Since u is in S, there is a way to send positive flow from s to u, and because v is not in S, there is no way to send positive flow from s to v. This implies that $c((u, v)) = f((u, v))$, since otherwise we could send positive flow from s to v by first sending it from s to u and then along the edge (u, v). In other words, the cut we find will be *full in the forward direction*. Thus, in order for our plan to work, we just need for there to be no flow crossing the cut in the backward direction.

Strictly speaking, however, this does not always happen. Figure 5 gives an example.

What we need to make this work is the following observation: if f is a flow through a network, and e is an edge in this network for which $f(e) > 0$, then we can view this flow as the *capacity to send flow in the reverse direction*. Thus what we want are augmenting paths in a different network, called the *residual* network with respect to the flow f.

Formally, let G be a network with capacity function c, and let f be a flow through G. We define the *residual network* of G with respect to f to be the network $G_f = (V_f, E_f)$ with capacity

Figure 5: The necessity of residual networks



The left figure features a simple network with a flow with value one, formed by augmenting one unit of flow along the path $s \to a \to b \to t$. There is no path from s to t along which the flow values are strictly less than the capacity; in particular, the only vertices reachable from s in this manner are s and b. But the cut $(\{s, b\}, \{a, t\})$ has capacity two, which is larger than one. And indeed, there is a flow with value two, as shown in the figure on the right.

function c_f defined as follows:[1,2]

- **(Vertices)** G_f is a network on the same vertices as G; *i.e.*, $V_f = V$.
- **(Forward edges)** Let e be an edge in E. We say that e is a *forward edge*. If the flow across e is equal to the capacity (*i.e.*, $f(e) = c(e)$), then we do *not* include e in E_f. Otherwise, we do include e in E_f. The capacity of a forward edge in the residual network is simply the *left-over capacity*,
$$c_f(e) \doteq c(e) - f(e)$$
- **(Backward edges)** Let e be an edge in E, and let $\overleftarrow{e}$ denote the *reverse* of e. *i.e.*, if $e = (u, v)$, then $\overleftarrow{e} = (v, u)$. We say that $\overleftarrow{e}$ is a *back edge*. If the flow across e is zero (*i.e.*, $f(e) = 0$), then we do not include $\overleftarrow{e}$ in E_f. Otherwise, we do include $\overleftarrow{e}$ in E_f. The capacity of a backward edge in the residual network is simply the *flow in the forward direction*,
$$c_f(\overleftarrow{e}) \doteq f(e)$$

Figure 6 sketches a flow and associated residual network in a simple network.

We also need to adapt our notion of an augmenting path to fit the new idea. One way to do this is to say that an augmenting path through the residual network G_f is a path P through G_f, from s to t, so that, for every edge e in P, the residual capacity of e is strictly positive. More succinctly, for every edge e in P, $c_f(e) > 0$. The old notion of augmenting path then corresponds to augmenting paths in G_f that only use forward edges.

The important property of residual networks is that augmenting paths in the residual network can always be used to extend a flow.

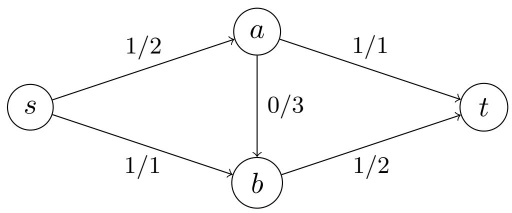
The augmentation itself is straightforward; we simply realize the intuition that capacity on backward edges represents the potential to subtract flow. Specifically, let P be an augmenting path

---

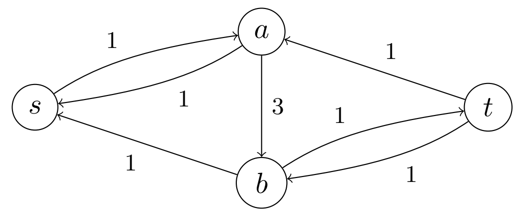
[1] If G has a pair of vertices u and v for which the edges (u, v) and (v, u) exist and have positive capacity, this scheme could introduce multiple edges of the form (u, v) or (v, u), which is technically not allowed in our definition of a directed graph. For the rest of this discussion, however, simply extend the definition of a network to allow for multiple such edges, and with different capacities on each. We then think of "the forward edge (u, v)" and "the backward edge $\overleftarrow{(v, u)}$" (which are both directed edges from u to v) as distinct edges in the residual network.

[2] This definition also can introduce edges which make the in-degree of s and the out-degree of t greater than zero. Strictly speaking, this doesn't fit our definition of a network; however, as long as we stick to the convention that the value of a flow is the *net* flow across a cut, then we can allow for edges to enter s or exit t without any problems.

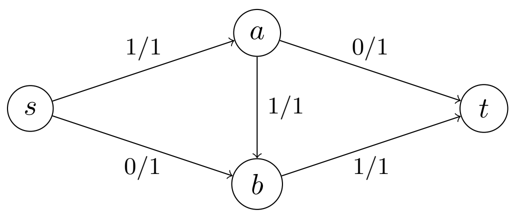
Figure 6: Residual networks



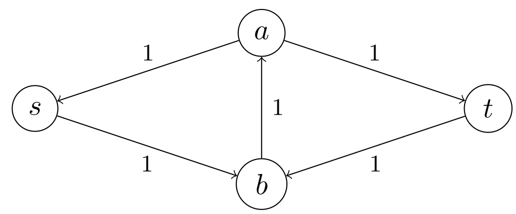
(a) A flow f through a network G



(b) The residual network G_f



(c) The flow from Figure 5.



(d) The residual network for the flow in Figure 5.

The residual network G_f is a network for which flows can be added to f without violating the capacity constraints in G. We view existing flow in f as having the *capacity to be undone*. Adding flow along backward edges really means subtracting flow from the forward edges.

in the residual network G_f, and let $\alpha > 0$ be so that $\alpha \leq c_f(e)$ for every edge e in P. We will define a function $f' : E \to \mathbb{R}$ which will be the flow we are interested in.

- For every forward edge e in P, we set $f'(e) = f(e) + \alpha$.
- For every backward edge $\overleftarrow{e}$ in P (where e is the corresponding forward edge in G), we set $f'(e) = f(e) - \alpha$.
- For every edge e for which both e and $\overleftarrow{e}$ are not in P, we just set $f'(e) = f(e)$.

An example of this augmentation (and a hint of what is to come) is given in Figure 7.

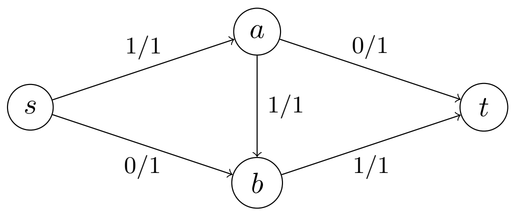
What remains to show is that the result of this augmentation is in fact a flow, *i.e.*, that it is nonnegative and satisfies the capacity and conservation constraints. We state and prove this formally in Proposition 2.

Proposition 2. *The function f' defined above is non-negative, and thus can be regarded as a function $f' : E \to [0, +\infty)$. Furthermore, f' is a flow.*

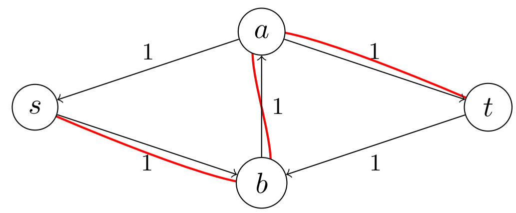
Proof. The function f' is nonnegative. Since $\alpha \geq 0$ and $f(e) \geq 0$ for all edges e, we have $f(e) \geq 0$ and $f(e) + \alpha \geq 0$ for all edges e. Thus we only have to worry about $f'(e)$ which was updated by a backward edge. But in this case, we know that $\alpha \leq c_f(\overleftarrow{e}) = f(e)$ for all backward edges $\overleftarrow{e}$, so $f'(e) = f(e) - \alpha \geq 0$.

The function f' additionally meets the capacity constraints. Since $f(e) \leq c(e)$ for all edges e, and $\alpha \geq 0$, it follows easily that $f(e) \leq c(e)$ and $f(e) - \alpha \leq c(e)$ for all edges e. Thus we only have to worry about the capacity constraints when f' was updated by a forward edge. But in this case, because $\alpha \leq c_f(e) = c(e) - f(e)$ for all forward edges, it follows that $f'(e) = f(e) + \alpha \leq c(e)$.

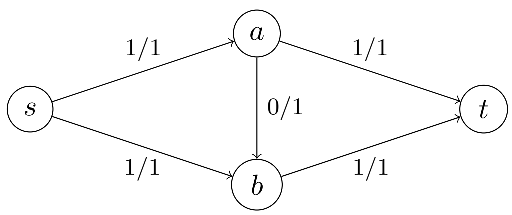
Figure 7: Augmenting paths in residual networks



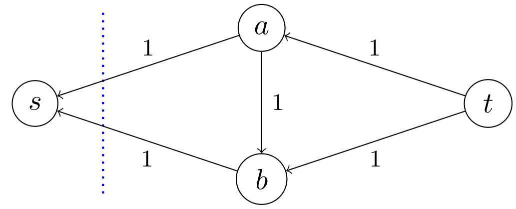
(a) The flow f from Figure 5.



(b) The residual network G_f with augmenting path.



(c) The flow f' after augmentation.



(d) The residual network $G_{f'}$ after augmentation, with cut.

The red path gives an augmenting path through the residual network G_f. We think of augmenting f along this path as pushing one unit of flow from s to t along the red path. In the case of the edge $a \to b$, this means *reducing* the flow. The result is a flow f' satisfying the conservation and capacity constraints. In this case, the vertices reachable from s in the residual network $G_{f'}$ define an s-t cut in G, the capacity of which is equal to the value of the flow f'.

Finally, the function f' satisfies the conservation constraints. For vertices v which are not visited in the path P, there are no changes to the total flow entering and exiting v; therefore, since f satisfies the conservation constraints, so too does f'.

For each vertex v along P besides s and t, P enters v along some edge a and exits v along some edge b. There are a few cases:

- If both a and b are forward edges, then the flow coming into v by a and the flow leaving v by b both increase by α.
- If both a and b are backward edges, then the flow coming into v by $\overrightarrow{b}$ and the flow leaving v by $\overrightarrow{a}$ both decrease by α. (Here $\overrightarrow{b}$ for a backward edge b just means its corresponding forward edge.)
- If a is a forward edge and b is a backward edge, then the flow coming into v by a increases by α and the flow coming into v by $\overrightarrow{b}$ decreases by α, resulting in no net change of flow entering v. Additionally, there is no change to the flow leaving v.
- If a is a backward edge and b is a forward edge, then the flow coming into v does not change. The flow leaving v by $\overrightarrow{a}$ decreases by α, and the flow leaving v by b increases by α, resulting in no net change of flow leaving v.

In any case, we have conservation of flow at v (even if v is visited multiple times by P).

Finally, the value of f' is α more than the value of f. This follows from applying a similar case analysis as above to the first edge of P, *i.e.*, the edge leaving s. $\square$

We are now ready to prove strong duality. We'll start by recalling the alternative we stated above, except adapted to the setting of residual networks. Let f be a flow through a network G, and G_f be the associated residual network. Let S denote the set of vertices reachable from s in the residual network along edges e with $c_f(e) > 0$. The alternative states that one of the following is true:

- The sink t is in S. In this case there is an augmenting path in G_f.
- The sink t is not in S. In this case the cut $(S, V \setminus S)$ is an s-t cut.

This suggests the following algorithm for computing the maximum flow in a network G with capacity function c.

1. Initialize f to be the zero flow, and compute G_f (which is just G).
2. While there is a path P from s to t in G_f, do the following: Compute the maximum amount of flow that can be pushed along P; this is just the minimum residual capacity of the edges in P. Update f and G_f according to pushing this amount of flow along the path P.
3. Output the flow f.

Note that the process of finding a path from s to t is not completely determined here; there may be many paths to choose from, and choices of different paths may lead to different answers. In fact, this algorithm is better thought of as a *family* of algorithms, where different subroutines for finding the augmenting path lead to different behaviors. It is referred to as the "Ford–Fulkerson scheme" (or Ford–Fulkerson algorithm), after its discoverers.

In general, however, these different choices will lead only to different performance guarantees. The *partial correctness* of the algorithm will not depend on the choice of augmenting path. We will focus on that now, since this is where strong duality comes in, and return to the termination and performance guarantees later.

Thus we want to prove the partial correctness of the above algorithm. So let's suppose that the algorithm was given a network G with capacity function c, and it computed a flow f. We know by the termination condition of the while loop in step 2 that there is no augmenting path in the residual network G_f. Thus we are in the second case of our alternative, and so we have an s-t cut (S, T), where S is defined by the vertices reachable from s in G_f. We want to show that this implies that f is a maximum flow. The following theorem does this:

Theorem 1. *The following are equivalent:*

1. *f is a maximum flow.*
2. *There is no augmenting path from s to t in G_f.*
3. *The cut (S, T), where S is the set of vertices reachable from s in G_f, is an s-t cut. It furthermore satisfies the property that, for every edge e in $E(S, T)$, $f(e) = c(e)$, and for every edge e in $E(T, S)$, $f(e) = 0$.*

Proof. We prove that (1) implies (2), (2) implies (3), and (3) implies (1), showing that all three statements are in fact equivalent.

(1) $\implies$ (2) We prove this by contraposition. Suppose that there is an augmenting path in G_f. Then by augmenting f along this path, f becomes a flow with strictly larger value, implying f was not originally a maximum flow.

(3) $\implies$ (1) The capacity of the cut (S, T) can be written as

$$\sum_{e \in E(S,T)} c(e) + \sum_{e \in E(T,S)} 0$$

which is term-by-term equal to $\mu(f, (S, T))$ by (3). By Proposition 1, we have $\mu(f, (S, T)) = \mu(f)$. Thus, by weak duality, every flow has value at most $\mu(f)$, which means that f is a maximum flow.

(2) $\implies$ (3) Let S be the set of vertices reachable from s in G_f, and T be its complement in V. By assumption, (2) tells us that t is not in S. Therefore (S, T) is an s-t cut.

For every edge (u, v) in $E(S, T)$, we know that u is reachable from s in G_f, and that v is not reachable from s in G_f. Thus the forward edge (u, v) cannot be present in $E(S, T)$. Since it is a forward edge, this means that $f((u, v)) = c((u, v))$.

For every edge (v, u) in $E(T, S)$, we know that v is reachable from s in G_f, and that u is not reachable from s in G_f. Thus the backward edge $\overleftarrow{(v, u)}$ cannot be present in G_f. Since it is a backward edge, this means that $f((v, u)) = 0$. $\square$

Phrased in more general terms, Theorem 1 tells us that the maximum value of any flow in any network is exactly equal to the minimum capacity of any s-t cut in this network. More succinctly, we have the following strengthening of weak duality:

$$\textbf{Max-Flow} = \textbf{Min-Cut}$$

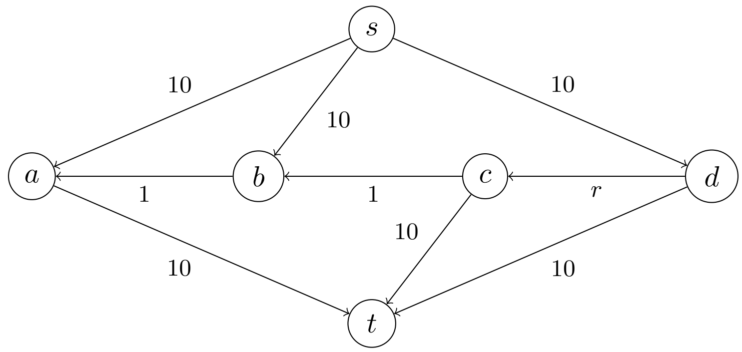
Finally, Theorem 1 tells us that the Ford–Fulkerson scheme, when it terminates, will always yield a maximum flow through its input network.

1.3 Termination and performance of the Ford–Fulkerson scheme

The final point of discussion for the Ford–Fulkerson scheme is its termination. It's easy to see that, with every augmentation, the value of the flow strictly increases: an augmenting path by definition has positive residual capacity on every edge, so increasing the flow by the smallest of these capacities is still a strictly positive increase. Furthermore, there is an upper bound on the total value of flow, witnessed by any s-t cut in the network. At an intuitive level then, it seems as though the Ford–Fulkerson algorithm will always terminate: it always makes progress, and it only has to make a finite amount of progress before it terminates.

Unfortunately, this is not always the case. In some specially-crafted networks and with poor choices of augmenting paths, the Ford–Fulkerson scheme can be made to run indefinitely. It always finds an augmenting path, which strictly increases the value of the flow, but this increase may shrink over time. For instance, one might imagine the first augmentation increases the flow value by 1, the second by 1/2, the third by 1/4, and so on, but the maximum flow has value 2 or larger. More concretely, the network given in Figure 8 gives an example for which this actually happens.

Figure 8: An example on which the Ford–Fulkerson scheme might not terminate



The number r is the (unique) positive real number satisfying $1 - r = r^2$ This value r additionally satisfies $1 - 2r = r^3$, $r^k(1 - r) = r^{k+2}$, and $(1 - r) < r < 1$. It is easy (if tedious) to use these facts to check that one can repeatedly augment using the sequence of paths P_1, P_2, P_1, P_3, defined by $P_1 = s \to d \to c \to b \to a \to t$, $P_2 = s \to b \to c \to d \to t$, and $P_3 = s \to a \to b \to c \to t$.

On the other hand, there are settings in which we can realize our intuition. The missing piece to our approach is that we need for there to be *finitely* many values between 0 and the value of the maximum flow. This suffices since, if we start with a flow of value 0, and each augmentation strictly increases the value of the flow, then the value of the maximum flow must eventually be reached. The example in Figure 8 takes advantage of the fact that there are infinitely many real values between 0 and any positive number.

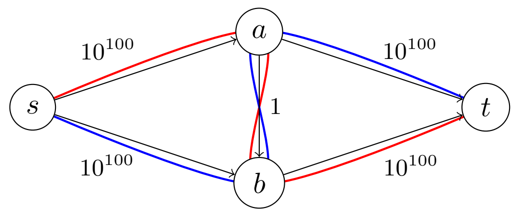
However, the counterexample has to do so in a rather careful way. The reason is that there is a relationship between the capacities in the original network and the values of the flows seen during the Ford–Fulkerson algorithm. To demonstrate this, let's focus on the case where all of the capacities in the input network are integers. When this is the case, it's easy to see that the Ford–Fulkerson algorithm will push an integral amount of flow along the first augmenting path. When this happens, the residual capacities are either increased or decreased by this integral value. Since the set of integers is closed under addition and subtraction, this means that all the residual capacities in the network will again be integers. Thus the Ford–Fulkerson scheme will again push an integral amount of flow along its second augmenting path, and so on. In other words, the value of the flow is *always an integer*. Since the number of integers between 0 and a fixed positive integer is finite, this means that the Ford–Fulkerson algorithm will always terminate on networks with integral capacities. More formally, we have Claim 1.

Claim 1. *If the input network's capacities are integers, then the Ford–Fulkerson scheme will always terminate. In particular, if the maximum flow has value F, then the Ford–Fulkerson scheme terminates after at most F augmentations. Furthermore, the flow computed by the Ford–Fulkerson scheme will have integral values.*

Proof. As argued above, if the input network's capacities are integers, it follows that every augmentation increases the value of the flow by an integer. Since each augmentation strictly increases the value of the flow, this means that the value of the flow always increases by at least 1.

This means that after at most F augmentations, the Ford–Fulkerson scheme will have found a flow with value at least F. Since the maximum flow has value at most F, it follows that after at most F augmentations, the Ford–Fulkerson scheme will find a maximum flow.

Figure 9: An instance of network flow for which the Ford–Fulkerson algorithm is slow



The Ford–Fulkerson algorithm may choose to augment along the red path, then the blue path, then the red path, etc. The effect is that the intermediate flow values only increase by one at each augmentation. The result is that the Ford–Fulkerson algorithm needs $2 \cdot 10^{100}$ iterations to find the maximum flow in this network.

The fact that the flow returned by Ford–Fulkerson is integral is a consequence of the fact that every augmentation increases the values of the flow by integral amounts. $\square$

Since the augmenting path can be found using *e.g.*, breadth-first search, Claim 1 implies that the Ford–Fulkerson scheme can be implemented to run in time $O(F \cdot (n+m))$ on any network with at most n vertices and m edges and maximum flow value F.

This is not a particularly nice upper bound. In particular, it is only a pseudo-polynomial time bound. However, without specifying any information on how the augmenting paths are chosen, this is essentially the best bound achievable by the Ford–Fulkerson scheme. Consider for instance Figure 9, where $2 \cdot 10^{100}$ augmentations are needed when the augmenting path is chosen as described in the figure. Thus a natural question is where there are smart ways to choose the augmenting paths, so that the running time does not depend so poorly on the value of the flow.

A better bound It turns out that there are a few approaches that succeed at this. One approach is to simply always choose the augmenting path to be one for which the *most* flow can be pushed. Finding this path can still be done in linear time (using the routine for median-finding as a subroutine!) for each iteration, and the end result is an overall $O(\log(F)(n+m))$ running time. (We leave out the analysis.)

Another approach completely removes the dependence on F from the running time. This approach is deceptively simple—the augmenting path is found by simply doing a breadth-first search! The difficulty lies in the analysis, however, which we do not cover here. The end result, however, is a bound of $O(n \cdot m)$ on the number of iterations, and thus a bound of $O(n \cdot m \cdot (n+m)) = O(nm^2 + n^2 m)$ on the running time. This approach is generally referred to as the Edmonds–Karp algorithm, after the authors who proved this bound on the running time.

There are yet more approaches to the network flow problem. An algorithm known as "Dinic's Algorithm" is a variation on the Ford–Fulkerson scheme, and can be implemented straightforwardly to run in time $O(n^2 m)$. Using more sophisticated data structures, this can be reduced to $O(nm \log(n))$, though this asymptotic performance is rarely worth the extra effort.

A rather different approach, known both as the "push–relabel algorithm" and the "preflow–push algorithm", achieves a running time of $O(n^2 \sqrt{m})$ when implemented simply. Sophisticated data structures can be employed here as in Dinic's Algorithm to bring the running time down to $O(nm \log(n^2/m))$.

2 Matching

Our first application of network flow is to the bipartite maximal matching problem. In a bipartite maximal matching problem, the input is a bipartite graph $G = (L \cup R, E)$. A matching in this graph is a subset $M \subseteq E$ of the edges with the property that each vertex is incident to at most one edge of M; if $(u, v) \in M$, then one thinks of v as being matched to u, and vice-versa. A maximal matching is a matching of maximum size. Formally, we have the following problem description:

Input: A bipartite graph $G = (L \cup R, E)$

Output: A matching M of maximum size

As promised, the bipartite maximum matching problem can be solved via reduction to network flow. The reduction is fairly straightforward: the main intuition is to think of a match (u, v) as being represented by a unit of flow going from u to v through G. If we can accomplish this, then flows will correspond to matchings, and maximal flows will correspond to maximal matchings.

Since we want flows to go "through" G, it makes sense to have the network be based on G itself. Toward this end, we'll just add a source s and a sink t to the vertices, add edges from the source to the vertices in L, edges from the vertices in R to the sink, and finally direct all the edges of G from L to R. The end result is that every path from s to t has to travel into a vertex in L, through an edge of G, into a vertex in R, and finally on to t. We refer to the network that results from this as G'. Pictorially, we have Figure 10.

To enforce that the flow represents a matching, we need to make sure there is at most one unit of flow through each vertex of L and R. We can accomplish this by imposing a capacity constraint of 1 on all the edges of the form (s, u) and (v, t), where u is in L and v is in R. This will suffice for all of our arguments (as long as the edges from E have capacity at least one), but we still have to choose capacities for the edges in E. It will simplify our arguments in Section **??** if we can suppose that the edges in E have *infinite* capacity. Strictly speaking, however, we can't do that, since the capacity function has to have the form $E \to [0, +\infty)$ according to our definition of a network.[3]

However, the only reason to give these edges infinite capacity is just to make sure they don't appear in any minimum cut. We can accomplish the same thing by simply giving these edges a sufficiently large finite capacity. Note that there is already a finite-capacity s-t cut, namely (S, T) where $S = \{s\}$ and $T = L \cup R \cup \{t\}$. The capacity of this cut is the number of vertices in L, $|L|$. What this means for us is that any edge with capacity larger than $|L|$ cannot appear in a minimum cut. Thus if we give the edges in E a capacity of $|L| + 1$, then they will behave identically to infinite capacity edges for our purposes. So for now on, we will simply regard these edges as having infinite capacity.

This completes our construction of the flow network G'. We now wish to formally argue that G' realizes our intuition that units of flow correspond to matches in matchings. We formalize this in the following claim:

Claim 2. *There is a one-to-one and onto correspondence between matchings in G and integral flows in G'. Furthermore, maximal matchings correspond to maximal flows and vice-versa.*

[3] It is possible and quite reasonable to extend the notion of a network to allow for infinite capacity edges. We didn't do that, because it requires adding a special case to all of the duality theory to handle the possibility of infinite flow.

Proof. We give both directions of the correspondence, and then observe that the two directions are inverses of each other, which implies the correspondence is one-to-one and onto. We then inspect the correspondence to see that matchings of size k correspond with flows of value k, which implies that the maximal matchings correspond to maximal flows. Figure 10 serves as a useful visual aide for understanding the correspondence.

Matchings → Flows This direction of this correspondence is a straightforward interpretation of our intuition that units of flow are the same as matches. Let M be any matching in G. We create a flow f through the network G' whose value is the number of matches in M.

For each edge (u, v) in M, we want f to have a unit of flow across the edge (u, v) in G'. Of course, this flow has to come from s and end in t. The only way to do this is to have a unit of flow going from s to u and going from v to t. So let $f_{(u,v)}$ represent this single unit of flow from s to u to v to t. Note that it is itself a flow through G', since it satisfies the capacity and conservation constraints.

To construct the flow f, we simply superimpose all of these flows. Symbolically, $f = \sum_{e \in M} f_e$. The function f is indeed a flow:

- Since it is a sum of flows that satisfy the conservation constraints, it satisfies the conservation constraints as well.
- The only way to violate the capacity constraints is to send multiple units of flow along the same edge. But since M is a matching, each vertex appears in at most one edge in M. This means each edge of the form (s, u) or (v, t) has positive flow in at most one f_e, so the capacity constraints on these edges are satisfied.

Moreover, the flow f is integral, because it is a sum of integral flows.

Flows → Matchings For the other direction of the correspondence, we want to give an inverse to the above map from matchings to integral flows. We can do this by observing the following structure in our network:

- For every vertex u in L, there is only one incoming edge (from s), and this incoming edge has capacity one.
- For every vertex v in R, there is only one outgoing edge (to t), and this outgoing edge has capacity one.

These conditions imply that there is at most one unit of flow entering each vertex in L, and at most one unit of flow leaving each vertex in R. By conservation, this means that there is at most one unit of flow leaving each vertex in L, and at most one unit of flow entering each vertex in R. Thus, in any *integral* flow, each vertex in L sends positive flow to *at most one* vertex in R, and each vertex in R receives positive flow from at most one vertex in L.

This tells us how to define our matching: Let f denote any integral flow through G'. Then we define M to be the set of pairs (u, v) with u in L and v in R so that f sends positive flow from u to v. This is a matching by the properties we stated above. It is a matching in G because the units of flow in f can only be sent along edges present in G.

It's easy to check that these two correspondences are inverses of each other: if we start with a matching, apply the first transformation, then apply the second transformation, we get the matching we started with; if we start with a flow, apply the second transformation, then apply the third transformation, we get the flow we started with.

Finally, a straightforward verification realizes that flows of value k correspond with matchings of size k. $\square$

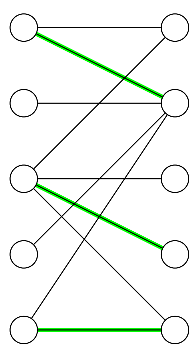
Thus, with Claim 2 in hand, we have a complete algorithm for bipartite matching. The algorithm is simply to construct the flow network G' above, and then run an algorithm for network flow. Using Ford–Fulkerson with a linear-time augmenting path finding subroutine, we get a running time bound of $O(M \cdot (n+m))$ when G has n vertices and m edges, and where M denotes the size of the maximal matching. Since $M \leq n$, we can eliminate the dependence on M to achieve a running time bound of $O(n(n+m))$.

Note that, for this specific problem, this application of Ford–Fulkerson out-performs the strongly polynomial-time bounds for general network flow.

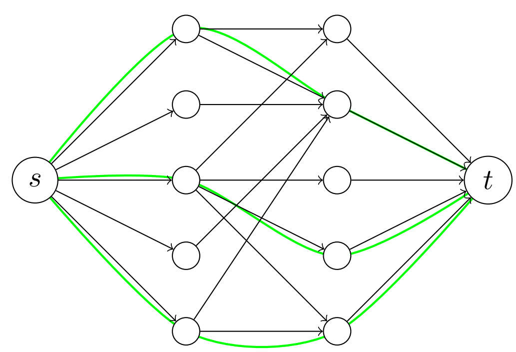
2.1 Hall's marriage theorem

(Forthcoming: Hall's marriage theorem as a consequence of strong duality)

Figure 10: Flow network for maximal matching



(a) Bipartite graph



(b) Flow network

The bipartite graph on the left is converted into a flow network on the right, with the property that matchings in the bipartite graph correspond to flows in the flow network. All the edges in the flow network which are incident to s or t have capacity 1; the remaining edges have infinite capacity. The matching in green on the left is represented by the green flow on the right.

3 Edge-disjoint paths

(Forthcoming: the edge-disjoint paths problem and its reduction to network flow)

4 Image segmentation

(Forthcoming: the image segmentation problem and its reduction to network flow)

5 Project selection

The next problem we will reduce to network flow is the problem of Project Selection. The basic setup here is that some engineers have some projects that they would like to complete. Each project requires a set of tools in order to be completed, and these tools have costs associated with them that the engineers would like to keep small. However, tools can be re-used between projects, and have no additional cost for multiple uses.

Formally, there is a set of projects P and tools L, as well as a dependency relation, $R \subseteq P \times L$, where (p, ℓ) is in R if and only if the project p requires the tool ℓ to be completed. Each project p has a value v_p, and each tool ℓ has a cost c_ℓ. For subsets of projects $P' \subseteq P$, we let $\Gamma(P')$ denote the set of tools which are required by some project in P'; symbolically,

$$\Gamma(P') = \{\ell : \left(\exists p \in P'\right)(p, \ell) \in R\}$$

The goal of the engineers is to find a set of projects P^* so that the total value of the projects in P^* minus the total cost of the tools required by projects in P^* is maximized. Symbolically, we want to find a set P^* for which $P' = P^*$ achieves the maximum in the expression

$$\max_{P' \subseteq P} \sum_{p \in P'} v_p - \sum_{\ell \in \Gamma(P')} c_\ell \tag{2}$$

Reduction to network flow We will solve this problem via reduction to minimum cut. The intuition is that we want to partition the jobs and tools into those that we do use and those that we don't use, and cuts in graphs are a way of expressing this. The rest of the reduction is just figuring out how to represent our problem as a flow network so that s-t cuts correspond to ways of selecting projects and tools such that finding a minimum s-t cut is equivalent to finding an optimal set of projects and tools.

A first idea to try is just to make a flow network G whose vertices are a special source and sink vertices, s and t respectively, as well as a vertex for each project and a vertex for each tool. We can then think of an s-t cut (S, T) as telling us to choose the projects and tools whose vertices appear in S. This will be the right idea for us, but we still have to specify the edges and their capacities in this flow network.

Our first step is to model the constraints of this problem. The constraints for this problem only come from the dependency relation R: for a project p and tool ℓ for which ℓ is required to complete p, we cannot allow s-t cuts (S, T) which have p in S but ℓ not in S. One way to do this is to introduce an infinite capacity edge from p to ℓ in G for every pair (p, ℓ) in R. This has the effect that, whenever an s-t cut (S, T) has p in S and ℓ in T, the capacity of this cut is infinite. As long as we design the network to have some finite-capacity cut, this means that no minimum cut will violate the constraints from R, which is good enough for our purposes.

Now it only remains to make sure that minimum cuts maximize the value in (2). Making a minimization objective achieve a maximization goal seems difficult at first, but this turns out to be a simple matter of algebra for our case. Note that for any real-valued function f, the expression "min f" has the same value as "$-$ max $(-f)$", and the expression "max $(f+\alpha)$" has the same value as "$\alpha +$ max f" for any constant α. Furthermore, the minima and maxima are attained at the same points of the domain of f. These facts give us a fair amount of flexibility in adjusting (2) to suit our situation.

Specifically, we'll use the first property to turn (2) into a maximization objective:

$$(2) = -\min_{P' \subseteq P} \sum_{\ell \in \Gamma(P')} c_\ell - \sum_{p \in P'} v_p \tag{3}$$

This can work well for us for the purposes of computing a minimum cut. We can account for the contribution from the first sum by simply adding edges from the vertices corresponding to tools to the sink vertex t. The capacity of the edge added from the tool ℓ to t is just c_ℓ. It's easy to verify that for any s-t cut (S, T), the contribution of these new edges to the capacity of this cut is exactly $\sum_{\ell \in S} c_\ell$. Since we are thinking of the S-part of s-t cuts as telling us which tools to select, *i.e.*, the tools in $\Gamma(P')$ for the projects P' selected by S, these new edges exactly account for the costs of tools.

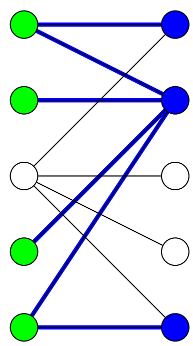
The projects, however, seem less hopeful. The capacity of a cut is simply the sum over some positive quantities, where as $-\sum_{p \in P'} v_p$ is the negative of this. This is where our second property about min's and max's comes into play. The value $\sum_{p \in P} v_p$ is just a constant for the purposes of (2) and (3). This means we can rewrite (3) as

$$(3) = -\sum_{p \in P} - \min_{P' \subseteq P} \sum_{\ell \in \Gamma(P')} c_\ell + \sum_{p \in P \setminus P'} v_p \tag{4}$$

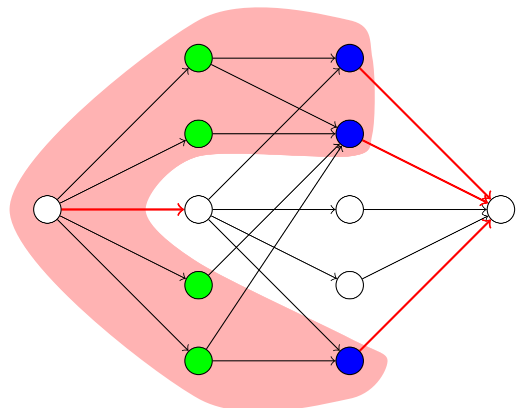
So now we can apply the same general idea as before: since we're thinking of the T-part of s-t cuts as telling us which projects *not* to select, we want to count the contribution of v_p to the value of a cut only if it puts p on the T-side of the cut. Thus we want to add edges from s to p for every project p, and give these edges capacity v_p.

These are all the edges that we will need. They are represented pictorially in Figure 11. Our reduction from project selection to network flow is simply to create the flow network described above and run a maximum flow algorithm on it. The value of the maximum flow, which is equal to the value of the minimum cut, isn't equal to the optimal project selection cost, but we can plug the value into (4) and determine the exact cost of the optimal project selection. To recover an optimal selection of projects and dependencies, we just find a minimum cut in the flow network, and using the S-side of the cut to tell us what projects to select.

Figure 11: Flow network for project selection



(a) A project/tool dependency graph



(b) Corresponding flow network

The bipartite graph on the left represents the project/tool dependency relation R. The vertices on the left are the projects, and the vertices on the right are tools. An edge (p, t) exists exactly when (p, t) is in R. An example selection of projects is given in green, and the corresponding tools are given in blue. The figure on the right gives the flow network corresponding to this instance. The edges of the form (s, p) have capacity v_p; the edges of the form (ℓ, t) have capacity c_ℓ; and the edges of the form (p, ℓ) have capacity $+\infty$. The cut corresponding to the selection of projects and tools is indicated in red.

6 Circulations

(Forthcoming: the problem of deciding existence of circulations in a circulation network, and its reduction to network flow.)

7 Survey Design

(Forthcoming: the problem of survey design, and its reduction to circulations)

8 Airline Scheduling

(Forthcoming: a simple airline scheduling problem, and its reduction to network flow)