# Combinatorial Optimization

Jacob Denson

May 17, 2017

# Table Of Contents

# Chapter 1

# Matchings and Flows

## 1.1 Bipartite Matching

Let $G = (V, E)$ be a graph. A **matching** is $M \subset E$ such that no vertex in $V$ is the endpoint of more than one edge in $M$. $M$ is maximal if $|M| \geqslant |M'|$ for any other matching $M'$. The maximal matching problem asks us to find a fast algorithm to find a maximum matching in any graph.

There is a polynomial time algorithm which can find matchings on any graph, but the problem is much more simple if $G$ is bipartite – that is, if we may partition $V$ into the disjoint union $W \amalg U$ of two sets of vertices, such that every edge in $E$ contains a point in $W$ and a point in $U$. The bipartite matching asks us to find a maximal matching in a bipartite graph.

Suppose we are given a particular matching in a bipartite graph. Is there a reliable procedure to improve the matching? We could proceed by a guess and check method – we remove an edge in our matching, then try and add an edge using one of the vertices which has been freed up. If this edge cannot be taken because the end point is attached to an edge still in the matching, we remove that edge, freeing up more vertices. If we ever end up adding more edges than we started with (which occurs when we add an edge not attached to any points in the current matching), then we find a matching with an extra edge than before. This process is formalized by the 'augmenting paths' construction.

Given a particular matching $M$, construct a directed graph $G_M = (V \cup \{s, t\}, E_M)$, where $s$ and $t$ are new vertices. Let $w \in W$, $u \in U$. Construct the edges $E_M$ such that

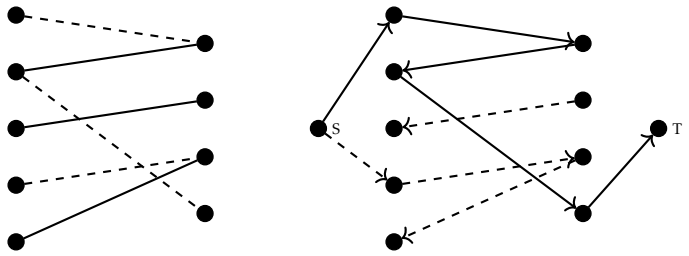


Figure 1.1: The left graph shows a matching in a graph of cardinality 3. The solid lines correspond to the edges chosen. On the right is the augmenting path graph corresponding to the matching. It contains a path from $s$ to $t$, hence there is a way to improve the matching to a match of cardinality 4.

- $wu \in E_M$, if $wu \in M^c$.
- $uw \in E_M$, if $uw \in M$.
- $sw \in E_M$, if $w$ is not the endpoint of any edge in $M$.
- $ut \in E_M$, if $u$ is not the endpoint of any edge in $M$.

Let $(s, w_1, u_1, \ldots, w_n, u_n, t)$ be a path in $G_M$ from $s$ to $t$, with $w_i \in W$, $u_i \in U$. Then $w_i u_i \in M^c$ and $u_i w_{i+1} \in M$, and $w$ and $u$ are both unused in $M$. Let $M'$ be obtained from $M$ by removing all edges of the form $u_i w_{i+1}$, and adding all edges of the form $w_i u_i$. It is easy to convince yourself that $M'$ is a matching, with one more edge than $M$.

Conversely, suppose that there is no path from $s$ to $t$ in $G_M$. We claim $M$ is then a maximal matching on $G$. Otherwise, we would have a matching $M^*$ with $|M^*| > |M|$. Consider the multigraph $H = (V, M \amalg M^*)$. Every vertex in $V$ has at most degree two (for it can only be the endpoint of a single edge in $M$, and a single endpoint in $M^*$), and thus $V$ breaks into cycles and paths. For any cycle $C$, we find $|M \cap C| = |M^* \cap C|$, for the edges in the cycle must alternate being being in $M$ and $M^*$. Since $|M^*| > |M|$, there must be a path $P$ in $H$ containing more edges in $M^*$ than in $M$. $P$ must therefore begin and end with a path in $M^*$, and we see that this path is exactly an augmenting path for $M$.

This argument justifies the correctness of the following matching algorithm, a variant of the 'Ford Fulkerson Algorithm'. We take any initial

matching $M$. Then, we construct the augmenting path graph $G_M$, and use any of the standard algorithms to find a path from $s$ to $t$. If we have $2n$ nodes, and $m$ edges, then we can find a path in $O(m)$ time, and we can bound the number of augmentations we require to $O(n)$, so the algorithm runs in $O(nm)$ time.

## 1.2 Vertex Covers

A vertex cover $C \subset V$ is a set of points containing at least one endpoint of every edge in $E$. Clearly if $M$ is a maximal matching in $G$, then $|M| \leqslant |C|$, because there is an injective function $f : M \to C$, obtained by mapping an edge to one of its endpoints which lies in $C$.

**Theorem 1.1** (König - Egerváry)**.** *Let $G = (W \amalg U, E)$ be bipartite, and let $M$ be a maximal matching in $G$. Let $Z \subset V \cup \{s,t\}$ be the nodes reachable from $s$ in the augmenting path graph $G_M$. Then $C = (W - Z) \cup (U \cap Z)$ is a cover with $|C| = |M|$.*

*Proof.* First we prove $C$ is a cover. Let $wu \in E$ be arbitrary.

- Suppose that $wu \in M$. If $u \in Z$, then $w \in Z$, because $uw$ is an edge in $G_M$, so that if $w \notin C$, $u \in C$.
- Suppose that $wu \notin M$. Then $wu$ is an edge in $G_M$, so that if $w \in Z$, $u \in Z$, and therfore if $w \notin C$, $u \in C$.

Now we show $|U| = |M|$. First, note that $C$ contains *exactly one* of each of the endpoints of $M$, so $|U| \geqslant |M|$. Conversely, let $wu \in E - M$ be arbitrary. If $u \in C$, then $u$ is reachable from $s$, and hence $u$ is the endpoint of an edge in $M$, for otherwise $t$ is reachable from $s$ through $u$. Conversely, if $w \in C$, then $w$ is not reachable from $s$, hence $w$ is on an edge in $M$ (for otherwise there is an edge directly to $w$ from $s$ in $G_M$). This shows that every vertex in $C$ is on one of the edge in $M$, and hence $|U| = |M|$. □

This gives us a method of finding a min cover in a graph. One equivalent form of this theorem is very useful. Given an undirected graph $G = (V, E)$, let $N$ be the neighbour function $N(S) = \{v \in V : sv \in E, s \in S\}$.

**Theorem 1.2** (Hall)**.** *Let $G = (W \amalg U, E)$ be a bipartite graph. Then $G$ has a matching $M$ which covers $W$ if and only if $|N(S)| \geqslant |S|$ for all $S \subset W$.*

*Proof.* Certainly if a matching $M$ covers $W$, then it induces an injective function from $S$ to $N(S)$ for each $S \subset W$. Conversely, assume no matching of $M$ covers $W$. Then König's theorem tells us there is a cover $C$ of $W \amalg U$ containing less than $|W|$ nodes. Then write $A = C \cap W$, $B = C \cap U$. We find $N(W - A) \subset B$, so $|N(W - A)| \leqslant |B| < |W| - |A|$. □

## 1.3 The Hopcroft-Karp Matching Algorithm

The augmenting path approach to finding a maximal matching in a Bipartite graph improves the length of a candidate matching by one for each cycle of the algorithm. If we were able to consistently find augmentations which improved matchings by more than one edge, we could find a faster algorithm for maximal matching.

If $M$ is a matching, and $N_M$ be the length of a shortest alternating path in $G_M$, we shall let a collection of vertex disjoint alternating paths $P_1, \ldots, P_n$ be called **M-blocking** if $|P_i| = N_M$, and every alternating path of length $N_M$ shares a vertex with one of the $P_i$. The next lemma is a simple generalization of the augmenting paths approach.

**Theorem 1.3.** *If $P_1, \ldots, P_n$ are vertex disjoint $M$-alternating paths, then*

$$M' = (M - \cup P_i) \cup (\cup P_i - M)$$

*is a matching with $|M'| = |M| + n$.*

The Hopcroft-Karp algorithm proceeds as in Ford-Fulkerson, albeit improving matchings by taking $M$-blocking paths as augmentations, improving matchings in chunks. We shall discover a method to find an $M$-blocking in linear time, and also show the chunk method yields an asymptotic speedup in the number of cycles to find a maximal matching, so that Hopcroft-Karp is objectively faster than Ford-Fulkerson.

**Lemma 1.4.** *If $M$ is a matching, and $P_1, \ldots, P_n$ an $M$-blocking set of paths. Then $M' = (M - \cup P_i) \cup (\cup P_i - M)$, and $N_{M'} \geqslant N_M + 1$.*

*Proof.* Let $Q = (v_1, \ldots, v_k)$ be an alternating path in $G_{M'}$ with $|Q| \leqslant |N_M|$, where $v_1$ and $v_k$ are $M'$ exposed. Then $v_1$ and $v_k$ are also $M$ exposed, because switching from $M$ to $M'$ only removes exposed vertices, not adds them.

Let $H$ be the directed graph with edges taken from all the edge lying on the $P_i$, and the directed edges of $Q$, except that we remove both copies of duplicate edges. The reverse $vu$ of any edge $uv$ in $Q$ that is not an edge on some $P_i$ must be on some $P_j$ □

## 1.4 Network Flow

Let $G = (V, E)$ be a directed graph with two identified vertices $s \neq t$. Let $\mu : E \to \mathbf{R}^+$ be a function measuring the 'capacity' of each edge in the graph (Thinking of the edges as if they were 'pipes' which can only carry a certain throughput). Define, for $U \subset V$,

$$\delta_{\text{out}}(U) = \{w \in V - U : vw \in E\} \qquad \delta_{\text{in}}(v) = \{w \in V - U : wv \in E\}$$

A flow is a mapping $f : E \to \mathbf{R}^+$ such that $0 \leqslant f \leqslant \mu$, and for any $v \neq s, t$,

$$f(\delta_{\text{out}}(v)) = f(\delta_{\text{in}}(v)) \ ^*$$

a relation known as the flow conservation law. The aim of the maximum flow problem is to find $f$ such that the value function

$$\text{val}(f) = f(\delta_{\text{out}}(s)) - f(\delta_{\text{in}}(s))$$

is maximized. This is essentially the amount of flow which is created at $f$. It is also the amount of flow which is 'absorbed' at $t$, because

$$\begin{aligned} f(\delta_{\text{out}}(t)) - f(\delta_{\text{in}}(t)) &= f(\delta_{\text{out}}(t)) - f(\delta_{\text{in}}(t)) + \sum_{v \neq s,t} f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v)) \\ &= \sum_{v \neq s} \left( \sum_{vw \in E} f(vw) - \sum_{wv \in E} f(wv) \right) \\ &= \left( \sum_{\substack{v \neq s \\ vs \in E}} f(vs) + f(ss) \right) - \left( \sum_{\substack{v \neq s \\ sv \in E}} f(sv) + f(ss) \right) \\ &= -[\delta_{\text{out}}(f, s) - \delta_{\text{in}}(f, s)] \end{aligned}$$

Thus the maximum flow problem tells us a method of maximizing the amount of flow which gets to $t$.

*We extend functions $g : X \to \mathbf{R}$ to $g : 2^X \to \mathbf{R}$ by defining $g(A) = \sum_{x \in A} g(x)$

There is an interesting relation between flows on graphs, and another structure on these graphs known as a cut, which we will take full advantage of in finding solutions to the max flow problem. Define a $(s,t)$ cut on a directed graph $G = (V,E)$ with vertices $s$ and $t$, to be a partition of $V$ into two sets, one containing $s$, and one containing $t$. It is however more simple to consider a cut to be a subset $C$ of $V$ containing $s$, but not $t$. If we have a capacity function $\mu$, then we define the value of the cut to be $\text{val}(C) = \mu(\delta_{\text{out}}(C))$. The min cut problem is to find a cut of smallest value.

**Lemma 1.5.** *If $f$ is a flow, and $C$ is an $(s,t)$ cut, then $val(f) \leqslant \mu(\delta_{out}(C))$.*

*Proof.* Since $t \notin C$, similar manipulations to the ones above show that

$$\begin{aligned}\text{val}(f) &= \sum_{v \in C} f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v)) \\ &= f(\delta_{\text{out}}(C)) - f(\delta_{\text{in}}(C)) \\ &\leqslant \mu(\delta_{\text{out}}(C))\end{aligned}$$

Note that we obtain equality if $f(\delta_{\text{out}}(C)) = \mu(\delta_{\text{out}}(C))$ and $f(\delta_{\text{in}}(C)) = 0$, in which case $f$ must be a maximal flow, and $C$ a min cut. □

As with maximal matchings, we attempt to find a maximal flow by finding ways of augmenting suboptimal flows to a maximal solution. Given a flow $f$, we construct the residual graph $G_f$, which has the same vertices as $G$, but whose edges are defined to be the set

$$E_f = \{uv \in E : f(uv) < \mu(uv)\} \coprod \{vu : uv \in E, f(uv) > 0\}$$

We shall denote the element $vu$ in the second set which makes up $E_f$ by $\overleftarrow{uv}$, since we obtained in from an edge $uv$ in $G$. Also define a capacity function

$$\mu_f(uv) = \mu(uv) - f(uv) \qquad \mu_f(\overleftarrow{uv}) = f(uv)$$

Suppose we can find a simple path $P$ from $s$ to $t$ in $G_f$, and define $\alpha = \min_{e \in P} \mu_f(e)$. Consider a new flow

$$f'(uv) = \begin{cases} f(uv) + \alpha : uv \in P \\ f(uv) - \alpha : \overleftarrow{uv} \in P \end{cases}$$

First we show that $f'$ is a flow. By the choice of $\alpha$, $0 \leqslant f' \leqslant \mu$. For each vertex $v \neq s, t$, we must show that flow conservation still holds. Let $e_1, e_1', \ldots, e_n, e_n'$ be the edges in $G$ containing $v$ as an endpoint obtained from $P$, by reversing edges of the form $\overleftarrow{uv}$, considered in the order they appear in $P$. We may write pair up the edges in this manner because, for every edge that enters $v$ in $P$, there must be an edge that leaves $v$ in $P$. Let

$$S_i = \begin{cases} f(e_i) - f'(e_i) & e_i = uv \\ f'(e_i) - f(e_i) & e_i = vu \end{cases}$$

Define $S_i'$ similarily for $e_i'$. Then, since no edges are repeated,

$$f'(\delta_{\text{out}}(v)) - f'(\delta_{\text{in}}(v)) = f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v)) + \sum_{i=1}^{n} (S_i + S_i')$$

We now show $S_i + S_i' = 0$ for each $i$. This breaks into several cases.

- If $e_i = uv$, $e_i' = vw$, in which case $f'(e_i) = f(e_i) + \alpha$, $f'(e_i') = f(e_i') + \alpha$, and $S_i + S_i' = \alpha - \alpha = 0$.
- If $e_i = uv$, $e_i' = wv$, then $e_i'$ was obtained from an edge of the form $\overleftarrow{vw}$ in $P$, in which case $f'(e_i') = f(e_i') - \alpha$, and so $S_i + S_i' = -\alpha - (-\alpha) = 0$.
- If $e_i = vu$, $e_i' = wv$, then $e_i$ and $e_i'$ were obtained from reversed edges in $P$, and so $S_i + S_i' = (-\alpha) - (-\alpha) = 0$.
- If $e_i = vu$, $e_i' = vw$, then $e_i$ was obtained from reversing edges, and so $S_i + S_i' = (-\alpha) + \alpha = 0$.

And it follows, because flow conservation holds at $v$ for $f$, that it holds at $v$ for $f'$ as well. Finally, we find that $\text{val}(f') = \text{val}(f) + \alpha$, which can be shown by performing an analysis, similar to $v$ above, for $s$, noting that $s$ will have an extra edge at the beginning of the path, which causes the extra $\alpha$. All that remains is to show this augmentation method yields a maximum flow in all cases, after enough iterations.

Let $f$ be a flow for which $G_f$ has no augmenting path. Let $U$ be the nodes reachable from $s$ in $G_f$, easily verified to be a cut.

$$\text{val}(f) = \mu(\delta_{\text{out}}(U))$$

proving maximality of the flow, and minimality of the cut $U$. To see this, let $uv \in \delta_{\text{out}}(U)$. Then $f(uv) = \mu(uv)$, for otherwise $v$ would be reachable from $s$ in $G_f$. Similarily, we must have $f(\delta_{\text{in}}(U)) = 0$, for if $vu \in \delta_{\text{in}}(U)$, and $f(vu) > 0$, then $\overleftarrow{vu} \in G_f$, and so $v$ is reachable from $s$, a contradiction. We conclude that

$$\text{val}(f) = f(\delta_{\text{out}}(U)) - f(\delta_{\text{in}}(U)) = \mu(\delta_{\text{out}}(U))$$

which shows that our augmenting paths approach works. Notice that if all edge weights are integral, then the augmenting flows are always integral, and thus there exists a maximal flow with interal weights.

The Ford Fulkerson algorithm solves the max flow problem by repeatedly augmenting an initial flow. In principle, this approach is correct, but if you do not find a residual path in a smart way, this algorithm does not terminate in polynomial time for all inputs. If all edge weights are integers $n_1, \ldots, n_m$, then each augmentation increases the value of the flow by at least the greatest common factor of the $n_i$. Since there is a bound on the size of a maximal flow, obtained by taking the trivial cut $\{s\}$, the algorithm will eventually terminate in time proportional to

$$O\left(\frac{n_1 + \cdots + n_m}{\text{lcm}(n_1, \ldots, n_m)}\right)$$

which is exponential in the bit complexity of the representation. We cannot do any better than this, as the graph in the figure above provides an example. If the edge weights are rational, similar results can be obtained by multiplying out denominators.

We achieve much better estimates if we use breadth first search to find our residual path – that is, we always take the shortest path (in length, not in weight) from $s$ to $t$. But this, of course, requires a careful analysis.

**Theorem 1.6** (Mader)**.** *Let $G$ be an undirected graph, and $s, t$ vertices not directly connected by an edge. Then the maximal number of vertex-disjoint $(s,t)$ paths is equal to the minimum cardinality of a set $B \subset V - \{s,t\}$ which contains a vertex in any path from $s$ to $t$.*

*Proof.* Form a directed bipartite graph $W$, which consists of two copies of the vertices in $G$. If $v$ is a vertex, then we shall separately denote its copies by $v^1$ and $v^2$. If $uv$ is an edge in $G$, attach edges $u^2v^1$ and $v^2u^1$ to $W$, and attach an edge $v^1v^2$ for each vertex $v$. Define a capacity function on $w$ by

$$\mu(u^2v^1) = \mu(v^2u^1) = \infty \qquad \mu(v^1v^2) = 1$$

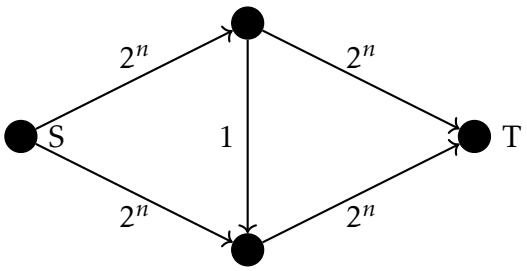


Figure 1.2: If our algorithm continuously switches between adding weights from left to right via the central edge, then we will need to compute $O(2^n)$ residual graphs before termination, even though the graph can be represented in $O(n)$ bits.

Note that any $(s,t)$ path in $G$ leads to an alternating path in $W$ from $s^2$ to $t^1$, which must cross at least one 1-capacity edge because there is no edge from $s$ to $t$.

Let $f$ be an integer-valued flow in $W$. Then $f \leqslant 1$, because an infinite capacity edge $u^2v^1$ satisfies $f(u^2v^1) \leqslant f(\delta_{\text{in}}(u^2)), f(\delta_{\text{out}}(v^1))$ □

## 1.5 Goldberg-Tarjan Push Relabel Algorithm

Our previous algorithm found max flows in a graph by starting with a flow, then slowly augmenting it, improving the flow value until it is maximum. An alternative idea would be to take a flow-like function, which would be a maximum flow if it obeyed flow conservation, and slowly correcting the flow until we have a maximum flow. Define a **preflow** on a graph with capacity function $\mu$ to be a positive real-valued function $f$, defined on the edges, such that $f \leqslant \mu$, and $f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v)) \geqslant 0$ for all $v \neq s$. A preflow defines a residual graph $G_f$ in the same way that a flow does. A non-negative integer valued function $\psi$ on vertices is a **distance label** for a particular preflow $f$ if $\psi(s) = |V|$, $\psi(t) = 0$, and $\psi(v) \leqslant \psi(w)+1$ for $vw \in G_f$. Notice that if there is a path from $s$ to $t$ in $G_f$, then a distance label cannot exist, because if it has length $k$, then the inequality property gives

$$\psi(s) \leqslant (k-1) + \psi(t) \leqslant (|V|-1) + \psi(t) < |V|$$

Thus if we continuously keep track of a preflow and a distance label, and keep adjusting the preflow and distance label to obtain a flow with a distance label, we have obtained a maximal flow. In order to describe a method achieving this, we call a vertex $v \neq s,t$ **active** if $f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v)) > 0$, and an edge $vw \in G_f$ **admissable** if $\psi(v) = \psi(w) + 1$. The goal is to find a preflow with no active vertices, in which case we have a flow.

We now describe the push-relabel algorithm for finding the maximal flow in a graph. Begin by setting

$$f(e) = \begin{cases} \mu(e) & e = sv \\ 0 & \text{otherwise} \end{cases} \qquad \psi(v) = \begin{cases} |V| & v = s \\ 0 & v \neq s \end{cases}$$

Then $\psi$ is a distance label for the preflow $f$, since $G_f$ has no edges leaving $s$.

Suppose there is an active vertex $v$ relative to $f$. If there are no admissable edges leaving $v$, we may introduce an admissable edge by setting $\psi(v) = \min_{vw \in G_f} \psi(w) + 1$. This leaves the $f$ a preflow, and $\psi$ a distance label. If there is an admissable edge $vw$ leaving $v$, we set $f(vw)$ to be

$$\min[\mu_f(vw), f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v))]$$

Then $f \leqslant \mu$ still holds, and $f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v)) \geqslant 0$, since we can only have let at most $f(\delta_{\text{out}}(v)) - f(\delta_{\text{in}}(v))$ flow out. Also $f(\delta_{\text{out}}(w)) - f(\delta_{\text{in}}(w)) \geqslant 0$, since we have only let more flow in. The distance labeling is still a distance labelling, because we introduce an edge $wv$ in $G_f$, which requires that $\psi(w) \leqslant \psi(v) + 1 = \psi(w) + 2$, and an edge from $vw$, so that $\psi(v) = \psi(w) + 1 \leqslant \psi(w) + 1$. Thus the fact that $\psi$ is a distance labelling of the preflow $f$ is an invariant of the algorithm.

**Lemma 1.7.** *If $f$ is a preflow and $\psi$ a distance label for $f$, then if $f(\delta_{out}(v)) - f(\delta_{in}(v)) > 0$, there is a $(v,s)$ path in $G_f$.*

*Proof.* s □

## 1.6 Cuts in Undirected Graphs

Let $G$ be an undirected graph with capacities $\mu \geqslant 0$. For a subset of vertices $U$ let $\delta(U)$ denote all edges with a single endpoint in $U$. A **cut** in

such a graph is $\varnothing \subsetneq U \subsetneq V$, with value $\mu(\delta(U))$. The global minimum cut problem is to find the cut with $\mu(\delta(U))$ minimized, without regard to where any particular pair $(s,t)$ resides. A naive way to calculate the global minimum cut is to calculate a minimum $(s,t)$ cut, for all pairs $(s,t)$. This requires $n^2$ calls to the maximum flow problem.

And while we discuss this problem, we briefly sketch out how to compute a max flow in an undirected graph. Given such a graph $G$ with capacity $\mu$, we define $H$ to be the directed graph with the same vertices as $G$, and for each edge $uv \in G$, two edges $uv$ and $vu$ in $H$, with a capacity function $\psi$ such that $\psi(uv) = \psi(vu) = \mu(uv)$. Given any flow $f$ in the directed graph, we obtain a flow $f'$ in the undirected graph of the same value by subtracting $\min(f(uv), f(vu))$ from each edge. Thus we can compute a max flow in $O(V^2E)$ time in an undirected graph as well. The value of min cuts in the directed graph is the same as the value of min cuts in the undirected graph, so the max flow min cut theorem holds here as well.

More smartly, we notice that if $s$ is fixed, $s$ lies on either side of the cut, and since $\mu(\delta(U)) = \mu(\delta(U^c))$, we need only try each $(s,t)$ cut as $t$ ranges over all $t \neq s$, which only requires $n-1$ calls to the max flow function. In a directed call, we need try all $(s,t)$ cuts and $(t,s)$ cuts, which requires $2(n-1)$ calls to max flow.

We can do much better than just computing a min-cut in $O(n)$ times. It turns out that there is a way to compactly represent all minimum cuts via $O(n)$ calls to a min-cut algorithm. The first step to the representation is to find a useful relationship between the values of the numbers $\lambda_{u,v}$, the capacities of a min $(u,v)$ cut.

**Lemma 1.8.** *Let $v_1, \ldots, v_k$ be vertices. Then $\lambda_{v_1,v_k} \geqslant \min(\lambda_{v_1,v_2}, \ldots, \lambda_{v_{k-1},v_k})$.*

*Proof.* Assume that $k \geqslant 3$, since the proof is trivial for $k = 2$. Let $C$ be a $(v_1, v_k)$ cut with $\mu(\delta(C)) < \min(\lambda_{v_1,v_2}, \ldots, \lambda_{v_{k-1},v_k})$. Inductively, we conclude that $v_2, \ldots, v_k \in C$. But then by definition $C$ cannot be a $(v_1, v_k)$ cut. □

Let $T = (V,E)$ be a tree, and $e \in E$ an edge in the tree. Then the set of nodes in some connected components of the graph $T' = (V, E - \{e\})$ is known as the **fundamental cut**. There are only two connected components, one for each vertex on the ends of $e$. A **Gomory-Hu Tree** for an undirected graph $G$ is a tree $T$ with the same vertices as $G$, such that the fundamental cut for every edge $uv \in T$ is a minimal $uv$ cut. It turns out

that Gomory-Hu trees always exist, can be computed in $O(n)$ calls to a min $(s,t)$ cut function, and compactly represent every min cut in a graph, for if $(v_1,\ldots,v_n)$ is the unique simple path in $T$, then

$$\lambda_{v_1,v_n} = \min(\lambda_{v_1,v_2},\ldots,\lambda_{v_{n-1},v_n})$$

Certainly the left side is $\geqslant$ to the right side. On the other hand, by construction of the tree, the fundamental cut of $(v_i,v_{i+1})$ is also a $(v_1,v_n)$, hence the left side is $\leqslant$ to the right side.

**Theorem 1.9** (Cut Submodularity)**.** *For $A,B \subset V$, we have*

$$\mu(\delta(A)) + \mu(\delta(B)) \geqslant \mu(\delta(A \cap B)) + \mu(\delta(A \cup B))$$

*Proof.* Fix an edge $uv$. We shall show that every edge counted on the right side occurs at least as many times on the left side.

- Suppose $uv \in \delta(A \cap B)$, $uv \notin \delta(A \cup B)$. Then we may assume $u \in A \cap B$, $v \notin A \cap B$, $v \in A \cup B$. Then $\mu(uv)$ is counted once on the right hand side, and once on the left hand side.
- Suppose $uv \notin \delta(A \cap B)$, $uv \in \delta(A \cup B)$. Thus we may assume $u \in A \cup B$, $v \notin A \cup B$, $u \notin A \cap B$. Then $\mu(uv)$ is counted once on the right hand side, and once on the right hand side as well.
- Suppose $uv \in \delta(A \cap B)$, $uv \in \delta(A \cup B)$. Then we may assume $u \in A \cup B$, $v \notin A \cup B$, and so $u \in A \cap B$. Then $\mu(uv)$ is counted once on the right hand side, and twice on the left hand side.

Equality holds when $\delta(A \cap B)$ is disjoint from $\delta(A \cup B)$. □

**Lemma 1.10.** *Let $s \neq t$ be two vertices in a graph $G$, and let $C$ be a min $(s,t)$ cut. If $u \neq v$ are also distinct vertices, then there is a min $(u,v)$ cut $D$ with $C \subset D$ or $D \cap C = \varnothing$.*

*Proof.* Note that the statement is really redundant, because $C \subset D$ occurs exactly when $D^c \cap C = \varnothing$. Let $B$ be any minimum $(u,v)$ cut. We will modify $B$ to another min cut with the desired properties. Note that

$$\mu(\delta(C)) + \mu(\delta(B)) \geqslant \mu(\delta(C \cap B)) + \mu(\delta(C \cup B))$$

We may assume $s,u \in B$. Then $t \notin B$, $C \cap B$ is an $(s,t)$ cut, hence $\mu(\delta(C \cap B)) \geqslant \mu(\delta(C))$, and so

$$\mu(\delta(B)) \geqslant \mu(\delta(C \cup B))$$

Now $u \in C \cup B$, and $v \notin C \cup B$ (WHY?), so $D = C \cup B$ satisfies the properties of the lemma. □

## 1.7 Potentials

Fix a directed graph $G$ with edge costs $c$. A **potential** for $G$ is a mapping $\phi : V \to \mathbf{R}$ such that

$$c_\phi(uv) = c(uv) - (\phi(v) - \phi(u)) \geqslant 0$$

for al edges $uv$ in $G$. The costs $c_\phi$ are known as the potential costs.

**Lemma 1.11.** *For any $s - t$ path $P$ in $G$,*

$$c_\phi(P) = c(P) - (\phi(t) - \phi(s))$$

*Proof.* If $P = (v_1, \ldots, v_n)$, then we have an alternating sum

$$c_\phi(P) = \sum_{k=1}^{n-1} c(v_k v_{k+1}) + \phi(v_k) - \phi(v_{k+1}) = \sum_{k=1}^{n-1} c(v_k v_{k+1}) + \phi(v_1) - \phi(v_n)$$

and the right side is exactly the form considered in the statement of the theorem. □

This is a discrete version of the fact that the energy change of a particle under a conservative force is invariant of the path of the particle, but only the change of the particles position in space. A simple corollary to the lemma is that the minimal cost paths relative to the metric $c$ are the same as the minimal cost paths relative to $c_\phi$.

If a graph has negative edge weights, but possesses some potential $\phi$, then we can compute minimal paths under the cost function $c_\phi$ (which is now has positive edge weights everywhere) using Dijkstra's algorithm. There is a simple method to finding a potential, which we now detail.

If $P$ is a cycle in a graph $G$ with potential $\phi$, we calculate

$$c_\phi(P) = c(P) \geqslant 0$$

so that $G$ has no negative weight cycles. Conversely, if $G$ has no negative weight cycles, then it turns out that $G$ has a potential, and this potential can be computed in polyomial time.

**Theorem 1.12.** *A potential can be computed for a graph $G$ can be computed in $O(nm)$ time.*

*Proof.* Given $G$, compute a graph $H$ by adding a new vertex $r$ to $G$, and adding edges from $r$ to every vertex in $G$. Set $c(rv) = 0$ for all vertices $v \in V$, and let $\phi(v)$ be the cost of the cheapest walk from $r$ to $v$. This is well defined since $G$ has no negative weight cycles (there is a cheapest walk with no repeated vertices, for if a path has a repeated vertex, it contains a cycle, which must have non-negative weight, and this cycle can be removed with no penalty to the length of the walk – we then need only consider the finite set of paths with no repeated vertices to obtain an answer). Note that $c(uv) + \phi(u) \geqslant \phi(v)$, so $\phi$ is a potential for $G$. The Bellman-Ford can compute $\phi$ in $O(nm)$ time. □

Let $\omega$ be a differential 1-form on a manifold $M$ such that

$$\int_\gamma \omega \geqslant 0$$

for any cycle $\gamma$. Then if $\gamma$ and $\delta$ are two paths from a point $x$ to a point $y$, we find

$$\int_\gamma \omega - \int_\delta \omega, \int_\delta \omega - \int_\gamma \omega \geqslant 0$$

so $\int_\gamma \omega = \int_\delta \omega$, and given a fixed $x$, we can define

$$F(y) = \int_\gamma \omega$$

where $\gamma$ is a path from $x$ to $y$, and one verifies that $dF = \omega$, so that

$$\int_\gamma \omega - (F(y) - F(x)) = 0$$

a stronger fact which is analogous to the result above, although our result is weaker because in directed graphs we cannot perform the swapping trick to obtain equality of all paths.

## 1.8 Minimum Cost Flows

Throughout this section, we fix a directed graph with edge costs $c$, edge capacities $\mu$, fixed vertices $s, t$, and a target flow value $\gamma$. The minimum cost

flow problem tries to find a flow $f$ with value $\text{val}(f) \geqslant \gamma$, which minimizes the cost $\text{cost}(f) = \sum c(e)f(e)$, or to determine if no such flow exists.

We can certainly compute in polynomial time if a flow exists satisfying the constraints, because we can find a maximal flow in the graph. It is not so clear that we can find such a flow with minimal cost. It shall always be the case that the minimal flow satisfies $\text{val}(f) = \gamma$, because given any flow $f$, the value of the flow $g = \text{val}(f)/\gamma f$ satisfies $\text{val}(g) = \gamma$, and $\text{cost}(g) = \text{val}(f)/\gamma \sum c(e)f(e)$, which will be smaller than $\text{cost}(f)$ if $\text{val}(f) \neq \gamma$.

## 1.9 Minimal Cost Bipartite Matching

We now take a brief aside to discuss how minimal cost flows can be used to solve the analogous problem in bipartite graphs. Given the name, the minimal cost bipartite matching problem should be fairly self explanatory. We take a particular Bipartite graph $G = (L \cup R, E)$ with edge costs $c$, and we must find a minimum cost perfect matching on $G$ in polynomial time. That is, our goal is a perfect matching $M$ which minimizes

$$\text{cost}(M) = \sum_{e \in M} c(e)$$

The similarity to the minimal cost flow problem immediately suggests a connection between the two problems.

We can formulate the perfect matching problem as a minimal cost max flow problem, by modifying slightly the standard reformulation of the maximal matching problem as a max flow problem. The original process is to construct a directed bipartite graph $H$, whose vertices are obtained from $G$ by adding two new vertices $s$ and $t$, orienting all edges in $E$ so that they point from $L$ to $R$, and adding new edges $sv$, for $v \in L$, and $wt$, for $w \in R$. If we define a capacity function $\mu = 1$, then there is a one-to-one correspondence between integral valued $s - t$ flows and matchings in the graph. To see this, note that an integral valued flow $f$ gives rise to a characteristic function $\chi_{A_f}$ on the edges of $G$, and the flow conservation law shows that if $e$ and $e'$ are adjoint, then $\chi_{A_f}(e) + \chi_{A_f}(e') \leqslant 1$, so that $A_f$ forms a matching on $G$. Correspondingly, the reverse process takes a matching to a characteristic function on the edges of $G$, which can easily be completed into a flow on $H$.

Now we define a cost function $c'$ on $H$ in a way which mirrors the cost function $c$ on $G$. We let

$$c'(sv) = c'(wt) = 0 \qquad c'(vw) = c(vw)$$

Then we see that the correspondence described above preserves the cost function. That is, if a matching $M$ corresponds to a flow $f$, then $\text{cost}(M) = \text{cost}(f)$, because $f$ is obtained from $M$ by adding edges of the form $sv$ and $wt$, which add no value to the cost of the flow. Thus if we find a minimal cost maximal flow in $H$, we can find a minimal cost perfect matching in $G$, and we know we can find such a maximal flow using the successive shortest paths algorithm, and we need only compute $|L| = |R|$ successive paths, since each iteration of the algorith improves the length of the candidate path, hence now have an algorithm for the minimal cost perfect matching problem which runs in $O(n(m + n\log n))$ time.

## 1.10 Minimum Mean Cycle Cancelling

We already know that a flow $f$ has minimal cost if $G_f$ contains no negative-cost cycles. This suggests that a strategy for finding minimal cost flows is obtained from successively pruning negative-cost cycles from $G_f$ until there are none left. As with Ford Fulkerson, we must be careful which cycles we prune in order to guarantee a good asymptotic speed to our algorithm. It turns out that fast asymptotic results are available if we prune negative cycles with 'minimum mean'.

**Algorithm 1** The Minimum Mean Cycle Canceling Algorithm

1: $f \leftarrow$ any $(s,t)$ flow of value $\gamma$ (found in $O(n^3\sqrt{m})$ time)
2: **while** $G_f$ contains negative weight cycles **do**
3: $\quad C \leftarrow$ a cycle in $G$ which minimizes its mean cost $c(C)/|C|$.
4: $\quad$ Augment $f$ along $C$.
5: **return** $f$

The algorithm certainly terminates, since the cost of $f$ decreases for every iteration of the algorithm, at a rate bounded by the rational values of the edge costs. It remains to be seen that the algorithm terminates in a

polynomial amount of time (and this is not immediately obvious, and is not true if we do not take cycles which minimize the mean cost).

First, we argue why we can find a minimum ratio cycle $C$ in polynomial time.

**Lemma 1.13.** *Any Eulerian multigraph $H = (W, F)$ with edge costs $c$ contains a cycle $C$ such that*

$$\frac{c(C)}{|C|} \leqslant \frac{c(F)}{|F|}$$

*Proof.* Perform a cycle decomposition on $H$, writing $F = C_1 \amalg \cdots \amalg C_n$, we find

$$\frac{c(F)}{|F|} = \frac{c(C_1) + \cdots + c(C_n)}{|C_1| + \cdots + |C_n|}$$

If $c(C_i)/|C_i| > c(F)/|F|$ for each $i$, then $c(C_i)|F| > c(F)|C_i|$ also holds for each $i$, and so, summing up we find

$$c(F)|F| = \sum_i c(C_i)|F| > \sum_i c(F)|C_i| = |F|c(F)$$

a clear contradiction which proves the existence of a $C_i$ satisfying the constraints of the lemma. □

Given any cycle $C$, the corresponding multigraph (which contains duplicates of repeated edges in the cycle) is Eulerian, and can thus be broken up into disjoint cycles. The lemma above shows that if $C$ is not a simple cycle, we can always find a strictly smaller cycle $C'$ without increasing the mean value of the cycle, so that we need only search simple cycles to find a minimum mean.

For $k \leqslant n$, define $\delta_k(v)$ to be the cheapest closed walk from $v$ to itself using exactly $k$ (not necessarily distinct) edges. If we let $\delta_k(v, w)$ be the cheapest path from $v$ to $w$ using $k$ edges, then we find

$$\delta_k(v, w) = \min_{u \in V} \; c(v, u) + \delta_{k-1}(u, w)$$

Each value of $\delta_k(v, w)$ can be computed in $O(m)$ time, and so the set of all $\delta_k(v, w)$ can be computed in $O(mn^3)$ time. Let $v^*$ and $k^*$ minimize $\delta_{k^*}(v^*)/k^*$, and pick $k^*$ to be smallest to break ties. Then the walk from $v^*$ in $k^*$ stops is verified to be a cycle, from the above remark, and thus is a minimum ratio cycle, since it is a minimum ratio walk in general.

**Theorem 1.14.** *The number of iterations of the minimum mean cycle cancelling algorithm is $O(m^2 n \log n)$.*

*Proof.* Consider the sequence of flows

$$f_1, f_2, \ldots, f_N$$

which are found in the MMCC algorithm, together with the cycles

$$C_1, \ldots, C_{N-1}$$

which are augmented to produce the sequence of flows. For $i < j$, define

$$\alpha(i,j) = \{e, \overleftarrow{e} : e, \overleftarrow{e} \in C_i \cup C_j\}$$

**Lemma 1.15.** *If $\alpha(i,k) = \varnothing$ for $i < k < j$, then we find*

$$\frac{c(C_i)}{|C_i|} \leqslant \frac{n}{n - |\alpha(i,j)|} \frac{c(C_j)}{|C_j|}$$

*Proof.* Let $H = C_i \coprod C_j - \alpha(i,j)$ (essentially, $H$ is obtained from the cycles $C_i$ and $C_j$ by keeping duplicate edges, and 'cancelling out' edges with their reverse). Then $H$ is Eulerian, for $C_i \coprod C_j$ is Eulerian, and removing edges along with their reverse changes both the in degree and out degree of a vertex by the same amount. Furthermore, each edge in $H$ is in $E_{f_i}$ because $\alpha(i,k) = \varnothing$, so

$$\begin{aligned}
\frac{c(C_i)}{|C_i|}(|C_i| + |C_j|) &= \frac{c(C_i)}{|C_i|}(|H| + |\alpha(i,j)|) \\
&\leqslant c(H) + |\alpha(i,j)|\frac{c(C_i)}{|C_i|} \\
&= c(C_i) + c(C_j) + |\alpha(i,j)|\frac{c(C_i)}{|C_i|}
\end{aligned}$$

where we have used the fact that $c(H) = c(C_i) + c(C_j)$ since the cost of reverse edges cancels out. This can be rephrased as

$$\frac{c(C_i)}{|C_i|}\left(|C_j| - |\alpha(i,j)|\right) \leqslant \frac{c(C_j)}{|C_j|}|C_j|$$

And dividing out, we find

$$\frac{c(C_i)}{|C_i|} \leqslant \frac{|C_j|}{|C_j| - |\alpha(i,j)|} \frac{c(C_j)}{|C_j|}$$

and this inequality can be weakened to the form above, because $x/(x-y)$ is a decreasing function of $x$ for $y > 0$, and $n/(n - |\alpha(i,j)|) \geqslant 0$. □

Returning to our original proof, we see that $j = i + 1$ satisfies the theorem, and proves that the minimum mean ratios are non-decreasing. If $i < j$ is the smallest value such that $\alpha(k,i) \neq \varnothing$, for some $i \leqslant k < j$, then we find

$$\frac{c(C_i)}{|C_i|} \leqslant \frac{n}{n-2} \frac{c(C_j)}{|C_j|} = \left(1 - \frac{n}{2}\right) \frac{c(C_j)}{|C_j|}$$

and we have a bound $j \leqslant i + m$. To see this, note that at least one edge is removed in the residual graph from each of the cycles $C_i, C_{i+1}, \ldots, C_{i+m}$ during the algorithm. Thus we conclude

$$\frac{c(C_i)}{|C_i|} \leqslant \left(1 - \frac{n}{2}\right)^n \frac{c(C_{i+nm})}{|C_{i+nm}|} \leqslant e^{-1/2} \frac{c(C_{i+nm})}{|C_{i+nm}|} \leqslant 2 \frac{c(C_{i+nm})}{|C_{i+nm}|}$$

If the edge weights are integral (and all rational cost problems can be reduced to this form), this already gives us enough information to prove a polynomial time bound. Note that since the minimum mean cycles are simple, we must have

$$\frac{|c(C_i)|}{|C_i|} \geqslant \frac{1}{n}$$

and

$$\frac{|c(C_1)|}{|C_1|} \leqslant \min_{e \in E} |c(e)|$$

so if $N$ is large enough that $2^{-N} \min_{e \in E} |c(e)| \leqslant n^{-1}$, then

$$\frac{c(C_{i+Nnm})}{|C_{i+Nnm}|} \geqslant \frac{1}{2^N} \frac{c(C_1)}{|C_1|} \geqslant -\frac{\min_{e \in E} |c(e)|}{2^N} \geqslant -\frac{1}{n}$$

which forces the mean cycle ratio to be positive, and hence the algorithm will terminate. We can choose any $\lg(n \min_{e \in E} |c(E)|) < N$, so the algorithm terminates in at most

$$nm\lceil \lg n \min_{e \in E} |c(E)| \rceil$$

iterations, which is polynomial in the bit complexity of the algorithm, since if the costs are given with $l$ bits, then the total bit complexity is about $\Theta(n+ml)$, and $|c(E)| \leqslant 2^l$, so

$$nm\lceil \lg n \min_{e\in E} |c(E)| \rceil \leqslant nm \lg(n) + lnm$$

This finishes our initial analysis of the mean cycle canceling algorithm. □

The polynomial bound for this algorithm is absolutely awful, and we can do much much better. In the next lecture we will begin the arguments that give us a much better bound on the runtime on this algorithm.

## 1.11 Minimum Spanning Trees and Arboresences

Let $G$ be an undirected graph, together with a (possibly-negative) edge cost function $c$. A **spanning tree** $T$ on $G$ is a connected subset of edges containing no cycles (a tree) such that every vertex is covered by an edge in the tree. Note that this definition is equivalent to $T$ having no cycles, and $n-1$ edges (a fact proven most easily by induction, by removing vertexes and edges from $T$ which form the leaves of $T$). There is another equivalent definition which is more important to the spanning tree problems we'll discuss. $T$ is a spanning tree if it has $n-1$ edges, and $|E(S)| \leqslant |S|-1$ for all vertex sets $S$ (where $E(S) = \{uv : u,v \in S, uv \in T\}$).

$$TODO: Draw a picture of S bounding the number of edges in the set S$$

The minimum spanning tree problem is self explanatory – given $G$, find a spanning tree containing the fewest edges. We can formulate the problem as an $LP$ in exponentially many constraints in functions $x : E \to \mathbf{R}$,

$$\begin{aligned} \min \quad & \sum_e c(e)x(e) \\ \text{s.t.} \quad & x(E(S)) \geqslant |S|-1 \quad \forall S \subsetneq V \\ & x(E(V)) = n-1 \\ & 0 \leqslant x \end{aligned}$$

Note that the constraint $x \leqslant 1$ is already encoded in the problem by taking $S$ to be a pair of vertices corresponding to an edge. Our discussion above entails that any integral solution to this algorithm corresponds to a spanning tree in $G$. It can be shown that all extreme points are integral, but we shall only show that we can choose an optimal extreme point which is integral.

In it's current form, the minimal spanning tree linear program is unfeasible, having far too many constraints. We obtain a much better optimization problem if we switch to the dual of this program. This can be motivated by the fact that $x$ behaves more like a linear functional than a vector in this linear program (the constraints of the program are expressed as $x$'s operations on the vector space $\mathbf{R} \cdot E$ generated by the edges $E$), so that the program is the dual of some other, more easily understood linear program. The dual program finds functions $y : 2^V - \{\varnothing\} \to \mathbf{R}$ which satisfy

$$\begin{aligned} \max \quad & -\sum (1 - |S|) y(S) \\ \text{s.t.} \quad & \sum_{e \in E(S)} y(S) \geqslant -c(e) \quad \forall e \in E(S) \\ & y(S) \geqslant 0 \qquad\qquad \forall S \neq V \end{aligned}$$

Kruskal's algorithm, a method for finding the minimal spanning tree, can be viewed as a combinatorial method to solving the dual LP of the spanning tree problem. We first recall the simple, greedy method to form a spanning tree. We can verify this algorithm's correctness using the dual-

**Algorithm 2** Kruskal's Algorithm

1: Set $T = \varnothing$, $K = E$.
2: **while** $T$ is not a spanning tree **do**
3: Remove $e \in K$ with minimal weight.
4: Append $e$ to $T$ if it connects two connected components of $T$.
5: **return** $T$

ity of linear programming. Suppose that $T$ contains the edges $e_1, \ldots, e_n$, which are placed in the order they were added to $T$. Let $S_k$ be the component containing $e_k$ in the graph consisting only of the edges $e_1, \ldots, e_k$ (it is the component that was freshly merged together in the $k$'th iteration of the algorithm). Let $x : E \to \{0, 1\}$ be the indicator function of $T$, and

$y : 2^V - \{\varnothing\} \to \mathbf{R}$ be defined by letting $y(S_i) = c(e_j) - c(e_i)$, where $j$ is the smallest index greater than $i$ such that one of the endpoints of $e_j$ contains points in $X_i$, and define $y(S_n) = y(V) = -c(e_n)$. Define $y(S) = 0$ otherwise. For any edge $e$, by the telescoping sum property of our definition we have

$$\sum_{e \subset S} y(S) = -c(e_i)$$

where $i$ is the smallest index such that $X_i$ contains both endpoints of $e$. The way we selected edges guarantees that $c(e_i) \leqslant c(e)$, so that our constraints are satisfied, and our tight for $e_1, \ldots, e_n$.

We now verify complementary slackness, so that $x$ is verified optimal. If $x(e) > 0$, then our calculation above shows

$$-\sum_{e \subset S} y(S) = c(e)$$

If $y(S) > 0$, then $S = S_i$ for some $i$, hence $S_i$ is a connected tree, and contains $n-1$ vertices, so $x(E(S)) = |S| - 1$. Complementary slackness guarantees that $x$ and $y$ are optimal solutions to their corresponding algorithms, verifying correctness of Kruskal's algorithm.

## 1.12 Arboresences

An (out) **arboresence** in a directed graph with root $s$ is a subset $T$ of $n-1$ edges, such that there is a unique directed path from $s$ to any other vertex. Given a cost function $c \geqslant 0$ and vertex $s$, we want to try and find the min-cost arboresence at $s$. Like the minimal cost spanning tree problem, we can express this problem as an LP,

$$\begin{aligned} \min \quad & \sum_e c(e)x(e) \\ \text{s.t.} \quad & x(\delta^{\text{in}}(U)) \geqslant 1 \quad \forall U \subset V - \{s\} \\ & x \geqslant 0 \end{aligned}$$

If $T$ is an arbitrary arboresence at $v$, then the corresponding characteristic function certainly is a feasible solution to the problem. The problem with this LP is that there are solutions (possibly even optimal ones) which don't look like arboresences. However, we shall not use the LP to solve

linear programs, but instead use properties of linear programs to guarantee optimality. We can form a simple argument to show the existence of optimal solutions which are arboresence. First, note that we only every have upper bounds to the algorithm, and there is no need to have $x(e) > 1$ to satisfy any of the constraints, due to the positivity of $x$, so we may always assume that an optimal integral solution takes value in $\{0, 1\}$, and thus correspond to characteristic functions of edges. Second, we see that the subgraph formed contains paths from $v$ to any other vertex. Since $x(\delta^{\text{in}}(V - s) \geqslant 1$, there is an edge connecting $s$ to some over vertex $v_1$. Then $x(\delta^{\text{in}}(V - \{s, v_1\}) \geqslant 1$, so there is some edge from $s$ to $v_1$, or $v_1$ to $v_2$. In either case, there is an edge to $v_2$. Continuing this process gives you a path to any vertex in the graph. We may now assume our subgraph is a tree, because we can always remove edges to obtain an algorithm that is at least as optimal. Thus we have argued that there is an optimal solution which is an arboresence, like we are looking for.

Let us form the dual LP, so we can use complementary slackness to verify optimality.

$$\begin{aligned} \max \quad & \sum_S y(S) \\ \text{s.t.} \quad & \sum_{e \in \delta^{\text{in}}(S)} y(S) \leqslant c(e) \\ & y \geqslant 0 \end{aligned}$$

We now describe the algorithm which gives us minimal cost arboresences. Note that $y$ is only included here for purposes of algorithmic correctness, and needn't be coded up in real algorithms.

* DRAW A PICTURE OF CONTRACTING CYCLES

**Algorithm 3** Edmond's Algorithm

$y_{\{v\}} = \min c(uv)$, for $v \neq r$.
Set $F = \{e_w : w \in V - \{r\}\}$, where $e_w$ is the cheapest edge entering $w \neq r$.
**while** $F$ has a cycle **do**
    Let $G'$ be the graph obtained by contracting each cycle in $(G, F)$.
    Define $c'(uv) = c(uv) - y(\{v\})$ if $v$ is on a cycle, else $c'(uv) = c(uv)$.
    Let $(T', y')$ be a min-cost arboresence in $G'$.
    Define $T = (F \cup T') - \{uv : u \text{ on a cycle} wv \in T'\}$.
    Set $y(S) = 0$ if $S$ cuts a cycle, and $y'(S)$ if $S = [S]$ after contraction.
**return** $F$

# Chapter 2

# Matroids

If you have a problem, and a matroid is involved, you can probably use a greedy algorithm to solve the problem. This is very useful, because greedy solutions are normally far faster than conservative methods. A matroid is a set of sets which can be 'greedily expanded' without losing the properties defining a matroid. Formally, a matroid is a collection of subsets of some set $X$ containing the emptyset, closed under the subset relation, and such that if two sets $A$ and $B$ are in the matroid with $|A| < |B|$, there is $z \in B - A$ such that $A \cup \{z\}$ is in the matroid.

**Example.** *If $G$ is a graph, the set of edge subsets which contain no cycle forms a matroid. Certainly it contains the emptyset, and certainly a subset of edges with no cycle contains no cycle.*

**Example.** *If $V$ is a vector space, the collection of linearly independant vector sets forms a matroid. Essentially, you can greedily expand vectors to form a basis.*

**Example.** *If $X$ is a finite set, and $L$ is a laminar family over $X$, together with a function $f \geqslant 0$ defined on the Laminar family, then the same of subsets $A$ such that $|A \cap B| \leqslant f(B)$ for each $B \in L$, forms a matroid known as a laminar family.*

**Example.** *If $G$ is a directed graph, and $A, B \subset V$ are given, the set of subsets of $A$ which contains vertex disjoint paths into $B$ is a matroid, known as a Gammoid.*

**Example.** *Given a connected graph $G$, the set of bond matroid.*

**Example.** *Given a field extension $F \subset K$, the set of finite subsets $Y$ of $F$ with trancendence degree $|Y|$ forms a matroid, known as the algebraic matroid.*

An independence oracle is an algorithm to decide whether an arbitrary subset is an element of a given matroid in polynomial time. This is key to having efficient algorithms for solving matroids.

A base of a matroid is a maximal set – an element $B$ such that if $B \subset B'$, and $B'$ is in the matroid, then $B = B'$. If our matroid consists of finite sets, then two bases have the same cardinality. Indeed, if $|B| < |B'|$, there is $z \in B' - B$ such that $B + z$ is in the matroid, which cannot occur since this is an expansion of $B$.

The first algorithm we consider is to efficiently find a min weight base for a matroid over a set $X$, for some function $c : X \to \mathbf{R}$. This is a generalization of the minimum spanning tree problem. The algorithm to do this is to just greedily add the minimum element of $X$ to a set until we cannot anymore. The set $B$ we end up with is a base by construction, because the order we add elements is immaterial. To prove that $B$ is minimal, consider a minimal base $B^*$, and order the elements of both sets

$$B = \{x_1, \ldots, x_n\} \qquad B^* = \{x_1^*, \ldots, x_n^*\}$$

We prove by induction that $w(x_i) \leqslant w(x_i^*)$. Indeed, consider the partial set $B_{i-1}\{x_1, \ldots, x_{i-1}\}$ and $B_i^*\{x_1^*, \ldots, x_i^*\}$. There is a minimal weight element $x_k^*$ of $B_i^*$ which is not in $B_{i-1}$, and can be safely added to $B$, and $w(x_i) \leqslant w(x_k^*)$. If $k \leqslant i$, this shows that $w(x_i) \leqslant w(x_i^*)$. (FIGURE THIS PROOF OUT LATER)

The **rank** $r(A)$ of a subset $A$ of $X$, with respect to a matroid $M$ is the cardinality of the largest subset of $A$ in $M$. Then $r$ is order preserving, and $r(A) \leqslant |A|$. More interestingly,

$$r(A) + r(B) \geqslant r(A \cup B) + r(A \cap B)$$

To prove this, let $J$ be a maximal subset of $A \cap B$ in the matroid, $K \subset A - B$ a complementary set for $J$, so that $|K| + |J| = r(A)$. This is always possible because we can extend $J$ to a base in $A$. Do the same for $J$ in $B$, forming $L$ such that $|K| + |J| + |L| = r(A \cup B)$. By construction, $J \cup K$ is independant in $A$, and $L$ is independent in $B$. But then by maximality we obtain the bound above. It turns out that the rank function of a matroid uniquely determines the matroid, but we won't discuss that.

A subset $C$ of $X$ is a **circuit** if it is not independent, but removing any particular element of $C$ makes it independent. If $Y$ is any independent set, and $Y \cup \{x\}$ is independent, then there is a unique circuit contained in $Y \cup \{x\}$. We call this circuit $C(Y,x)$. To find this circuit, consider a minimal element of $Y \cup \{x\}$ which is not independent. The hard part is showing this is the only circuit. Let $C_1$ and $C_2$ be two circuits. If $C_1 \neq C_2$, let $x \in C_1 \cap C_2$, and $y \in C_1 - C_2$ (without loss of generality). We will show there is a circuit contained in $(C_1 \cup C_2) - \{x\}$ contains a circuit. Indeed, using submodularity we find

$$\begin{aligned} |C_1| - 1 + r((C_1 \cup C_2) - \{x,y\}) + |C_2| - 1 &= r(C_1) + r((C_1 \cup C_2) - \{x,y\}) + r(C_2) \\ &\geqslant r(C_1) + r((C_1 \cup C_2) - \{y\}) + r(C_2 - \{x\}) \\ &\geqslant r(C_1 - \{y\}) + r(C_1 \cup C_2) + r(C_2 - \{y\}) \\ &= |C_1| - 1 + r(C_1 \cup C_2) + |C_2| - 1 \end{aligned}$$

The point of this calculation is that now we know $r(C_1 \cup C_2) = r((C_1 \cup C_2) - \{x,y\})$, so that we cannot add $x$ or $y$ to any maximal independent set in $(C_1 \cup C_2) - \{x,y\}$ while keeping it independant. In particular, $(C_1 \cup C_2) - \{x\}$ is not independent, so there must be a circuit contained in the set.

That there is always a unique circuit 'screwing us over' is very useful in analyzing the matroid intersection problem. Given two matroids $M$ and $N$ over a set $X$, the intersection problem aks us to find the maximal set which is contained in both matroids. Similarily, we may have a subset $Y$ independent in both $M$ and $N$, and we wish to know whether we can extend $Y$ while preserving independence. Consider the following algorithm: given $Y$, we construct a directed graph with nodes in $Y$ and $X - Y$, and directed edges $ab$ (for $a \in Y$, $b \in X - Y$) if $Y - a + b \in M$, and $ba$ if $Y - a + b \in N$. We find a shortest path from $Y_1$ to $Y_2$ where

$$Y_1 = \{x \in X - Y : Y + x \in M\} \qquad Y_2 = \{x \in X - Y : Y + x \in N\}$$

and we then augment along that path to find a better matroid. If there is no path from $Y_1$ to $Y_2$, we have a maximal matroid.

## 2.1 Linear Programs for Matroids

Let $M$ be a matroid over a set $X$, together with a weight function $w : M \to \mathbf{R}$. We can formulate the maximum weight independent set problem over

$X$ as a linear program. Specifically, we consider the space of functions in $\mathbf{R}^X$, and take the linear program

$$\begin{aligned} \max \quad & \sum_{e \in X} w(e)x(e) \\ \text{s.t.} \quad & x(A) \leqslant r(A) \quad \forall A \subset X \\ & x \geqslant 0 \end{aligned}$$

in particular, the constraints guarantee that $0 \leqslant x(e) \leqslant 1$ for all $e \in X$, so integral points correspond to indicator functions over $X$. What's more, integral points also correspond to independent sets in the matroid. If $x \in \chi_B$, then $B = x(B) \leqslant r(B) \leqslant B$, so $r(B) = B$, and so $B$ is independant.

It turns out that (surprise surprise!) all extreme points are integral, so we can solve the problem in polynomial time. Our proof will use an interesting technique known as the uncrossing method, which 'thins' the constraints of the LP so they correspond to a Laminar family (If we say $A$ and $B$ are crossed if $A \cap B \neq \varnothing$, then by breaking them into disjoint chains we have 'uncrossed them'). First, we embed $2^X$ in $\mathbf{R}$ by mapping $A \subset X$ to the $0/1$ indicator function $\chi_A$. Then we let $\mathcal{C}$ to be the largest linearly ordered subset of $2^X$ (a *chain* in $2^X$) only containing subsets $A$ for which we have a tight constraint, $x(A) = r(A)$, and such that the $\chi_A$ are independent. We claim that the $\chi_A$ span $\mathbf{R}^X$.

Next, note that the functionals $x \mapsto x(A)$ are essentially the duals of the characteristic function $\chi_A$, and since $2^X$ spans $\mathbf{R}^X$, the rows of the constraint matrix which $x$ satifies span $\mathbf{R}^X$. In particular, since $x$ is extreme, the set of tight constraints spans $\mathbf{R}^X$, and it suffices for us to prove that $\operatorname{span}(\mathcal{C})$ contains all $\chi_A$ such that $x(A) = r(A)$. Suppose that $\chi_B$ was not in $\operatorname{span}(\mathcal{C})$, where $r(B) = x(B)$. If such a $B$ exists, we may choose $B$ such that the number of elements is minimized

$$\tau(B) = \{A \in \mathcal{C} : A \not\subseteq B \text{ and } B \not\subseteq A\}$$

is minimized. Note that we must have $\tau(B)$ must contain at least one element, or else $\mathcal{C} \cup \{B\}$ would be a chain of linearly independent vectors which is larger than $\mathcal{C}$. So let $T \in \mathcal{C}$ be such that $B \notin T$ and $T \notin B$. Note that by using the inclusion/exclusion principle, optimality, and the submodu-

larity property,

$$\begin{aligned} r(B) + r(T) &= x(B) + x(T) \\ &= x(B \cup T) + x(B \cap T) \\ &\leqslant r(B \cup T) + r(B \cap T) \\ &\leqslant r(B) + r(T) \end{aligned}$$

So that $x(B \cup T) + x(B \cap T) = r(B \cup T) + r(B \cap T)$, and since $x(B \cup T) \leqslant r(B \cup T)$, $x(B \cap T) \leqslant r(B \cap T)$, we can actually split this inequality into the equalities $x(B \cup T) = r(B \cup T)$, $x(B \cap T) = r(B \cap T)$.

It turns out that $\tau(B \cup T), \tau(B \cap T) \subsetneq \tau(B)$, which by optimality implies $\chi_{B \cup T}, \chi_{B \cap T} \in \operatorname{span}(\mathcal{C})$. This is clearly in contradiction to our assumption, since then

$$\chi_B = \chi_{B \cup T} + \chi_{B \cap T} - \chi_T \in \operatorname{span}(\mathcal{C})$$

so all that remains is to prove properties of the $\tau$ function. This is most easily shown by drawing out Venn diagrams, which is often useful in matroid theory, but we'll also provide a textual description (and drawing your own diagram will probably be much more revealing than looking at someone elses).

Consider $S$ in $\mathcal{C}$. Then either $S \subset T$, or $T \subset S$. If $S \subset T$, then $S \subset B \cup T$ and so $S \notin \tau(B \cup T)$. If $T \subset S$, then $B \cap T \subset S$, so $S \notin \tau(B \cap T)$. If $S \in \tau(B \cup T)$ but $S \notin \tau(B)$, then we would have to have $T \subset S$. This also shows $S \subset B$, because if $B \subset S$, then $B \cup T \subset S$. But then $T \subset B$, which is a contradiction. Similarily, if $S \in \tau(B \cap T)$ but $S \notin \tau(B)$, then $S \subset T$ and $B \subset S$, so $B \subset T$, another contradiction. We finish this conclusion by noticing that $T \notin \tau(B \cap T), \tau(B \cup T)$ by obvious relations.

Thus we conclude that $\mathcal{C}$ forms a basis of $\mathbf{R}^X$ (in fact, $\mathcal{C}$ forms a basis for $\mathbf{Z}^X$ – the same proof follows through since we've never used real numbers, but we won't need this in our proof. Now our homework comes in handy. Consider the laminar constrainted matching problem, over the Bipartite graph whose left vertices are subsets of $X$, and with only one right vertex, with an edge between every left vertex. We consider $L_L = \mathcal{C}$ as a laminar family over the left vertices, and $L_R = \varnothing$, with $b_A = r(A)$ for $A \in \mathcal{C}$. Every solution to this problem can be identified with a characteristic function over the vertices, and by the identification $x$ can be seen as a feasible solution to the Laminar constraints. Since $x$ is tight for *all* constraints, it must be a feasible solution to this problem, and in our homework we proved that therefore $x$ has integral coordinates.

The matroid intersection problem can be formulated as an LP with integral extreme points in a very similar manner. We consider two matroids $M$ and $N$, take the same solution space as the maximal independent set LP, and find solutions such that

$$\begin{aligned} \max \quad & \sum_{e \in X} w(e)x(e) \\ \text{s.t.} \quad & x(A) \leqslant \min(r_M(A), r_N(A)) \quad \forall A \subset X \\ & x \geqslant 0 \end{aligned}$$

The extreme points of this set are integral, which can be verified from the uncrossing technique of the last problem. Here we will end up with a laminar family of tight constraints $\mathcal{C}_M$ and $\mathcal{C}_N$ over each matroid, rather than a chain,prove that $\text{span}(\mathcal{C}_M \cup \mathcal{C}_N)$ is full, and then use the laminar family $L_L = \mathcal{C}_M$, $L_R = \mathcal{C}_N$ to prove integrality. Our homework problem does not generalize to intersections of three or more matroids, and for good reason! The matroid intersection problem for more than 2 matroids is NP complete.

# Chapter 3

# Submodular Functions

A **submodular function** is a function $f : 2^{\Omega} \to \mathbf{R}$, such that

$$f(S) + f(T) \geqslant f(S \cup T) + f(S \cap T)$$

One algorithm that occurs across computing science is to find a maximum set for a submodular function. Consider the greedy approximation which constructs a set by taking the point which increases the current set by the largest amount. If $Z$ is the set constructed, and $Z^*$ is an optimum set, we claim that $f(Z) \geqslant (1 - 1/e) f(Z^*)$ – this is the best approximation one can do in polynomial time, unless $P = NP$.

**Lemma 3.1.**

$$f(Z^*) \leqslant k(f(z) - f(z))$$

*Proof.*

$$f(Z^*) \leqslant f(Z^* \cup Z_{i-1})$$

□

# Chapter 4

# Perfect Graphs

There are many numbers of interest associated to graphs. For instance, we have $\alpha(G)$, the size of the maximum independent set, $\omega(G)$, the maximum clique size, $\chi(G)$, the chromatic number, and $k(G)$, the clique cover number (the minimum number of cliques required to cover the vertices of $G$). A graph is **perfect** if $\chi(H) = \omega(H)$ for every subgraph of $G$. For each graph, we associate the polytope

$$P_{\text{clique}}(G) = \{x \in \mathbf{R}\langle V\rangle : x \geqslant 0, x(C) \leqslant 1, \ \forall \text{ cliques } C : x(C) \leqslant 1\}$$

We claim this polytope is integral if and only if $G$ is perfect.

Perfect graphs include the families of bipartite graphs and chordal graphs. Given a graph $G$, define the complement of $G$ to be the graph $\overline{G}$ obtained by 'flipping edges'. Then

$$\alpha(G) = \omega(\overline{G}) \quad \chi(G) = k(\overline{G})$$

It is an important theorem of Lovatz, proved in 1972, that a graph is perfect if and only if its complement is perfect. Beyond the scope of this course is the proof that $G$ is perfect if and only if $G$ and $\overline{G}$ contain no cyclic graphs of size $\geqslant 5$ (the strong perfect graph theorem).

To prove Lovatz's theorem, consider the graph $G^{+v}$, for $v \in V$, obtained by adding a new vertex connected to ever vertex but $v$.

**Lemma 4.1** (Replication Lemma)**.** *$G$ is perfect if and only if $G^{+v}$ is perfect.*

*Proof.* The lemma is trivial if $G$ has one vertex. For $|V| \geqslant 2$, it suffices to check $\chi(G^{+v}) = \omega(G^{+v})$. It is always true that $\alpha(G^{+v}) = \alpha(G) + 0/1$. If $\alpha(G^{+v}) = \alpha(G)$ □

So suppose $G$ is perfect. Let $x \in P_{\text{clique}}(G)$ be an extreme point. From what we know in class, we know $x \in \mathbf{Q}$. Thus there is an integer $\eta \neq 0$ such that $\eta x \in \mathbf{Z}$. So consider $H$, obtained by copying $G$ $\eta$ times, and then connecting all edges $(u_i, v_i)$, if $u = v$ or $(u, v)$ is an edge in the original graph. The lemma above implies $H$ is perfect. Let $C_H$ be a max clique in $H$, with $|C_H| = \omega(H)$.

# Chapter 5

# Polynomials

In this chapter, we discuss various computational problems which occur in the theory of polynomials. We will begin with the algebraic operations of addition and multiplication of polynomials. The naive addition operation is asymptotically optimal, but nontrivial techniques in the harmonic analysis of abelian groups allow us to compute multiplication in a much faster way than the naive multiplication algorithm.

First, we discuss the various ways we can represent polynomials computationally. The standard way to represent a degree $n$ polynomial $\sum a_k X^k$ is as a collection of $n+1$ numbers $a_0, a_1, \ldots, a_n$

As we are working over finite dimensional spaces, the Fourier transform can be given a matrix representation. First, note that there is a one to one correspondence between functions $a : \mathbf{Z}_n \to \mathbf{C}$, and polynomials $f(X) = (1/n)\sum_{k=0}^{n-1} a(k)X^k \in \mathbf{C}[X]$ of degree $n-1$. If $z$ is a primitive $n$'th root of unity, then $f(z^{-l}) = (1/n)\sum_{k=0}^{n-1} a(k)z^{-kl}$ gives the $l$'th Fourier coefficient of the function $a$. Thus the Fourier transform is just the values of the polynomial at the $n$'th roots of unity, and if we consider the Fourier transform with respect to the basis $1, X, X^2, \ldots, X^{n-1}$, then the matrix representation of the transform is given by the Vandermonde matrix

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & z^{n-1} & z^{n-2} & \ldots & z \\ 1 & z^{n-2} & z^{n-4} & \ldots & z^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z & z^2 & \ldots & z^{n-1} \end{pmatrix}$$

which is the standard trick to evaluate the polynomial as a matrix. The Fourier inversion formula tells us that the inverse of the Vandermonde matrix, which represents the matrix of the inverse Fourier transform, is

$$(1/n)\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & z & z^2 & \dots & z^{n-1} \\ 1 & z^2 & z^4 & \dots & z^{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z^{n-1} & z^{n-2} & \dots & z \end{pmatrix}$$

Essentially, this implies that a slight modification of the fast Fourier transform obtained by swapping the root $\nu$ used to calculate the transform with $\nu^{-1}$ gives us the inverse Fourier transform, provided we multiply the result by $1/n$ at the end.

Continuing with the computational aspect of discrete Fourier analysis, we now use the fast Fourier transform to multiply two polynomials $f(X) = \sum a_k X^k$ and $g(X) = \sum b_k X^k$, where $f(X)g(X) = \sum c_k X^k$. We can write $c_k = \sum a_i b_{k-i}$, and computing these values for all coefficients $c_k$ takes $O(n^3)$ operations, where $n$ is the degree of the polynomial. However, the definition of $c_k$ involves convolution, which tells us Fourier analysis is probably applicable. However, to employ the fast Fourier transform, we have to compute convolution in $\mathbf{Z}_m$, which is done via modulo arithmetic, the values of the polynomial are done via convolution over $\mathbf{N}$. The trick is that these two convolutions will be equal provided we set $m$ large enough, and pad the values of the polynomials with zeroes, so that $a_i = b_i = 0$ for $m > i > n$. Consider the convolution $\sum a_i b_{k-i}$, where $i$ and $k - i$ are taken as elements of $\mathbf{Z}_m$. Certainly every term defining $c_k$ occurs in this sum, so we need only choose $m$ such that we can add no additional terms, which will occur when $i > k$. These terms can be written as

$$\sum_{i=k+1}^{m-1} a_i b_{k-i} = \sum_{i=1}^{m-k-1} a_{k+i} b_{m-i}$$

where the right side sum now can be computed with considering any modulo tricks. If $a_{k+i} b_{m-i} \neq 0$, then $a_{k+i} \neq 0$, hence $0 \leqslant k + i \leqslant n$, and also $b_{m-i} \neq 0$, so $0 \leqslant m - i \leqslant n$. Adding together both these inequalities gives $0 \leqslant m + k \leqslant 2n$, hence $0 \leqslant m \leqslant 2n$. If we choose $m > 2n$, then this problem is avoided, and we can compute $c_k$ as convolution modulo $m$. Thus $c = a * b$,

and since the Fourier transform of $a * b$ is the pointwise product $\hat{a}\hat{b}$, we can compute the fast Fourier transform of $a$ and $b$ in $O(m \log m)$ operations, multiply them together with $m$ operations, and then compute the inverse Fourier transform in $O(m \log m)$ operations, giving us the values of $c$ in $O(m \log m) = O(n \log n)$ operations.

Another important use of the fast Fourier transform is to compute multiplication of two numbers. The naive algorithm runs in $O(\log^2 n)$ time, for numbers of size $\leqslant n$. However, we can use the fast Fourier transform to reduce the computation to $O(\log(n) \log\log(n) \log\log\log(n))$ (and all the logs are important, those the third iteration of the logarithm has almost no effect on the speed of the algorithm for the numbers we deal with (even $x = e^{1000000}$ has $\log\log\log x \leqslant 1$). Given two $n$ digit numbers $x$ and $y$, write

$$x = \sum_{k=0}^{n} a_k 2^k \qquad y = \sum_{k=0}^{n} b_k 2^k$$

Then

$$xy = \sum_{k=0}^{n} \left( \sum_{i=0}^{k} a_i b_{k-i} \right) 2^k$$

s

# Part I

# Approximation Algorithms

# Chapter 6

# Set Cover

In the set cover problem, we are given a particular set $X$, and a family of subsets $S_1, \ldots, S_n$ whose union is $S$. Each set $S_i$ is associated a particular weight $w_i \geqslant 0$, and the goal is to find the cheapest family of subsets which cover $S$. The case where $w_1 = w_2 = \cdots = w_n = 1$ is known as the unweighted version of the set cover problem. The problem is easily seen to be NP hard, because it generalizes the vertex cover problem, which asks, given a graph, to find a subset of vertices which contain one end to every edge.

## 6.1 Approximations Via Linear Programming

We cannot express the set cover problem as a linear program in polynomially many dimensions, because then we could solve the set cover program in polynomial time (using some linear program algorithm, like the ellipsoid method). However, we can express the set cover problem as a 0/1 *integer* linear program. Our choice of including $S_i$ in some solution to the set cover problem can be taken as some binary variable $x_i \in \{0,1\}$. The constraint that the union of the $S_i$ contain every element in $S$ can be expressed in terms of the $|S|$ constraints

$$\sum_{s \in S_i} x_i \geqslant 1$$

for each $x \in X$. We aim to choose a $\{0,1\}$ valued solution minimizing $\sum x_i w_i$. We can solve this problem by removing the $\{0,1\}$ integer constraint, and by adding the constraint that $0 \leqslant x_i \leqslant 1$ for each index $i$. This

is known as the *relaxation* of the integer linear program. This gives a solution $x^* : I \to [0,1]$. While $\sum x_i^* w_i$ will certainly be a smaller quantity than the set cover instance, we cannot really interpret the solution $x^*$ of the LP as a solution of the original set cover problem. However, we would hope that $x^*$ gives an approximate solution to the problem, and by *rounding* this solution to an integer solution, we can obtain a bounded approximation to the original problem.

In this case, we begin by setting $M$ to be the maximal number of sets which contain a particular element $x$, over all elements $x \in X$. To round $x^*$ to an integer valued solution, we define $x : I \to \{0,1\}$ by setting $x_i = 1$ if $x_i^* \geqslant 1/M$. Then $x$ gives values corresponding to a set cover, because since for each $s \in S$, $1 \leqslant \sum_{i \in s} x_i^* \leqslant M \max_{i \in s} x_i^*$, so that there is at least one index $i$ with $s \in S_i$ and $x_i = 1$. We claim that if OPT gives the optimal value of set cover, then $\sum w_i x_i \leqslant M \cdot \mathrm{OPT}$, implying that we have obtained an $M$-approximation algorithm for set cover. It is clear that $\sum w_i x_i^* \leqslant \mathrm{OPT}$, because every solution to set cover can be interpreted to a solution of the set cover LP. Furthermore, we know that $M x_i^* \geqslant 1$ whenever $x_i = 1$, and therefore

$$\sum w_i x_i \leqslant \sum_{i=1}^{n} w_i (M x_i^*) = M \sum w_i x_i^* \leqslant M \cdot \mathrm{OPT}$$

This concludes the discussion of the algorithm. In the case of vertex cover, each edge is covered by two vertices, and therefore this algorithm gives a 2-approximation.

Often, it is useful to use the dual of a linear program to verify how close we are to the original algorithm. In this case, the dual is to minimize $\sum_{i=1}^{n} y_i - \sum_{s \in S} y_s$, such that $w_i + y_i \geqslant \sum_{s \in S_i} y_s$, and $y_s, y_i \geqslant 0$. In an optimal solution to this linear program, we may always assume $y_i = 0$, for if $y_i > 0$, we can always let $y_i \to 0$ while decreasing some $y_s$ with $y_s > 0$ without changing the value of the solution (and if we have no $y_s > 0$ for $s \in S_i$, then the solution cannot be optimal because we still have a solution obtained by setting $y_i = 0$, and changing none of the values $y_s$). Thus a dual linear program can be obtained by removing the variables $y_i$, and considering the linear program which maximizes $\sum_{s \in S} y_s$, such that $\sum_{s \in S_i} y_s \leqslant w_i$, with $y_s \geqslant 0$. The theory of dual linear programs implies that $\sum_{s \in S_i} y_s$ gives a lower bound for any feasible solution $y_s$. We can use it to obtain another approximation algorithm. Consider an optimal solution $y^*$ to the dual LP, and take a solution $x$ to the original LP, with $x_i = 1$ if the corresponding

constraint in the dual LP is tight with respect to $y^*$. Now for every $s$, there must be an index $i$ such that $s \in S_i$, and $\sum_{s \in S_i} y_s = w_i$, hence $x_i = 1$, and so $s$ is covered. We claim that $x$ gives an $M$ approximation to the set cover problem, as for the original rounding solution to the linear program. The central idea is that if $S_i$ is in a cover, we 'pay' for it by charging $y_s^*$ to each $s \in S_i$. Each element is charged at most once for each set that contains it, so the total cost is at most $M \sum y_s^*$. In detail, we find that $x_i = 1$ if and only if $w_i = \sum_{s \in S_i} y_s^*$, so

$$\sum w_i x_i = \sum x_i \sum_{s \in S_i} y_s^* \leqslant M \sum_{s \in S} y_s^* \leqslant M \cdot \mathrm{OPT}$$

hence we have an $M$-approximation.

## 6.2 Using Duals to Combinatorialize LP Approximations

The dual algorithm is actually always worse than the original LP approximation. We know that $\sum y_s \leqslant \sum w_i x_i$ for any feasible $y$ in the dual, and $x$ in the original LP. What's more, $\sum w_i x_i^* = \sum y_s^*$. Because of the theory of complementary slackness, we also know that $y_s^* > 0$ implies that $\sum_{s \in S_i} x_i^* = 1$, and whenever $x_i^* > 0$, then $\sum_{s \in S_i} y_s^* = w_i$. In the first algorithm, we include a set in our cover if $x_i^* \geqslant 1/M$, and complementary slackness implies that the set is included in the cover determined by the second algorithm. Thus the second algorithm is always at least as inefficient as the first.

However, the dual LP is still useful, because we can use it in tandem with the original LP to obtain a combinatorial approximation algorithm for set cover. This is analogous to the combinatorial algorithms for bipartite matching and max flow, which can be obtained by combinatorializing the linear program formulations of the problem. The dual algorithm consists as follows: begin with some feasible solution $y$ to the dual program, and consider some family of sets $S_i$ such that the corresponding constraint to $S_i$ is tight. If this family is a set cover, then we are done. Otherwise, there is some $s \in S$ not covered by the family of sets, hence $y_s$ is not tight, and we may increase $y_s$ until it makes some dual constraint corresponding to some $S_j$ tight, and we add it to our family of sets. Eventually we obtain a set cover, and the same argument for the dual algorithm implies that this

solution is also an $M$ approximation. We know from the last argument that this will always give worse results than the original algorithm, but it is likely much faster in practice to compute this solution than to optimize the LP formulation of set cover.

## 6.3 A Greedy Solution

We now present a natural greedy heuristic to set cover, which actually leads to a much better approximation to set cover. At each point in the algorithm, we have a partial set cover which covers some subset $T \subset S$ of all the points in the set. If $T \neq S$, then we add the set $S_i$ to the cover which maximizes the ratio $w_i|S_i - T|^{-1}$ of weight to the number of points which can be added to the existing solution. If $H_n = \sum_{k=1}^{n} k^{-1}$ is the $n$'th *harmonic number*, then we claim this algorithm gives a $H_n$ approximation algorithm to the set cover problem. First, we note that given any $a_1, \ldots, a_n > 0$ and $b_1, \ldots, b_n > 0$,

$$\min(a_i/b_i) \leqslant \frac{\sum a_i}{\sum b_i} \leqslant \max(a_i/b_i)$$

this follows by simple algebraic manipulations. Suppose the greedy algorithm takes $m$ iterations to form a cover. Let $n = n_1 > n_2 > \cdots > n_m > n_{m+1} > 0$ be the number of uncovered elements on the beginning of the individual iterations of the algorithm, let $\varnothing T_1 \subsetneq T_2 \subsetneq \cdots \subsetneq T_m \subsetneq S$ denote the set of covered elements, and for each $i$, let $S_i^k = S_i - T_k$ denote the number of unconvered elements in $S_i$ at the beginning of the $k$th iteration. The greedy algorithm takes $j$ which minimizes $w_j/|S_j^k|$ on the $k$'th iteration. We claim that for this $j$,

$$w_j \leqslant \frac{n_k - n_{k+1}}{n_k} \cdot \mathrm{OPT}$$

It then follows that for the solution we find corresponding to the indices $I$,

$$\begin{aligned}\sum_{i\in I} w_i &\leqslant \mathrm{OPT} \sum_{k=1}^{m} \frac{n_k - n_{k+1}}{n_k} \\ &\leqslant \mathrm{OPT} \sum_{k=1}^{m} \left( \frac{1}{n_k} + \frac{1}{n_k - 1} + \cdots + \frac{1}{n_{k+1}+1} \right) \\ &= \mathrm{OPT} \sum_{i=1}^{n} \frac{1}{i} = H_n \cdot \mathrm{OPT}\end{aligned}$$

Given us the desired approximation constant. To prove the minimization inequality, we first find that

$$\min\left(\frac{w_j}{|S_j^k|}\right) \leqslant \frac{\min w_j}{\max |S_j^k|} \leqslant \frac{\mathrm{OPT}}{n_k}$$

If we add $S_j$ to our set, then $n_{k+1} = n_k - |S_j^k|$, and so

$$w_j \leqslant |S_j^k| \frac{\mathrm{OPT}}{n_k} = \frac{n_k - n_{k+1}}{n_k} \mathrm{OPT}$$

and this completes the argument.

The theory of dual LPs allows us to slightly tighten the approximation ratio of set cover. Let $N$ be the maximum size of any $S_i$. Then we claim the greedy solution returns a set cover indexed by some $J$, then $\sum_{j\in J} w_j \leqslant H_N \mathrm{OPT}_{\mathrm{LP}}$, where $\mathrm{OPT}_{\mathrm{LP}}$ is the optimal value of the LP relaxation of the set cover problem. To find this solution, we use the technique of dual fitting. The idea is that if $\sum_{j\in J} w_j = \sum y_s$ for some infeasible solution $y$ to the dual of the set cover with $y/H_m$ a feasable solution, then by weak duality we find that $\sum_{j\in J} w_j = \sum y_s \leqslant H_m \mathrm{OPT}_{\mathrm{LP}}$, giving us the required bounds. To construct the infeasible solution $y$, we consider the iterations of the greedy algorithm. If we add the set $S_j$ on the $k$'th iteration, we let $y_s = w_j/|S_j^k|$ for each $s \in S_j^k$. Then each $y_s$ is set exactly once, because it is contained in a particular set which is added to the cover, and is then covered for the rest of the algorithm. It is easy to see that $w_j = \sum y_s$ because of the way the weights are distributed over the LP. It remains to prove that $y/H_N$ is feasible. We must show that $\sum_{s\in S_i} y_s \leqslant H_N w_i$ for each index $i$. To see this,

fix $i$, and let $a_k = |S_i^k|$ be the number of elements of $S_i$ uncovered in the $k$th iteration, and let $A_k$ be the set of elements covered in $i$ covered in the $k$th iteration, so that $|A_k| = a_k - a_{k+1}$. If $S_j$ is chosen in the $k$th iteration, then for each $s \in A_k$,

$$\frac{y_s}{H_N} = \frac{w_j}{H_N |S_j^k|} \leqslant \frac{w_i}{H_N a_k}$$

Now we find that

$$\begin{aligned}
\sum_{s \in S_i} \frac{y_s}{H_N} &= \sum_{k=1}^{n} \sum_{s \in A_k} \frac{y_s}{H_N} \\
&\leqslant \sum_{k=1}^{n} (a_k - a_{k+1}) \frac{w_i}{H_N a_k} \\
&= \frac{w_i}{H_N} \sum_{k=1}^{n} \frac{a_k - a_{k+1}}{a_k} \leqslant \frac{w_j H_{|S_j|}}{H_N} \leqslant w_j
\end{aligned}$$

Giving us the required bounds and giving the approximation ratio for set cover.

The $H_n$ algorithm is essentially an asymptotically optimal approximation algorithm for set cover. If we can prove that there is a $c \lg n$ approximation algorithm for set cover, for any $c > 0$, then there is an $O(n^{O(\log \log n)})$ time algorithm for any problem in NP, implying P = NP. What's more, we find that P = NP even if there is a $c \lg n$ approximation algorithm for the *unweighted* case of the set cover problem. The 2 approximation algorithm for vertex cover is the best known ratio, and assuming the unique games conjecture, there is no $\alpha$ approximation algorithm for any $\alpha < 2$.

## 6.4 Randomized Rounding

We now consider another technique for obtaining a solution to a combinatorial problem from an LP relaxation. Given an optimal solution $x^*$ to the LP, $0 \leqslant x_i^* \leqslant 1$, which we would like to interpret as a probability that we choose the set $i$ in a randomized solution strategy to the set cover. Choosing the $S_i$ independently with probability $x_i^*$ gives a randomized family of sets whose expected weight is $\mathrm{OPT}_{\mathrm{LP}}$. This is a great value to obtain, but unfortunately, it is likely the solutions we obtain will not be covers of

the entire set. However, this calculation shows we can often obtain good approximation algorithms by this randomized rounding technique.

For each $s$, the chance that $s$ is not covered by the randomized rounding algorithm is $\prod_{s \in S_i}(1 - x_i^*)$. Using the fact that $1 - x_i^* \leqslant e^{-x_i^*}$, we can bound the chance that $s$ is not covered by

$$\prod_{s \in S_i} e^{-x_i^*} = e^{-\sum_{s \in S_i} x_i^*} \leqslant e^{-1}$$

because $\sum_{s \in S_i} x_i^* \geqslant 1$. Although $e^{-1}$ is only an upper bound on this probability, it is possible to take this bound arbitrarily closely, so in the worst case it is highly unlikely we will obtain a set cover instance.

However, we can still use these ideas to obtain polynomial time algorithms whose chance of failure is $n^{-c}$, for any positive constant $c > 0$. This is known as an algorithm that works with high probability (even though this is really a family of algorithms, and we normally find that the speed of these algorithms increases as $c \to \infty$). If we can find a randomized procedure operating in polynomial time such that the probability that any $s$ is uncovered is bounded by $(1/n^c)$, then a union bound gives that the probability that a random choice is a set cover is $(1/n^{c-1})$, so we have obtained an algorithm that works with high probability. The idea is to consider $c \lg n$ Bernoulli trials which succeed with probability $x_i^*$, and to take a set $S_i$ in our cover if any of these trials pass. The probability that $S_i$ is not included is then $(1 - x_j^*)^{c \lg n}$, which is upper bounded by

$$\prod_{s \in S_i} (1 - x_i^*)^{c \lg n} \leqslant \prod_{s \in S_i} e^{-x_i^*(c \lg n)} = e^{-(c \lg n) \sum_{s \in S_i} x_i^*} = \frac{1}{n^c}$$

Now we need only prove that this algorithm has a good expected value. Let $P_i(x_i^*) = 1 - (1 - x_i^*)^{c \lg n}$ be the probability that $S_i$ is included in our set. If $x_j^* \in [0, 1]$, and $c \lg n \geqslant 1$, then

$$P_i'(x_i^*) = (c \lg n)(1 - x_i^*)^{c \lg n - 1} \leqslant c \lg n$$

since $P_i(0) = 0$, we conclude that $P_i(x_i^*) \leqslant (c \lg n) x_i^*$. Let $X_i$ be the random variable with $X_i = 1$ if $S_i$ is included in our random set cover. Then the expected value of the given solution is then

$$\mathbf{E}\left(\sum w_i X_i\right) = \sum w_i P_i(x_i^*) \leqslant w_i (c \lg n) x_i^* = (c \lg n) \mathrm{OPT}_{\mathrm{LP}}$$

However, it is more important to bound the expected value, given that the given family we obtain is a set cover. Let $A$ be the event that the $X_i$ give a valid set cover. Then $\mathbf{P}(A) \geqslant 1 - n^{c-1}$, and also that

$$\mathbf{E}\left(\sum w_i X_i\right) = \mathbf{E}\left(\sum w_i X_i | A\right)\mathbf{P}(A) + \mathbf{E}\left(\sum w_i X_i | A^c\right)\mathbf{P}(A^c)$$

since $w_i, X_i \geqslant 0$, $\mathbf{E}(\sum w_i X_i | A^c) \geqslant 0$, and so

$$\mathbf{E}\left(\sum w_i X_i | A\right) \leqslant \frac{\mathbf{E}\left(\sum w_i X_i\right)}{\mathbf{P}(A)} \leqslant \frac{(c \lg n)}{1 - n^{c-1}} \mathrm{OPT}_{\mathrm{LP}} \leqslant (2c \lg n)\mathrm{OPT}_{\mathrm{LP}}$$

where we let $n, c \geqslant 2$.