Chapter 10

Minimum cuts

In this note, we first look at an algorithm due to Stoer and Wagner for finding minimum cuts in graphs without computing flows. We also look at a randomized approach to finding min cuts due to Karger and co-authors that allows us to bound the number of minimum cuts in a graph. Then, we describe two kinds of spanning trees of a capacitated undirected graph that capture the structure of minimum cuts in the graph - flow-equivalent and cut-equivalent trees.

We then present a simple 2-approximation algorithm for the problem of breaking up an undirected graph into k connected components for a pre-specified k using edges of minimum total capacity: this is the k-cut problem. The multiway cut problem is to find a minimum capacity set of edges or nodes whose removal puts a given set of terminals $s_1, s_2, \ldots, s_k$ in an undirected graph in different connected components. We finally describe a 2-approximation algorithm for this problem based on a half-integral relaxation of an integer programming formulation of the problem.

10.1 A simple minimum cut algorithm

10.1.1 Introduction

Let $G = (V, E)$ be a connected undirected graph. Given a node set $Q \subseteq V$, $\delta(Q)$ denotes the set of all edges with one end in Q and the other end in $V \backslash Q$. (Informally, $\delta(Q)$ is the "boundary" of the node set Q in G.) A cut consists of all edges that have one end in Q and the other end in $V \backslash Q$, where Q is a node set such that $Q \neq \emptyset$ and $Q \neq V$; this cut is denoted $(Q, V \backslash Q)$.

Let every edge $ij \in E$ be assigned a nonnegative capacity $c(ij)$. The capacity of a cut is defined as the sum of the capacities of the edges in it, i.e. $c(Q, V \backslash Q) = \sum_{ij \in \delta(Q)} c(ij)$. The minimum cut problem is to find a cut in G with the smallest capacity.

Background

The minimum cut problem arose first in studies relating to how much "flow" could be sent between a source and destination in a network. A flow can be intuitively thought of as a set of paths between the source and destination that share the capacities of the edges without over-using them. An obvious upper limit on the number of such flow paths between a pair of nodes is the capacity of any cut that puts the pair in opposite sides of the cut. The well-known maximum-flow minimum-cut theorem [6, 7, 21] asserts that the maximum number of such flow paths between a pair is equal to the capacity of a minimum cut separating them. Notice that we are not talking about the minimum cut in the graph here but a minimum cut separating the pair in question.

Connectivity of a graph

The capacity of a minimum cut in an undirected graph is also called the *edge-connectivity* of the graph. Consider the case of an undirected graph with all edge capacities being one. The edge-connectivity between a pair of nodes is simply the maximum number of paths between the pair such that the paths are pairwise edge-disjoint. In the sense of the previous paragraph, this is just the maximum flow between this pair, whose value equals the minimum cut separating them. The edge-connectivity of the whole graph is defined as the minimum edge-connectivity between any pair of nodes in it. Thus, trees have edge-connectivity one, a cycle has edge-connectivity two and a k-dimensional hypercube has edge-connectivity k (check this yourselves!).

The notion of edge-connectivity can be easily generalized to node-connectivity by thinking of the nodes as being capacitated. Thus we now consider simple graphs where each node is assumed to have a capacity of one. The node-connectivity between a pair of nodes is defined as the maximum number of paths between the pair such that these paths are pairwise node-disjoint except at the two extremes (source and destination itself). Such paths are sometimes also termed openly-disjoint. There is a counterpart of the maximum-flow minimum-cut theorem for node-capacitated graphs, which ascertains that the maximum number of openly disjoint paths between a pair of *nonadjacent* nodes equals the minimum capacity of any *node-cut* that separates them. This node-cut is defined as a set of nodes that puts the pair in two different connected components upon its removal.

An undirected graph is said to be k-edge connected if its edge-connectivity is k and k(-node) connected if its node connectivity is k.

Algorithms for finding cuts

The first algorithms for computing the minimum cut in a graph were based on finding maximum flows. Combined with a structure called Gomory-Hu trees, these methods involved n different calls to an algorithm to find a maximum-flow between a pair of vertices on an n-node graph[1]. The fastest flow algorithm currently available is based on the Push-Relabel technique of Goldberg and Tarjan [10, 19] and run in time $\tilde{O}(nm)$ on an n-node m-edge graph [2]. Hao and Orlin [15] extended this method to piggyback all the n flow computations in asymptotically the same time for one, giving an algorithm for finding the minimum cut in time $\tilde{O}(nm)$.

An alternate set of algorithms for finding the minimum cut do not use flow. They grew out of early work of Mader [20] that showed that every k-edge connected graph has a subgraph that is k-edge connected and has only $O(kn)$ edges. Subsequent work of Nishizeki and Poljak [25] showed how this subgraph can be constructed as a union of forests. Nagamochi and Ibaraki [22, 23] gave fast algorithms for constructing such subgraphs. A short proof of a generalization of the results of Nagamochi and Ibaraki to mixed cuts containing both edges and nodes was presented by Frank,

[1] Actually, you don't really need to know about the Gomory-Hu tree to show that you can find the min cut in at most n flow computations. Find a simple way to do this yourself by a suitable choice of source-destination pairs.

[2] The $\tilde{O}$ hides poly-logarithmic factors.

Ibaraki and Nagamochi [8]. Since we only examining the edge-version of the minimum cut problem we follow an even simpler treatment due to Stoer and Wagner [27]. The paper by Frank, Ibaraki and Nagamochi remains a valuable reference for the most general result provable using this approach.

10.1.2 The Stoer-Wagner Algorithm

First we will introduce a simple operation on a graph G involving a pair of distinct nodes u and v called *node identification*. This operation simply identifies these two nodes into one new node, deleting self loops if any but retaining parallel edges; the resulting graph is denoted by G_{uv}. The following proposition sheds light on the effect of a node identification on a minimum cut in a graph.

Proposition 10.1 *The cuts of the graph G_{uv} are exactly the cuts of G that do not separate u and v.*

Therefore, we infer that a minimum cut in G is the minimum of two quantities: the minimum cut in G_{uv} and a minimum cut separating u and v (i.e., with these two nodes on the two sides of the cut). The proposition guarantees that these two cases are mutually exhaustive. Furthermore we have the following simple algorithm for finding a minimum cut.

Algorithm Node Identification Algorithm for finding a Minimum Cut
input: Graph $G = (V, E)$, nonnegative edge capacities c
output: A subset of nodes $S \subset V$ such that $\delta(S)$ is a minimum capacity cut of G.

STEP 1: initialize cut $S \leftarrow$ undefined and capacity $C \leftarrow \infty$;
    while G has more than one node do
STEP 2:   Pick two distinct nodes s and t and
        compute a minimum capacity cut $\delta(S')$ separating s and t;
STEP 3:   **if** $c(S', V \backslash S') < C$
STEP 4: $C \leftarrow c(S', V \backslash S')$ and $S \leftarrow S'$;
        end (if);
STEP 5:   replace G by G_{st};
    end (while);
    output the cut $\delta(S)$;

At the outset this algorithm does not appear particularly better than the earlier algorithms - in fact, it still requires the computation of $n-1$ minimum cut and hence maximum flow computations, on an n-node graph. The only minor advantage is that the size of the graph on which we are computing the flow is deceasing by one as we progress in the algorithm.

A critical advantage of the above algorithm reveals itself from a close inspection of STEP 2. In this step we have the choice of picking *any* two surviving nodes s and t in the graph to find a minimum cut separating s and t. The idea is to choose a pair s, t such that this minimum cut is easy to find without using flows.

Legal Orderings

Define an ordering of the nodes of G, say, $v_1, v_2, \ldots, v_n$ to be *legal* if v_1 is arbitrary and for every $i \geq 2$, v_i is the node with the maximum total capacity of edges joining it to the nodes $\{v_1, v_2, \ldots, v_{i-1}$. If we use $c_j(v)$ to denote $\sum_{x=1}^{j} c(v_x v)$, then v_i is a node maximizing $c_{i-1}(v)$ over all the remaining nodes v. The usefulness of legal orderings stems from the following result.

Theorem 10.2 *If $v_1, \ldots v_n$ is a legal ordering of the vertices of G, then $\delta(\{v_n\})$ is a minimum cut separating v_{n-1} and v_n in G.*

For any node subset V' of G, let $G(V')$ denote the subgraph of G induced by the nodes in V', namely, only the set of edges with both endpoints in V'. The theorem follows from the following lemma about legal orderings.

Lemma 10.3 *Let $v_1, \ldots v_n$ be a legal ordering of the vertices of G. Let v be any vertex in $\{v_{i+1}, v_{i+2}, \ldots, v_n\}$. Then in the graph $G(\{v_1, v_2, \ldots, v_i, v\})$, the minimum cut separating v and v_i has capacity at least $c_i(v)$.*

Proof: The proof is by induction on i. The basis when $i = 1$ is trivial. Assume the lemma is true for $i-1$; we'll prove for i. Consider a vertex $v \in \{v_{i+1}, \ldots, v_n\}$. By the induction hypothesis, in the graph $G(\{v_1, v_2, \ldots, v_{i-1}, v\})$, the minimum cut separating v and v_i has capacity at least $c_{i-1}(v)$. Similarly, in the graph $G(\{v_1, v_2, \ldots, v_i\})$, the minimum cut separating v and v_i has capacity at least $c_{i-1}(v_i)$. These imply that in the graph $G(\{v_1, v_2, \ldots, v_i, v\})$, the minimum cut separating v and v_i has capacity at least $c_i(v)$.

Suppose for a contradiction that in the graph $G(\{v_1, v_2, \ldots, v_i, v\})$, the minimum cut separating v and v_i has capacity less than $c_i(v)$. Since $c_i(v) = c(v_i v) + c_{i-1}(v)$, this implies that after deleting the edge $v_i v$ (if it exists) from this graph, the minimum cut separating v and v_i has capacity less than $c_{i-1}(v)$. Call this cut C. Consider which side of this cut lies the vertex v_{i-1}.

Case 1. If v_{i-1} lies on the same side of C as v, then C is also a cut separating v_{i-1} and v_i in the graph $G(\{v_1, v_2, \ldots, v_i\})$. Thus the capacity of C is at least $c_{i-1}(v_i)$. By the choice of v_i in the legal ordering, $c_{i-1}(v_i) \geq c_{i-1}(v)$. This contradicts the assumption that C has capacity less than $c_{i-1}(v)$.

Case 2. If v_{i-1} lies on the opposite side of C to v, then C is also a cut separating v_{i-1} and v in the graph $G(\{v_1, v_2, \ldots, v_{i-1}, v\})$. Thus its capacity must be at least $c_{i-1}(v)$, a contradiction.

This completes the proof of the lemma. ∎

How efficiently can a legal ordering be found? The key step is to find the next vertex in the ordering by looking for the vertex with the maximum capacity into the set of already assigned nodes. This step is similar to a step of Dijkstra's shortest path algorithm or a step of Prim's minimum spanning tree algorithm. By using similar data structures a legal ordering of an n-node m-edge graph can be found in time $O(m + n \log n)$.

Min cut algorithm using legal orderings

A legal ordering identifies a pair of vertices, v_{n-1} and v_n such that the cut around the singleton set $\{v_n\}$ is a minimum cut separating v_{n-1} and v_n. Notice that we cannot force a given pair of vertices to be the last two vertices of a legal ordering. Nevertheless, after an efficient computation, we find a pair of vertices and a minimum cut separating them. By using this pair of vertices in STEP 2 of the node identification algorithm, we can implement each iteration of the algorithm efficiently giving an overall running time of $O(mn + n^2 \log n)$ for our minimum cut algorithm.

10.1.3 A bound on the number of min cuts

A randomized version of the node identification algorithm due to Karger and Stein [18] can be used to show a simple upper bound on the number of minimum cuts in an undirected graph. Let $c(e)$ denote the capacity of an edge $e \in E$; we extend the notation and use $c(E)$ to denote the total capacity of all the edges in the graph.

Algorithm Random Contraction Algorithm
input: Graph $G = (V, E)$, nonnegative edge capacities c
output: A set of edges forming a minimum cut in G with probability at least $\frac{2}{n(n-1)}$.

while G has more than two nodes do
STEP 1: choose an edge randomly where edge e is chosen with probability $\frac{c(e)}{c(E)}$.
STEP 2: **if** the chosen edge is uv, replace G by G_{uv}.
output the set of edges forming the unique cut of the resulting graph G;

Theorem 10.4 *Fix C to be a minimum cut of a graph G on n nodes. Then the probability that the random contraction algorithm running on G outputs C is at least $\frac{2}{n(n-1)}$.*

Proof: We would like to lower bound the probability of the good event that no edge of C is touched in the course of the contraction algorithm.

First let us estimate the probability that an edge of C is contracted in the first iteration of the algorithm when the first edge to be contracted is chosen.

$$Pr(\text{an edge of } C \text{ is contracted}) = \frac{c(C)}{c(E)}$$

where $c(C)$ denotes the total capacity of the cut C. Note that we can write $c(E) = \frac{1}{2}\sum_{v\in G} c(\{v\}, V\backslash\{v\})$. Since C is a minimum cut, we have for any $v \in G$ that $c(C) \le c(\{v\}, V\backslash\{v\})$. Substituting back we get that $c(E) \ge \frac{1}{2} \cdot n \cdot c(C)$ and that

$$Pr(\text{an edge of } C \text{ is contracted}) \le \frac{2}{n}$$

It is straightforward to extend this argument to show that after i contractions by the algorithm,

$$Pr(\text{an edge of } C \text{ is contracted}) \le \frac{2}{n-i}$$

Therefore after i contractions, assuming that C survives,

$$Pr(\text{an edge of } C \text{ is not contracted}) \geq 1 - \frac{2}{n-i} = \frac{n-i-2}{n-i}$$

Finally we have that

$$\begin{aligned} Pr(\text{the algorithm outputs } C) &= Pr(\text{no edge of } C \text{ is ever contracted}) \\ &\geq (\frac{n-2}{n})(\frac{n-2-1}{n-1})\cdots(\frac{n-2-(n-3)}{n-(n-3)}) \\ &\geq \frac{(n-2)(n-3)\ldots 2\cdot 1}{n(n-1)\ldots 4\cdot 3} \\ &\geq \frac{2}{n(n-1)} \end{aligned}$$

∎

We have the following corollary.

Corollary 10.5 *The number of distinct minimum cuts in an undirected graph is at most* $\frac{n(n-1)}{2}$.

The corollary follows by using the Theorem to observe that for every distinct minimum cut, the probability of the algorithm returning that particular cut is at least $\frac{2}{n(n-1)}$. Returning a pair of distinct minimum cuts is a pair of mutually exclusive events and the bound follows.

This bound on the number of different minimum cuts is the best possible in the sense that there is an n-node graph with exactly $\frac{n(n-1)}{2}$ cuts. Namely, each pair of edges in an n-cycle is a minimum cut.

Karger and co-workers have extended this paradigm to provide a series of algorithms with better and better running times and better success probabilities of returning a minimum cut. For details of these methods and an empirical evaluation of the different approaches to computing minimum cuts, see [3] and the references therein.

10.2 Gomory-Hu Trees: Existence

In this section, we look at two kinds of spanning trees of a capacitated undirected graph that capture the structure of minimum cuts in the graph.

10.2.1 Introduction

Let $G = (V, E)$ be a connected undirected graph, and as before, let every edge $ij \in E$ be assigned a nonnegative capacity $c(ij)$. We define two types of trees on the same vertices as G. Note that we do *not* require that the edges of T be a subset of the edges of G – in both cases, the tree T is only intended to capture the structure of minimum cuts in G.

A **cut-equivalent tree** (relative to G) is defined as a tree T with $V(T) = V(G)$ with the following property: for every edge $xy \in E(T)$, the removal of xy from T partitions $V(T) = V(G)$

into two subsets $X \ni x$ and $Y \ni y$; then we require the cut $\delta(X)$ in G to be a minimum capacity cut separating x and y.

A **flow-equivalent tree** is defined as a tree with $V(T) = V(G)$ along with nonnegative capacities c' assigned to the edges of T with the following property: for any pair of nodes x and y, the maximum flow between x and y has the same value in both G and T. By the max-flow min-cut theorem, this translates to requiring that the minimum cut separating x and y has the same value in both G and T.

Note that a cut-equivalent tree is uncapacitated while the flow-equivalent tree is capacitated. However, there is a natural assignment of capacities c' to the edges of a cut-equivalent tree: for every edge xy in the tree, let X be the component of $T - xy$ containing x; then we assign $c'(xy) = c(X, V\backslash X)$, namely, the capacity of the cut $\delta(X)$ separating x and y in G. A cut-equivalent tree with these capacities is called a **Gomory-Hu tree**, named after its discoverers Gomory and Hu [12].

First we begin with a simple observation.

Proposition 10.6 *Let $v_1, v_2, \ldots, v_k$ be a sequence of distinct nodes of G. Let c_{ij} denote the value of a minimum cut separating v_i and v_j. Then*

$$c_{ik} \geq \min\{c_{12}, c_{23}, \ldots, c_{k-1,k}\}$$

Proof: Consider tracing the path from v_1 to v_k along $v_2, \ldots, v_{k-1}$ (this path need not exist in G). This path must cross any cut separating v_1 and v_k at least once. Hence, a minimum cut of value c_{1k} separating v_1 and v_k must also separate a consecutive pair of nodes v_i and v_{i+1} for some $i \in \{1, 2, \ldots, k-1\}$. This implies the claim. ∎

Proposition 10.7 *A cut-equivalent tree is also flow-equivalent, when each edge is assigned its capacity in the natural way described above.*

Proof: Let T be the cut-equivalent tree with capacities c', and let s and t be a pair of distinct nodes. We'll show that the minimum cut separating s and t has value equal to the minimum c'-value of an edge in the $s - t$ path in T. Let this path be $P = \{s = v_1, v_2, \ldots v_k = t\}$. Applying Proposition 10.6 to P yields that $c_{st} \geq c_{i,i+1} = c_{xy}$ for some pair $v_i = x$ and $v_{i+1} = y$ of consecutive vertices in the path.

Since T is cut-equivalent, the cut induced by the two components of $T - xy$ is a minimum cut separating x and y of value c_{xy} in G. Note that this cut also separates s and t, and has value c_{xy}, giving $c_{st} \leq c_{xy}$.

Combining the two inequalities above, we see that $c_{st} = c_{xy}$, which shows that T is flow-equivalent. ∎

10.2.2 Warm-up: Maximum spanning trees

The existence of flow-equivalent trees implies that there is a lot of redundancy in the minimum cut values between different pairs of vertices. Among the possible $\frac{n(n-1)}{2}$ values, there are at most $n - 1$ distinct ones represented in the flow-equivalent tree. Note that the minimum cut between a pair of vertices in the flow-equivalent tree is the smallest capacity of an edge in the unique tree path between them.

To build intuition, we will first show a simple proof of existence of flow-equivalent trees. This proof will imply an algorithm for constructing such trees which involves carrying out $\frac{n(n-1)}{2}$ minimum cut computations. In the next section, we will strengthen this result in two directions: by reducing the number of cut computations to $n-1$ and by building a cut-equivalent tree.

Given an undirected graph $G = (V, E)$ with nonnegative weights w_e on its edges $e \in E$, a maximum spanning tree of G is a spanning tree of maximum total weight in G. Methods for finding minimum spanning trees extend to find maximum spanning trees as well. For example, a Kruskal-like algorithm adds edges in decreasing order of weight disregarding edges that form a cycle, until a spanning tree is formed. A Prim-like algorithm grows a core of nodes and extends the core by always using the maximum weight edge coming out of the core to identify the next vertex to pull into the core. A simple property of maximum weight spanning trees is summarized below.

Proposition 10.8 *Let T be a maximum spanning tree and let $f = xy$ be a non tree edge. Then $w_f \le w_e$ for an edge tree e on the path between x and y in T.*

Proof: The argument is similar to that for minimum spanning trees. Assume for a contradiction that $w_f > w_e$ for appropriate edges e and f. Replacing e with f in T results in a different tree $T - e + f$ that has more weight than T, contradicting that T is maximum weight. ∎

To show that a capacitated undirected graph G has a flow-equivalent tree, we construct a complete weighted graph G^* with the same nodes as G. We define the weight of an edge xy in G^* to be the value c_{xy} of a minimum cut separating x and y in G.

Lemma 10.9 *A maximum spanning tree of G^* is a flow-equivalent tree for G.*

Proof: Let T be a maximum spanning tree of G^*, and let s and t be a pair of distinct nodes in T. Let $P = \{s = v_1, v_2, \ldots v_k = t\}$ be that path in T between s and t.

Suppose $k = 2$ and st is an edge of T, then trivially, by the definition of the weight function of G^*, the minimum cut separating s and t has value equal to the weight of the edge st in T.

Now suppose $k \ge 3$ and so the edge st is not in T. By Proposition 10.6, $c_{st} \ge c_{i,i+1} = c_{xy}$ for some pair $v_i = x$ and $v_{i+1} = y$ of consecutive vertices in the path P. By Proposition 10.8, $c_{st} \le c_{xy}$. Combining these gives that $c_{st} = c_{xy}$ and thus T is flow-equivalent for G. ∎

The above proof gives a simple algorithm for constructing a flow-equivalent tree. First run $\frac{n(n-1)}{2}$ minimum cut computations to determine the weights of the edges in G^*. Then we can use a maximum spanning tree algorithm to compute T from G^*.

10.3 Gomory-Hu Trees: Construction

In this section, we present an algorithm for constructing a cut-equivalent tree of an n-node graph using only $n-1$ cut computations. Before that we present a few preliminaries on uncrossing cuts.

10.3.1 Uncrossing cuts

A pair of sets S and Q (where $\emptyset \neq S, Q \neq V$) are said to be *crossing* if each of the four sets $S \cap Q, S - Q, Q - S$ and $V - Q - S$ is non-empty. A pair of cuts $\delta(S)$ and $\delta(Q)$ are said to be crossing if the sets S and Q are crossing sets.

In the sequel, we use the term $s-t$ minimum cut to denote a cut of minimum capacity separating a pair of distinct vertices s and t in G. Also we will always use $Y = V\backslash X, B = V\backslash A$ and similarly for Y' and B'.

A real-valued set function $f : 2^V \to \Re$ defined on node-subsets is *submodular* if it satisfies

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$$

for all $X, Y \subseteq V$. We may extend the definition and call f *weakly submodular* if it satisfies

$$f(X) + f(Y) \geq \max(f(X \cup Y) + f(X \cap Y), f(X - Y) + f(Y - X))$$

for all $X, Y \subseteq V$.

Consider the function defined on $S \subseteq V$ as $c(S) = c(S, V\backslash S)$, the capacity of the cut $\delta(S)$.

Proposition 10.10 *The capacity function on node subsets $c : 2^V \to \Re_+$ is weakly submodular.*

Proof: The proof proceeds by showing the following two inequalities.

(i) for all $X, A \subseteq V$,
$$c(X) + c(A) \geq c(X \cup A) + c(X \cap A)$$

(ii) for all $X, A \subseteq V$,
$$c(X) + c(A) \geq c(X - A) + c(A - X)$$

The proof for both parts follows from a simple counting argument: consider two cuts $\delta(X)$ and $\delta(A)$ and look at the different kinds of edges in the right-hand side to verify that they occur with at least the same multiplicity in the left hand side as well. See Figure 10.3.1. ∎

The main application of the weakly submodular property of cuts is in *uncrossing* a pair of crossing cuts. By uncrossing this pair of cuts, we mean replacing them by another pair, say $\delta(X')$ and $\delta(A')$ such that X' and A' do not cross, and the two new cuts have the same properties as the old cuts. For example, if $\delta(X)$ and $\delta(A)$ were both minimum cuts in G, than $\delta(X')$ and $\delta(A')$ will also both be minimum cuts.

As another example, suppose $\delta(X)$ and $\delta(A)$ were $x - y$ and $a - b$ minimum cuts respectively; Then, the pair of uncrossed cuts $\delta(X')$ and $\delta(A')$ will also be $x - y$ and $a - b$ minimum cuts respectively. We show these two applications of weak submodularity to uncrossing cuts in the next two lemmas.

Lemma 10.11 *Suppose $\delta(X)$ and $\delta(A)$ are crossing minimum cuts of G. Then $\delta(X \cap A)$ and $\delta(X \cup A)$ are also minimum cuts and they are non-crossing. Similarly, $\delta(X - A)$ and $\delta(A - X)$ are also non-crossing minimum cuts.*

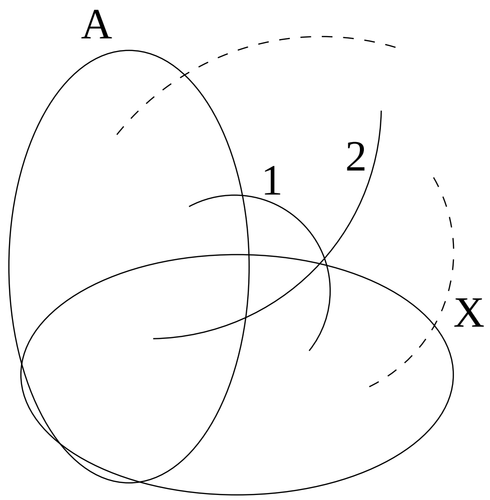

Figure 10.1: Proof of Proposition 10.10. The dotted edges are counted with the same multiplicity in both inequalities. Edges of the type labeled 1 are not counted in the right-hand side of (i), while edges of type 2 are not counted in the right-hand side of (ii). This results in the inequality rather than equality.

Proof: The proof is a direct application of proposition 10.10. Part (i) gives that

$$c(X) + c(A) \geq c(X \cup A) + c(X \cap A)$$

Since X and A cross, both $\delta(X \cup A)$ and $\delta(X \cap A)$ are nonempty sets of edges and have capacity at least that of a minimum cut. But both terms on the left hand side are values of the minimum cut by definition and so both $\delta(X \cup A)$ and $\delta(X \cap A)$ must be minimum cuts as well. The proof for the other pair is similar using the second part of proposition 10.10. ∎

Lemma 10.12 *Let $\delta(S)$ be an $s - t$ minimum cut where $s \in S$ and $t \in T = V \backslash S$, and let $\delta(X)$ be an $x - y$ minimum cut, where $x \in X$ and $y \in Y = V \backslash X$. Assume that S and X cross. Then there exists a pair $\delta(S')$ and $\delta(X')$ of noncrossing of $s - t$ and $x - y$ minimum cuts in G.*

Proof: The proof is essentially a lengthy case analysis investigating the various relative locations of the vertices x, y, s and t, and using Proposition 10.10 extensively. We will consider two possible cases for the location of x and y relative to s and t.

Case A: Both x and y are in the same side of the cut $\delta(S)$, say in S. Without loss of generality, assume that $s \in S \cap X$. There are two subcases to consider. (see Figure 10.3.1).

1. $t \in T \cap Y$: Apply proposition 10.10 (i) to the pair of crossing cuts $\delta(S)$ and $\delta(X)$ to get

$$c(S) + c(X) \geq c(S \cap X) + c(S \cup X)$$

Note that $\delta(S)$ and $\delta(X)$ are minimum cuts for the pairs $s - t$ and $x - y$ respectively. Furthermore, $\delta(S \cap X)$ separates x and y while $\delta(S \cup X)$ separates s and t in this case. Thus the inequality must be tight with $\delta(S \cap X)$ being a minimum cut separating x and y and $\delta(S \cup X)$ being a minimum cut separating s and t.

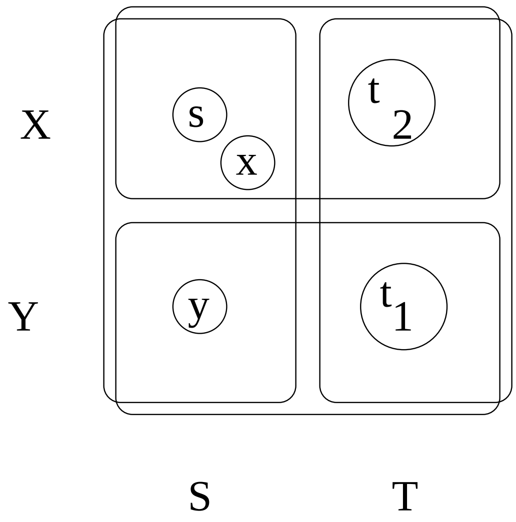

Figure 10.2: Case A in the proof of Lemma 10.12. The location of the node t in the two subcases 1 and 2 are shown as t_1 and t_2 respectively.

2. $t \in T \cap X$: The proof is similar. We apply proposition 10.10 (i) to the pair $\delta(S)$ and $\delta(Y)$ in this case to obtain
$$c(S) + c(Y) \geq c(S \cap Y) + c(S \cup Y)$$
Again $\delta(S)$ and $\delta(X)$ are minimum cuts for the pairs $s-t$ and $x-y$ respectively. Furthermore, $\delta(S \cap Y)$ separates x and y while $\delta(S \cup Y)$ separates s and t in this case. Thus the inequality must be tight with $\delta(S \cap Y)$ being a minimum cut separating x and y and $\delta(S \cup Y)$ being a minimum cut separating s and t.

Case B: x and y are in opposite sides of the cut $\delta(S)$, say $x \in S$ and $y \in T$. There are two subcases to consider depending on the location of s. (see Figure 10.3.1).

1. $s \in S \cap X$. In this case, the cut $\delta(S \cap X)$ is an $s - t$ cut while $\delta(S \cup X)$ is an $x - y$ cut. By submodularity, both are also minimum.
2. $s \in S \cap Y$. There are two further subcases depending on the location of t. In the first and more routine case, $t \in T \cap Y$ (denoted t_{21} in figure 10.3.1). In this case, $\delta(Y \cap T)$ is an $s - t$ cut while $\delta(Y \cup T)$ is an $x - y$ cut. Again, by submodularity, they are both minimum.

   The more interesting subcase is when $t \in T \cap X$ (denoted t_{22} in the figure). Note that we cannot directly apply submodularity to any pair of crossing cuts here. However, in this case, $\delta(S)$ is an $x - y$ cut while $\delta(X)$ is an $s - t$ cut. Since both cuts are minimum, they both have the same value. Applying weak submodularity for a single pair of cuts, say $\delta(S)$ and $\delta(X)$ now shows that all four cuts $\delta(S \cap X), \delta(S \cup X), \delta(S - X)$ and $\delta(X - S)$ have the same value. We can now use $\delta(S \cap X)$ and $\delta(S \cap Y)$ as the pair of minimum $x - y$ and $s - t$ cuts respectively.

∎

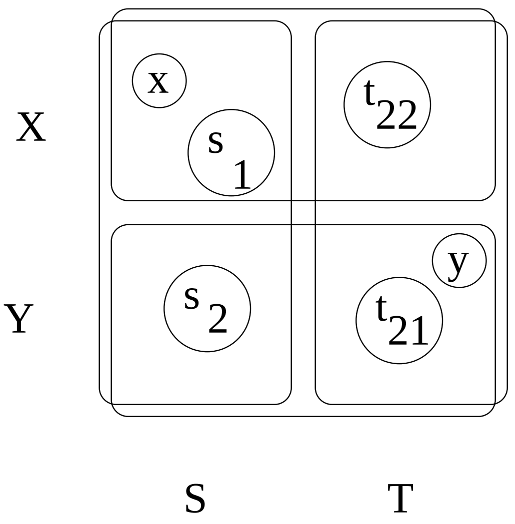

Figure 10.3: Case B in the proof of Lemma 10.12. The location of the node s in the two subcases 1 and 2 are shown as s_1 and s_2 respectively. In the subcase 2 when s is located at s_2, the two possible locations of the node t are labeled t_{21} and t_{22} respectively.

Corollary 10.13 *Consider an undirected graph G and let s and t be distinct nodes. Also let x and y be distinct nodes such that x may be a and y may be b. Let $\delta(X)$ be an $x-y$ minimum cut, where $x \in X$ and $y \in Y = V \backslash X$. Let $G(x^*)$ denote the graph obtained from G by identifying all the nodes in X to a single node, and let $G(y^*)$ be similarly defined. Then, either $\delta(X)$ is a minimum $s-t$ cut, or there is a minimum $s-t$ cut of G that is also a cut in either in $G(x^*)$ or $G(y^*)$.*

Proof: Suppose $\delta(X)$ is not a minimum $s-t$ cut. Let $\delta(S)$ be a minimum $s-t$ cut. If S and X do not cross, then the side of $\delta(X)$ containing S continues to contain a minimum $s-t$ cut of G.

Otherwise, S and X cross and we can use the proof of Lemma 10.12 to conclude that there is a minimum $s-t$ cut in G that is also a cut in either $G(x^*)$ or $G(y^*)$. ∎

10.3.2 The construction algorithm

The algorithm for constructing a cut-equivalent tree starts with the tree being a single "supernode" containing all the nodes of G. At each step, a supernode containing at least a pair of real nodes is split by computing a minimum cut between the pair, and a tree edge is added between the two resulting supernodes. The procedure stops when every supernode contains exactly one original node of G at which point we have a tree spanning the nodes of G.

Algorithm Cut-Equivalent Tree
input: Graph $G = (V, E)$, nonnegative edge capacities c.
output: A cut-equivalent tree T for G.
STEP 1: initialize T to be a single supernode containing all nodes of G;

```
        while there is a supernode containing more than a single node do
STEP 2:     pick a supernode U containing nodes x and y;
STEP 3:     form a new contracted graph G_c^U as follows:
            for each subtree of T attached to U,
STEP 4:         contract to a single node all the nodes of G contained in this subtree;
                (delete self loops but retain parallel edges)
STEP 5:     find a min cut δ(X) separating x and y in G_C^U;
STEP 6:     update T as follows:
STEP 7:         split supernode U into two supernodes X and Y
                partitioning the nodes in U according to the cut defined by X;
                for every supernode W adjacent to U originally
STEP 8:             reconnect W to either X or Y
                    depending on which side of the cut δ(X) that W appeared in G_c^U;
        output the tree T;
```

The time complexity of the algorithm is dominated by the minimum cut computations in step 5, and there are $n-1$ calls to this procedure for an n-node graph.

Correctness

Theorem 10.14 *The algorithm above outputs a cut-equivalent tree.*

The theorem is proved by showing the following lemma[3]. Let T_i represent the tree after i iterations of the algorithm through steps 2-8, i.e., after i minimum cut computations. Note that any edge T_i represents a unique cut in G defined by the bipartition of nodes according to this edge in T_i. Also the cut represented by any edge in T_i continues to be represented in all the subsequent trees $T_j,\ j > i$.

Lemma 10.15 *Consider the tree T_i. Let a and b be any two nodes in G where $a \in S(a)$ and $b \in S(b)$ for supernodes $S(a)$ and $S(b)$ in T_i. Suppose that the supernodes $S(a)$ and $S(b)$ are either identical or separated by at most one edge in T_i. Then there is a minimum $a-b$ cut such that*

(i) all of the edges of the cut are either in $G_c^{S(a)}$ or in $G_c^{S(b)}$ or

(ii) this cut is defined by the edge separating $S(a)$ and $S(b)$ in T_i.

Note that case (ii) of the claim applies only when $S(a)$ and $S(b)$ are distinct supernodes that are adjacent in T_i.

Proof: The proof is by induction on i. The basis when $i=0$ corresponds to a single supernode and the minimum $a-b$ cut obviously appears in the graph induced by all the nodes. Suppose the claim holds for i, we'll prove it for $i+1$. Suppose in T_i that $a \in S(a)$ and $b \in S(b)$. By the inductive hypothesis, the minimum $a-b$ cut is either in $G_c^{S(a)}$ or $G_c^{S(b)}$ (assume without loss of generality it

[3] Thanks to Rachel Rue for proposing the lemma that unifies two lemmas suggested earlier for the proof.

is in $G_c^{S(a)}$ in this case), or is the cut represented by the edge between the two distinct supernodes $S(a)$ and $S(b)$ in T_i. We consider these two cases separately.

1. The minimum $a-b$ cut is in $G_c^{S(a)}$. Let $\delta(A)$ be the minimum $a-b$ cut and $B = V(G_c^{S(a)})\backslash A$. If the $(i+1)$st iteration puts a and b in supernodes that are more than one edge away in T_{i+1} we cease to worry about this pair of nodes in enforcing the inductive claim.

   Suppose that a and b are in supernodes at most one edge away in T_{i+1}. Then the cut in $G_c^{S(a)}$ between them will be affected only if the $(i+1)$st cut was found on the supernode $S(a)$, i.e., in this iteration, in step 2, $U = S(a)$. In the following, if $S(a) \neq S(b)$, replace b by b^*, the supernode representing $S(b)$ in $G_c^{S(a)}$. Suppose the minimum $x-y$ cut found by the algorithm in the graph $G_c^{S(a)}$ in step $i+1$ is $\delta(X)$, where $a \in X$ without loss of generality. Note that x may be identical to a and/or y may be identical to b. Let $Y = V(G_c^{S(a)})\backslash X$. The supernode containing a after this iteration is $S'(a) = S(a)\backslash Y$, and that containing b is $S'(b) = S(b)\backslash X$. By Corollary 10.13 with $G = G_c^{S(a)}$ and $A = S$ and $B = T$, the minimum $a-b$ cut $\delta(A)$ of $G_c^{S(a)}$ is either $\delta(X)$ or it survives in one of $G_c^{S'(a)}$ or $G_c^{S'(b)}$. In the former case, we have a cut in T_{i+1} of type (ii), while in the latter, it is of type (i).

2. The minimum $a-b$ cut is defined by the edge separating $S(a)$ and $S(b)$ in T_i, where $S(a) \neq S(b)$. If the $(i+1)$st iteration puts a and b in supernodes more than one edge away, we do not worry about enforcing the inductive claim for the pair.

   Otherwise, irrespective of whether the cut computation in step $i+1$ involves node $S(a)$ or $S(b)$, the edge separating the supernodes $S'(a)$ and $S'(b)$ in T_{i+1} containing a and b represents the same cut as the edge between $S(a)$ and $S(b)$ in T_i. This is a cut of type (ii).

Notice that all pairs a, b obeying the condition of the lemma in T_{i+1} must have obeyed this condition even in T_i and hence the proof is complete. ∎

For an alternate version of the algorithm that allows for easy integration with existing code for finding minimum cuts, see [14]. This version does not require contractions and the necessary operations that go with it in finding the cut-equivalent tree. Other interesting properties of cut and flow-equivalent trees appear in [1, 16].

10.4 Multicuts

In this section, we examine an approximation algorithm for the problem of breaking up an undirected graph into k connected components for a pre-specified k using edges of minimum total capacity.

10.4.1 Introduction

Let $G = (V, E)$ be a connected undirected graph, and let every edge $ij \in E$ be assigned a non-negative capacity $c(ij)$. For a given positive integer k, the minimum k-cut problem is to find a set of edges of minimum total capacity whose removal from G leaves at least k connected components. Note that since the capacities are nonnegative, a minimal solution will result in exactly k components upon its removal (if there are more, we can "put back" some edges from the solution into the graph until we have exactly k components). Notice that a k-cut has more than two shores (connected component) for $k > 2$ and hence cannot be specified by a single subset of edges; Rather, we now need a partition of the vertices into multiple blocks to specify such a cut. Hence, such cuts also also termed *multicuts*.

The minimum k-cut problem is solvable in time $O(n^{k^2})$ on an n-node graph [11], while on a planar graph, the running time can be reduced to $O(n^{ck})$ [5]. However for arbitrary k (when k is part of the input specification), this problem is NP-hard [11]. We will look at an approximation algorithm for this problem due to Saran and Vazirani [26] that achieves performance ratio $2(1-\frac{1}{k})$. The algorithm they propose in fact gives an approximate solution for a minimum i-cut for every value of i from 2 to k due to a *extendible* property of the approximate solution – a near-optimal minimum i-cut can be extended to a near optimal $i+1$-cut.

10.4.2 Two simple algorithms

First we examine two simple heuristics – GREEDY and SPLIT. We then show how they are both dominated in performance and running time by a better heuristic EFFICIENT. Then, in the next section, we prove the performance of EFFICIENT. We shall use lower-case letters with subscripts (such as b_i, c_i, g_i, and s_i to denote subsets of edges in the sequel.

A greedy method

A simple greedy way to create more components is to pick the lightest (minimum capacity) cut that creates more components in the current graph. A naive method to implement this idea will look at a list of *all* cuts in G sorted by nondecreasing capacity $c_1, c_2, c_3, \ldots$, and iteratively picking the first one in the sequence that creates more components. Notice the resemblance to Kruskal's algorithm for finding minimum spanning trees.

It is however impractical to create such a list, but we can still implement this method by looking at a much more restricted list of cuts: for every edge e, we find a minimum cut s_e separating the endpoints of e, and sort this list of cuts by capacity. We run our algorithm on this smaller list. This gives us the first algorithm.

Algorithm GREEDY
input: Graph $G = (V, E)$, nonnegative edge capacities c.
output: A k-cut of capacity at most $2(1-\frac{1}{k})$ of the minimum.
STEP 1: Initialize the multicut $s \leftarrow \emptyset$, and $i \leftarrow 1$;
    for each edge e
STEP 2:   pick a minimum cut s_e separating the endpoints of e;
STEP 3: Sort these cuts in the order of nondecreasing capacity to get the list
    $s_1, s_2, \ldots, s_m$, where $m = |E|$;
    while $(V, E - s)$ has fewer than k connected components **do**
STEP 3:   **if** $s_i \not\subseteq s$, **then** $s \leftarrow s \cup s_i$;
STEP 4: $i \leftarrow i + 1$;
    output s;

Note that $E \subseteq s_1 \cup s_2 \cup \ldots \cup s_m$ and hence the algorithm will be able to find a k-cut for any value of $k \leq n$. A simplistic implementation of this algorithm may require up to m pairwise minimum cut computations. However, we can use a cut-equivalent tree to determine all the relevant cuts, leading to the algorithm EFFICIENT which is the subject of the next section.

A simple splitting method

Another simple idea to generate more components is to work in each of the current components in the graph and find the lightest cut that splits one of them. If this is repeated $k-1$ times, we get a k-cut, giving the following algorithm.

Algorithm SPLIT
input: Graph $G = (V, E)$, nonnegative edge capacities c.
output: A k-cut of capacity at most $2(1 - \frac{1}{k})$ of the minimum.
STEP 1: Initialize the multicut $s \leftarrow \emptyset$, and $i \leftarrow 1$;
    for i from 1 to $k-1$ **do**
        for each connected component G_j of the graph $(V, E - s)$
STEP 2:   pick a minimum cut s_j of G_j;
STEP 3:   Let s' be a cut with minimum capacity among the cuts s_j; Update $s \leftarrow s \cup s'$;
    output s;

Again, a simple implementation of the algorithm uses $i-1$ minimum cut computations at iteration i giving a total of k^2 minimum cut computations. The cut-equivalent tree may be used to find all the cuts s_j at each iteration leading to the algorithm EFFICIENT that we present next.

10.4.3 An efficient algorithm and analysis

Both the algorithms GREEDY and SPLIT when implemented using a cut-equivalent tree essentially boil down to picking the lightest $k-1$ cuts represented in this tree. This is due to the fact that every cut represented by the cut-equivalent tree leaves exactly two components on its removal from the graph. Thus we have the following algorithm.

Algorithm EFFICIENT
input: Graph $G = (V, E)$, nonnegative edge capacities c.
output: A k-cut of capacity at most $2(1 - \frac{1}{k})$ of the minimum.
STEP 1: Construct a cut-equivalent tree T for G where every edge $e \in T$
    is labeled with the capacity of the corresponding cut in G;
STEP 2: Sort the cuts represented by the tree in nondecreasing order of capacity to get the list
    $g_1, g_2, \ldots, g_{n-1}$;
    output $g_1 \cup g_2 \cup \ldots \cup g_{k-1}$;

Performance guarantee

The performance ratio of the above algorithm is proved by comparing it with the optimal solution, say $a \subseteq E$. Let $V_1, V_2, \ldots V_k$ denote that connected components of $(V, E - a)$. Let $a_i = \delta(V_i)$. Recall that $c(a)$ denotes the total capacity of the set of edges a. Then, we have $c(a) = \frac{1}{2}\sum_{i=1}^{k} c(a_i)$. Assume without loss of generality that $c(a_1) \leq c(a_2) \leq \ldots c(a_k)$. The performance ratio is shown by arguing that

$$\sum_{i=1}^{k-1} c(g_i) \leq \sum_{i=1}^{k-1} c(a_i) \leq 2(1 - \frac{1}{k})c(a)$$

To show this, consider the partition of the node set V into blocks $V_1, V_2, \ldots V_k$ by the optimal k-cut a. The number of distinct edges of the tree T that crosses this partition is at least $k-1$ since T spans V. Let these edges of T be $e_1 = u_1v_1, e_2 = u_2v_2, \ldots e_p = u_pv_p$ where $p \geq k-1$. Consider an auxiliary graph $G(a)$ whose vertices are $v_1, v_2, \ldots v_k$ representing the contracted versions of the node sets $V_1, V_2, \ldots V_k$ respectively and whose edges are $e_1, \ldots, e_p$. Note that $G(a)$ may have parallel edges and is connected. Let $T(a)$ be a directed spanning tree of $G(a)$ rooted at the node v_k, with all edges directed towards the root.

Suppose the edge $e = uv$ is directed from the node v_i to v_j in $T(a)$. Then, $u \in V_i$ and $v \in V_j$ in G. Moreover $c(e)$ represents the minimum capacity of any cut separating u and v in G. But $a_i = \delta(V_i)$ is one such cut and so $c(e) \leq c(a_i)$. This charging argument charges the $k-1$ different edges e in the tree $T(a)$ to the cuts $a_1, \ldots a_{k-1}$. Note that v_k is the root and therefore a_k is not charged. Thus the capacity of the cuts $a_1, \ldots a_{k-1}$ is at least as much as the $k-1$ edges from the cut-equivalent tree that appear in $T(a)$. This in turn is at least as much as the $k-1$ lightest edges in the cut-equivalent tree $g_1, \ldots g_{k-1}$ that was picked by the algorithm EFFICIENT completing the proof.

10.5 Multiway Cuts

The multiway cut problem is to find a minimum capacity set of edges or nodes whose removal puts a given set of terminals $s_1, s_2, \ldots, s_k$ in an undirected graph in different connected components. We will examine an approximation algorithm for this problem based on a relaxation of an integer programming formulation of the problem.

10.5.1 Introduction

Let $G = (V, E)$ be a connected undirected graph, and let every edge $ij \in E$ be assigned a nonnegative capacity $c(ij)$. Assume also that every node v is assigned a capacity c_v, and a set of k *terminal* nodes $t_1, t_2, \ldots t_k$ are specified. The multiway edge/node cut problem is to find a minimum capacity set of edges/nodes such that after the removal of these edges/nodes, no pair of terminals are in the same connected component.

A minimal solution to a multiway edge cut problem with k terminals is a k-cut, namely, a set of edges that leaves at least k components, one for each terminal. While the minimum k-cut problem is polynomial-time solvable for fixed k, the multiway edge cut problem is NP-hard even for $k = 3$ [5].

Note that the multiway node cut problem is more general than the edge version, since the edge version can be transformed to an instance of the node version as follows: subdivide every edge to introduce a new node whose capacity equals the capacity of the edge it represents. All the resulting half-edges may be assumed to have capacity zero. In fact, this shows that if we solve the node version of the problem we can even model the version where both nodes and edges have capacities and we are allowed to find a mixed cut containing both nodes and edges to disconnect the terminals. In what follows, we will therefore focus only on the node-capacitated version and refer to this as the multiway cut problem.

We will examine an approximation algorithm with performance ratio $2(1-\frac{1}{k})$, same as for the k-cut problem, for the multiway node cut problem with k terminals. This algorithm, due to Garg, Vazirani and Yannakakis [8], uses an integer programming (IP) formulation of the problem and shows that the linear programming (LP) relaxation of the formulation has an optimum solution all of whose components are half-integers. Their proof also gives a method to find an approximate multiway cut from any optimal linear programming solution. In the next section, we present the IP formulation and derive it's LP dual. In the following section we show that the LP has a half-integral solution and derive the approximation algorithm.

10.5.2 IP formulation of multiway cuts

A direct formulation uses a 0-1 choice variable x_v to indicate if node v is chosen in the cut. How can we enforce that the set of nodes chosen separate a pair of terminals? A simple (but costly) method is to simply insist that *every* path between this pair of terminals contains at least one chosen node. This gives the following formulation. We use T to denote the set of terminal nodes.

$$\begin{array}{llrcll}
\text{(IP)} & \text{minimize} & \sum_{v \in V \setminus T} c_v x_v & & & \\
& \text{subject to} & \sum_{v \in p_{ij}^k} x_v & \geq & 1 & \forall\ t_i - t_j \text{ paths } p_{ij}^k \\
& & x_v & \in & \{0,1\}, & \forall v \in V \setminus T
\end{array}$$

In the formulation, p_{ij}^k represents the k^{th} distinct path between the pair of terminals t_i and t_j. The number of constraints in this formulation is enormous on account of this. Furthermore, since IP is NP-hard as well, we shall only be interested in solving the LP relaxation of this IP.

$$\begin{array}{llrcll}
\text{(LP)} & \text{minimize} & \sum_{v \in V \setminus T} c_v x_v & & & \\
& \text{subject to} & \sum_{v \in p_{ij}^k} x_v & \geq & 1 & \forall\ t_i - t_j \text{ paths } p_{ij}^k \\
& & x_v & \geq & 0, & \forall v \in V \setminus T
\end{array}$$

Compact reformulation

Due to the enormous number of constraints in this LP, it is not clear at first sight if this LP can be solved in polynomial time. However, it is not hard to show that the ellipsoid method can be used to accomplish this. To do this, the key requirement is a polynomial-time *separation oracle* (for more details on why the separation oracle is sufficient for optimization, see [13]). Given a candidate solution to the LP, namely, a set of values x_v for $v \in V\backslash T$, the oracle must determine if all the constraints of (LP) are satisfied by this solution, and if not, find a constraint that is violated (not satisfied) by the solution. This is not hard for the given set of constraints – let us design a separation oracle for enforcing the constraints involving a specific pair of terminals t_i and t_j. Suppose the candidate solution violates some constraint for this pair. Then there is a path between them with total x-value less than one. This implies also that the smallest node-weighted path between this pair using the x-values as node-weights is less than one. Since the x-values are nonnegative, we can determine the value of such a smallest node-weighted path by a straightforward modification of a shortest path algorithm such as Dijkstra's method. This gives us the separation oracle.

The idea for the separation oracle also allows us to come up with a more compact reformulation of (LP). We will sketch how to reformulate the set of inequalities enforcing the requirement or a pair of terminals t_i and t_j using shortest paths. To model the smallest node-weight path between t_i and t_j, we use a distance label $d_{ij}(v)$ at each node v. We set $d_{ij}(t_i)$ to be zero and $d_{ij}(t_k)$ for $k \neq i, j$ to be ∞. The intent is to enforce that $d_{ij}(t_j)$ represents the weight of the smallest node weight of path from t_i to t_j. So we enforce that for every node $v \in V\backslash T$, the label $d_{ij}(v)$ represents the smallest weight of a path from t_i to the node v. This we model as for every node $v \in V\backslash T$, $d_{ij}(v) \leq d_{ij}(u) + x_v$ for every neighbor u of v. This alone is not sufficient to enforce that d_{ij} denote node-distances, since for instance, setting all of these variables to zero is still legitimate. To complete the implement of our separation idea, we also add the inequality that $d_{ij}(t_j) \geq 1$. This "tests" if the d_{ij} values can be adjusted to denote weights of the smallest paths from t_i such that this value is at least 1 for t_j. Note that we have only added about as many inequalities as the number of edges of G and as many new variables as the number of nodes in G. We can repeat this for every pair of terminals to get a full compact formulation. It is easy to see that this compact formulation is equivalent to (LP) and therefore has the same solutions. We will continue to work with (LP) due to its simplicity, but remember that we can either use the ellipsoid algorithm or this compact formulation to find a solution in polynomial time.

A dual linear program

Next we consider the dual linear program of (LP). Recall that we have one variable in the dual for every constraint of (LP), thus a variable f_{ij}^k corresponding to constraint for the path p_{ij}^k. The coefficients of these variables in the objective is the right-hand side of (LP), namely, all ones. Finally, we have one constraint in the dual program for every variable in the primal, thus we have a constrain corresponding to every node $v \in V\backslash T$. The dual program is given below.

$$
\begin{array}{llrcll}
\text{(DP)} & \text{maximize} & \sum_{i,j,k} f_{ij}^k & & & \\
 & \text{subject to} & \sum_{i,j,k:\ v \in p_{ij}^k} f_{ij}^k & \leq & c_v & \forall \text{ nodes } v \in V\backslash T \\
 & & f_{ij}^k & \geq & 0, & \forall\ t_i - t_j \text{ paths } p_{ij}^k
\end{array}
$$

We can interpret the nonnegative value f_{ij}^k associated with the path p_{ij}^k as a flow along this path from t_i to t_j (the direction is immaterial). The constraint corresponding too a node $v \in V \backslash T$ then specifies that the total of all such flows going through v must not exceed its capacity c_v. The objective of (DP) is to maximize the total such flow that can be routed. In short, the LP-dual of the minimum multiway cut problem is to maximize the total flow that can sent between all the pairs of terminals simultaneously (sometimes also called concurrent flow) under node capacities c_v.

As a quick check, let us consider the case when $|T| = 2$. This corresponds to finding a minimum capacity set of nodes that separate a pair of given nodes, namely, a minimum node cut between the pair. The dual of this problem, as in the max-flow min-cut theorem, is to maximize the total flow that can be sent between the pair under the node capacities in the graph. This is exactly what our dual interpretation boils down to in this case.

Complementary slackness

We can use our linear formulations to get a solution to the relaxation of the multiway cut problem. In fact, we can use linear programming to derive have a pair of optimal solutions for the pair of programs (LP) and (DP). The theory of linear programming (see, e.g., [4]) specifies that the pair of solutions obey two complementary slackness conditions; In general, for a pair of optimal solutions to two dual linear programs, this condition states that if a variable is non-zero in one program, then the constrain corresponding to this variable in the dual program must be satisfied with equality (must be tight). Applying this condition to the variables in (LP) and (DP) gives the following.

1. Any node $v \in V \backslash T$ with a non-zero value of x_v in (LP) is saturated, i.e., the total flow through this vertex in (DP) is exactly its capacity c_v.
2. Any flow path f_{ij}^k that carries non-zero flow must have node-distance (sum of x-values from (LP)) exactly one.

These conditions will be useful later.

Conditions for optimality

Before we investigate further structure of the solution to (LP), we recall another useful fact from linear programming - a pair of solutions to two dual linear programs are both optimal if both are feasible for their respective programs and both have the same objective value. This is a simple consequence of weak duality. In our case, weak duality implies that any primal solution to (LP) has objective value at least as much as that of any solution (DP) as shown below. Here let x and f denote feasible solutions to (LP) and (DP) respectively.

$$\begin{aligned}
\sum_{v \in V \backslash T} c_v x_v &\geq \sum_{v \in V \backslash T} x_v \sum_{i,j,k: v \in p_{ij}^k} f_{ij}^k \\
&= \sum_{i,j,k} f_{ij}^k \sum_{v \in p_{ij}^k} x_v \\
&\geq \sum_{i,j,k} f_{ij}^k \cdot 1
\end{aligned}$$

Thus if the pair x and f achieve the same objective value, then each has achieved its extreme value.

10.5.3 A half-integrality property

In this section, we will show that for any specification of terminals and for any capacity values on the nodes, there is always an optimal solution to (LP) that is half-integral. Assume that x and f represent a pair of dual optimal solutions in the following.

Other half-integral LPs

An immediate application of half-integrality will be to round up all the variables of (LP) that are non-zero. Since the constraint matrix has no negative co-efficient, it is easy to check that this maintains feasibility of the solution. Furthermore, since any variable whose value increases to one must have been set to a half to start with, we scale the linear solution by a factor of at most two to convert it to a rounded integer solution.

A similar simple rounding method of rounding is also well known for the vertex cover problem, and the half-integrality of the corresponding linear program is also known [24]. The most general case currently know for which the linear program has half-integral solutions is one where the constraint matrix has at most two non-zero entries in each column that are +1 or -1 [17].

A decomposition

Given an optimal solution x to (LP), we think of these as node-lengths as in the formulation. For each terminal t_i, we can then identify a subset of nodes T_i, all of distance zero from t_i under this length. In other words, T_i defines a region of the graph containing all nodes that can be reached via paths involving only zero-length nodes from t_i. We must address the issue of the node-length of any terminal for this definition. We can assume that this is zero for every terminal, and check that this is consistent with our definition of the length of a path in the formulation, even if we include the either or both endpoints of the path in computing the length of the path. Note that by the feasibility of (LP), the regions $T - i$ corresponding to different terminals t_i are disjoint.

Next we define the boundary of T_i denoted C_i as the set of nodes in $V \backslash T_i$ that are neighbors of some node in T_i. If a node v belongs to two different boundaries C_i and C_j, then $x_v = 1$ due to the distance constraint between t_i and t_j – there is a path between these nodes in which only v has non-zero node-length. Define $C = \cup_{i=1}^{k} C_i$ and $C^1 \subseteq C$ as the set of nodes in at least two different boundaries, and therefore set to node-length one. Let $C' = C \backslash C^1$.

Lemma 10.16 *Let f_{ij}^k be set to a non-zero value in (DP). Then the path p_{ij}^k contains exactly one node from C^1 or exactly two nodes from C'.*

Proof: Let the first and last nodes along p_{ij}^k in C_i and C_j be v_i and v_j respectively.

(i) $v_i = v_j$.
In this case, this node is in C^1 and we have exactly this node from C^1 and no other node of C in the path.

(ii) $v_i \neq v_j$.
Suppose that either v_i or v_j is in C^1. Then the total length of p_{ij}^k is greater than one, contradicting the second complementary slackness condition. Thus, both these nodes are in C'.

Next suppose that there is another node $v \in C'$ distinct from v_i and v_j in the path, and let $v \in C_k$. In this case, we can find a path between t_k and either of t_i or t_j with node length less than one contradicting the feasibility of x. To see this, consider the prefix of the path p_{ij}^k from t_i to node v, and extend this to a path ending in t_k via nodes in C_k. then length of this path is strictly less than that of p_{ij}^k since we do not have node v_j that has non-zero length and the extension has zero length. Finally, the length of path p_{ij}^k is exactly one from the second complementary slackness condition, giving that the new path has length less than one, a contradiction.

▌

Half-integrality

Theorem 10.17 *There is an optimal solution to (LP) that is half-integral.*

Proof: The half-integral solution we propose can be constructed from *any* given optimal solution x. Using these as node-distances, we compute C^1 and C'. Next we consider a different solution x^* to (LP) as follows.

$$x_v^* = \begin{cases} 1 & \forall v \in C^1 \\ \frac{1}{2} & \forall v \in C' \\ 0 & \text{otherwise} \end{cases}$$

We show next that x^* is feasible and optimal for (LP).

Any path from t_i and t_j must use nodes from both C_i and C_j, not necessarily distinct. Let v_i and v_j the corresponding nodes. If these are the same, then this node must be in ^{1} and the length of this path is one. Otherwise, the path uses at least two distinct nodes from C' and hence the length is at least one again. This shows that x^* is feasible.

To show optimality, we show that its value is equal to that of the dual objective associated with x. The value of the dual is the total concurrent flow between the terminals. From lemma 10.16, every flow path carrying non-zero flow crosses either one node in C^1 exactly once or exactly two distinct nodes in C'. Thus the total value of all the flow can be accounted as $\sum_{v \in C^1} c_v + \frac{1}{2}\sum_{v \in C'} c_v$. This is exactly the value of the solution x^*. hence x^* is optimal. ▌

An approximation algorithm

We present an algorithm that exploits the half-integrality property to find a rounded solution.

Algorithm MULTIWAY CUT
input: Graph $G = (V, E)$, nonnegative node capacities c,
and a node subset T of terminals with $|T| = k$.
output: A multiway cut for T of capacity at most $2(1 - \frac{1}{k})$ of the minimum.
STEP 1: compute an optimal solution x to (LP);
STEP 2: **for** every terminal $t_i \in T$, compute the set of nodes T_i
reachable from t_i via paths with zero x-value, and C_i, the node neighbors of T_i;

Set $C \leftarrow \cup_i C_i$;
STEP 3: **for** every i, compute C_i', the set of nodes
in C_i that are not in any other C_j for $j \neq i$;
STEP 3: find C_{max}', the set of maximum node capacity among the C_i''s;
output $C - C_{max}'$;

It is clear that the algorithm can be implemented in polynomial time by using a node-weighted version of Dijkstra's algorithm. It is not hard to show feasibility of the output solution as well. Any path between two terminals has to either contain one node in C^1 or at least two nodes in C_i's. Since we only exclude one such set C_i' in forming the final solution, every path must cross at least one node from the output set showing feasibility.

Performance guarantee

The performance ratio of the algorithm follows almost directly from Theorem 10.17. Define $C' = C - C^1$ where C and C^1 are computed by the algorithm. By a simple averaging argument, it follows that $c(C_{max}') \geq \frac{1}{k} c(C')$ where c denotes the capacity of the set.

$$\begin{aligned} c(C - C_{max}') &= c(C) - c(C_{max}') \\ &= c(C^1) + \sum_{i=1}^{k} c(C_i') - c(C_{max}') \\ &\leq c(C^1) + (1 - \frac{1}{k}) \sum_{i=1}^{k} c(C_i') \\ &\leq 2(1 - \frac{1}{k})(c(C^1) + \frac{1}{2} \sum_{i=1}^{k} c(C_i')) \end{aligned}$$

By the theorem, the right hand side of the last inequality is at most $2(1 - \frac{1}{k})$ times the value of an optimal solution to (LP). This is a value of a relaxation of the multiway cut problem and hence is a lower bound on the capacity of the minimum multiway cut, completing the proof.

Approximate min-max relation

Note that our proof generalizes the classical max-flow min-cut theorem for the node-capacitated case in a nice way. When $k = 2$, the performance ratio is one giving us that the cut we find by the rounding algorithm is a minimum cut and our results show that the value of a maximum node-capacitated flow is equal to the value of the minimum cut; this is the max-flow min-cut theorem.

For larger k, we continue to have an exact relation between the flow and cut values, if we allow both of them to be fractional. This is what linear programming duality implies for the pair of programs (LP) and (DP). However, if we strengthen our notion of a multiway cut to mean an integral solution to (LP), the theorem guarantees that we can achieve a total (fractional) flow of value at least a fraction half of the minimum such integral cut.

Recent Improvement

For the edge-capacitated case of the multiway cut problem described above, Călinescu et al. [2] have recently devised a $\frac{3}{2}$-approximation algorithm via a novel randomized rounding of a strengthened linear programming relaxation for the problem.

10.6 Exercises

1. A legal ordering of the nodes of G identifies a pair of vertices, v_{n-1} and v_n such that the cut around the singleton set $\{v_n\}$ is a minimum cut separating v_{n-1} and v_n. Show that if G has at least three vertices, there is a second pair of vertices with this property.
2. Does the framework of the node identification algorithm work for finding minimum node-cuts in an undirected graph? Why or why not? Answer the same question for finding a node-cut separating some pair of vertices using legal orderings.
3. Does the upper bound of $\frac{n(n-1)}{2}$ on the number of min cuts of an n-node graph that we proved for undirected graphs also hold for directed graphs? Why or why not? Also, is this bound tight for undirected graphs? (read as, are there capacitated undirected graphs which have as many as $\frac{n(n-1)}{2}$ min cuts?)
4. For any constant positive half-integer $\alpha \geq 1$, show that the number of α-minimum cuts in an undirected graph on n nodes is at most $O(n^{2\alpha})$. (An α-min cut is one whose capacity is at most α times that of a minimum cut).
5. Is there a compact representation of all the $n(n-1)$ minimum cuts in a directed graph? (The capacity of a cut separating nodes x and y is defined the sum of the capacities of all the edges coming out of the component containing x.) For instance, could you hope to represent all of the by a tree with only $n-1$ values. What is the maximum number of distinct values of minimum cuts you can find in a directed graph?
6. We saw in class that a cut-equivalent tree is also a flow-equivalent tree. Show that the reverse implication does not always hold.
7. Cheng and Hu showed that flow-equivalent trees exist even when the capacity of a cut is defined by an arbitrary general function, i.e., a function $f : S \Rightarrow R^+$ that assigns a nonnegative value to every subset $S \subset V$ of vertices of the given undirected graph. Prove the Cheng-Hu result that a flow-equivalent tree *exists* in this case.

   Extra credit: Assume that you have an oracle for determining a minimum cut separating a pair of nodes. Devise a method to construct such a flow-equivalent tree using at most $n-1$ calls to the oracle.
8. Show that every graph has a flow-equivalent tree that is a (Hamiltonian) path.
9. Give an example to show that the integrality gap of 2 is tight for the node-multiway cut problem. Do the same for the edge-multiway cut case.
10. Is the analysis of the performance guarantee for the k-cut problem tight? Show an example to illustrate this.

11. Does the hereditary property of approximate solutions for the k-cut problem carry over to optimal solutions? In other words, note that the first $k' - 1$ cuts among the $k - 1$ chosen as a 2-approximate solution to the k0cut problem, form a 2-approximate solution to the k' cut problem (for $k' < k$). Is it true that any optimal solution for such a k'-cut problem can be extended to an optimum solution for the k-cut problem?
12. The Optimal Tree Arrangement problem (OTA) is defined on an unweighted undirected graph G as follows. The goal is to find a spanning tree T on the vertices of the graph minimizing the sum over all the edges of G, of the distance in T between the endpoints of the edge. Give a polynomial time algorithm for the OTA problem.
13. The Multicommodity Flow Tree problem (MFT) is defined on an undirected graph G with nonnegative capacities c on the edges as follows. The goal is to find a spanning tree T with nonnegative capacities assigned to its edges such that for every pair of vertices s, t in the graph, the flow between s and t in the tree T is at least the maximum $s - t$ flow in the graph G. In this way, the tree T supports all the flows that G does. The objective is to minimize the the maximum capacity of any edge in T. Give a polynomial time solution to the MFT problem.
14. If all cuts in a cut-equivalent tree of a connected undirected graph are even, is the graph Eulerian (i.e., have even degree at all the nodes)?
15. (Extra credit) In showing a 2-approximation algorithm for node-multiway cuts, we showed half-integrality of optimal solutions for a certain linear relaxation of the problem. The dual to this relaxation is a linear program whose objective is to maximize the total concurrent flow between terminals specified in the problem. Prove or disprove: for any specification of integral capacities, there is a maximum flow solution that is also half-integral.

Bibliography

[1] A. Benczúr, *Counterexamples for directed and node capacitated cut-trees*, SIAM J. Comput. **24:3** (1995), pp. 505-510.

[2] G. Călinescu, H. Karloff, and Y. Rabani, *An improved approximation algorithm for Multiway Cut*, Proc. STOC '98 (1998), 48-52.

[3] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein, *Experimental study of minimum cut algorithms*, NECI TR 96-132 (October 1996). Available from `http://www.neci.nj.nec.com/homepages/avg/webpub/webpub.html`.

[4] V. Chvátal, *Linear Programming*, (1983) W. H. Freeman and Company.

[5] E. Dalhaus, D. S. Johnson, C. H. Papadimitriou, P. Seymour, and M. Yannakakis, *The complexity of multiway cuts*, Proc. 24th Annual ACM Symp. on Theory of Comput. (1992), pp. 241-251.

[6] P. Elias, A. Feinstein and C. E. Shannon, *Note on maximum flow through a network*, IRE Trans. Inform. Theory **IT-2** (1956), pp. 117-119.

[7] L. R. Ford and D. R. Fulkerson, *Maximal flow through a network*, canad. J. Math. **8** (1956), pp. 399-404.

[8] A. Frank, T. Ibaraki, and H. Nagamochi, *On Sparse Subgraphs Preserving Connectivity Properties*, J. Graph Theory **17:3** (1993), pp. 275-281.

[9] N. Garg, V. Vazirani and M. Yannakakis, *Multiway cuts in directed and node-weighted graphs*, Proc. ICALP '94 (1994), pp. 487-498.

[10] A. V. Goldberg and R. E. Tarjan, *A New Approach to the Maximum Flow Problem*, J. Assoc. Comput. Mach. **35** (1988), pp. 921-940.

[11] O. Goldschmidt and D. S. Hochbaum, *Polynomial algorithm for the k-cut problem*, Proc. 29th Annual Symp. on Foundations of Comp. Sci. (1988), pp. 444-451.

[12] R. Gomory and T. C. Hu, *Multi-terminal network flows*, J. SIAM **9** (1961), pp. 551-570.

[13] M. Grötschel, L. Lovász and A. Schrijver, *Geometric algorithms and combinatorial optimization*, (1988) Springer-Verlag.

[14] D. Gusfield, *Very simple methods for all pairs network flow analysis*, SIAM J. Comput. **19:1** (1990), pp. 143-155.

[15] J. Hao and J. B. Orlin, *A Faster Algorithm for Finding the Minimum Cut in a Directed Graph*, J. Algorithms **17** (1994), pp. 424-446.

[16] D. Hartvigsen, and F. Margot, *Multiterminal flows and cuts*, Oper. Res. Lett. **17** (1995), pp.201-204.

[17] D. S. Hochbaum, N. Meggido, J. Naor and A. Tamir, *Tight bounds and 2-approximations for integer programs with two variables per inequality*, Math. Programming **62** (1993), pp. 69-83.

[18] D. Karger and C. Stein, *An $\tilde{O}(n^2)$ algorithm for finding minimum cuts*, Proc. 25th Annual ACM Symp. on Theory of Computing (1993), pp. 757-765.

[19] V. King, S. Rao, and R. E. Tarjan, *A Faster Deterministic Maximum Flow Algorithm*, J. Algorithms **17** (1994), pp. 447-474.

[20] W. Mader, *Über minimal n-fach zusammenhängende unendliche Graphen und ein Extremal problem*, Arch. Mat. **23** (1972), pp. 553-560.

[21] K. Menger, *Zur allgemeinen Kurventheorie*, Fund. Math. **10** (1927), pp. 96-115.

[22] H. Nagamochi and T. Ibaraki, *Computing Edge-Connectivity in Multigraphs and Capacitated Graphs*, SIAM J. Disc. Meth., **5** (1992), pp. 54-66.

[23] H. Nagamochi and T. Ibaraki, *A Linear Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph*, Algorithmica **7** (1992), pp. 583-596.

[24] G. Nemhauser and L. Trotter, *Vertex packings: Structural properties and algorithms*, Math. Programming **8** (1975), pp. 232-248.

[25] T. Nishizeki and S. Poljak, *k-Connectivity and Decompositions of Graphs into Forests*, Disc. Appl. Math. **55:3** (1994), pp. 295.

[26] H. Saran and V. Vazirani, *Finding k-cuts within twice the optimal*, SIAM J. on Computing **24:1** (1995), pp. 101-108. Extended abstract appeared in Proc. 32nd Annual Symp. on Foundations of Comp. Sci. (1991), pp. 743-751.

[27] M. Stoer and F. Wagner, *A Simple Min Cut Algorithm*, Algorithms - ESA '94 **LNCS 855** (1994), pp. 141-147.