Massachusetts Institute of Technology
18.433: Combinatorial Optimization
Michel X. Goemans

Handout 18
April 27th, 2009

6. Lecture notes on flows and cuts

6.1 Maximum Flows

Network flows deals with modelling the flow of a commodity (water, electricity, packets, gas, cars, trains, money, or any abstract object) in a network. The links in the network are capacitated and the commodity does not vanish in the network except at specified locations where we can either inject or extract some amount of commodity. The main question is how much can be sent in this network.

Here is a more formal definition of the maximum flow problem. We have a digraph (directed graph) $G = (V, E)$ and two special vertices s and t; s is called the source and t the sink. We have an upper capacity function $u : E \to \mathbb{R}$ and also a lower capacity function $l : E \to \mathbb{R}$ (sometimes chosen to be 0 everywhere). A flow x will be an assignment of values to the arcs (directed edges) so that:

1. for every $e \in E$: $l(e) \leq x_e \leq u(e)$,
2. for every $v \in V \setminus \{s, t\}$:
$$\sum_{e \in \delta^+(u)} x_e - \sum_{e \in \delta^-(u)} x_e = 0. \tag{1}$$

The notation $\delta^+(u)$ represents the set of arcs *leaving* u, while $\delta^-(u)$ represents the set of arcs *entering* u.

Equations (1) are called *flow conservation* constraints. Given a flow x, its *flow value* $|x|$ is the net flow out of s:
$$|x| := \sum_{e \in \delta^+(s)} x_e - \sum_{e \in \delta^-(s)} x_e. \tag{2}$$

One important observation is that $|x|$ is also equal to the net flow into t, or minus the net flow out of t. Indeed, summing (1) over $u \in V \setminus \{s, t\}$ together with (2), we get:

$$\begin{aligned} |x| &= \sum_{v \in V \setminus \{t\}} \left(\sum_{e \in \delta^+(u)} x_e - \sum_{e \in \delta^-(u)} x_e \right) \\ &= \sum_{e \in \delta^-(t)} x_e - \sum_{e \in \delta^+(t)} x_e \end{aligned}$$

by looking at the contribution of every arc in the first summation.

The *maximum flow problem* is the problem of finding a flow x of maximum value $|x|$. This is a linear program:

$$\begin{aligned} \text{Max} \quad & \sum_{e\in\delta^+(s)} x_e - \sum_{e\in\delta^-(s)} x_e \\ \text{subject to:} \quad & \\ & \sum_{e\in\delta^+(u)} x_e - \sum_{e\in\delta^-(u)} x_e = 0 && u \in V\setminus\{s,t\} \\ & l(e) \le x_e \le u(e) && e \in E. \end{aligned}$$

We could therefore use algorithms for linear programming to find the maximum flow and duality to derive optimality properties, but we will show that more combinatorial algorithms can be developed and duality translates into statements about *cuts*.

In matrix form, the linear program can be written as:

$$\max\{c^T x : \quad \begin{array}{rcl} Nx & = & 0, \\ Ix & \le & u, \\ -Ix & \le & -l\} \end{array}$$

where N is the (vertex-arc incidence[1]) matrix with rows indexed by $u \in V\setminus\{s,t\}$ and columns indexed by arcs $e=(i,j)\in E$; the entry N_{ue} is:

$$N_{ue} = \begin{cases} 1 & u = i \\ -1 & u = j \\ 0 & u \notin \{i,j\}. \end{cases}$$

The constraints of the linear program are thus: $Ax \lesseqgtr b$ where

$$A = \begin{pmatrix} N \\ --- \\ I \\ --- \\ -I \end{pmatrix},$$

and some of the constraints are equalities and some are inequalities.

Lemma 6.1 *A is total unimodular.*

Proof: We could use Theorem 3.14 from the polyhedral chapter, but proving it directly is as easy. Consider any square submatrix of A, and we would like to compute its determinant up to its sign. If there is a row with a single $+1$ or a single -1 in it, we can expand the determinant and compute the determinant (up to its sign) of a smaller submatrix of A.

[1]More precisely, part of it as we are not considering vertices s and t

Repeating this, we now have a square submatrix of N. If there is a column with a single $+1$ or a single -1 then we can expand the determinant along this column and get a smaller submatrix. We are thus left either with an empty submatrix in which case the determinant of the original matrix was $+1$ or -1, or with a square submatrix of N with precisely one $+1$ and one -1 in *every* column. The rows of this submatrix are linearly dependent since their sum is the 0 vector. Thus the determinant is 0. This proves total unimodularity. △

As a corollary, this means that if the right-hand-side (i.e. the upper and lower capacities) are integer-valued then there always exists a maximum flow which takes only integer values.

Corollary 6.2 *If $l : E \to \mathbb{Z}$ and $u : E \to \mathbb{Z}$ then there exists a maximum flow x such that $x_e \in \mathbb{Z}$ for all $e \in E$.*

6.1.1 Special cases

Arc-disjoint paths. If $l(e) = 0$ for all $e \in E$ and $u(e) = 1$ for all $e \in E$, any integer flow x will only take values in $\{0, 1\}$. We claim that for an integer flow x, there exist $|x|$ arc-dsjoint (i.e. not having any arcs in common) paths from s to t. Indeed, such paths can be obtained by *flow decomposition*. As long as $|x| > 0$, take an arc out of s with $x_e = 1$. Now follow this arc and whenever we reach a vertex $u \neq t$, by flow conservation we know that there exists an arc leaving u that we haven't traversed yet (this is true even if we reach s again). This process stops when we reach t and we have therefore identified one path from s to t. Removing this path gives us a new flow x' (indeed flow conservation at vertices $\neq s, t$ is maintained) with $|x'| = |x| - 1$. Repeating this process gives us $|x|$ paths from s to t and, by construction, they are arc-disjoint. The paths we get might not be *simple*[2]; one can however make them simple by removing the part of the walk between repeated occurences of the same vertex. Summarizing, if $l(e) = 0$ for all $e \in E$ and $u(e) = 1$ for all $e \in E$, then from a maximum flow of value k, we can extract k arc-disjoint (simple) paths from s to t. Conversely, if the digraph contains k arc-disjoint paths from s to t, it is easy to construct a flow of value k. This means that the maximum flow value from s to t represents the maximum number of arc-disjoint paths between s and t.

Bipartite matchings. One can formulate the maximum matching problem in a bipartite graph as a maximum flow problem. Indeed, consider a bipartite graph $G = (V, E)$ with bipartition $V = A \cup B$. Consider now a directed graph D with vertex set $V \cup \{s, t\}$. In D, there is an arc from s to every vertex of A with $l(e) = 0$ and $u(e) = 1$. There is also an arc from every vertex in B to t with capacities $l(e) = 0$ and $u(e) = 1$. Every edge $(a, b) \in E$ is oriented from $a \in A$ to $b \in B$ and gets a lower capacity of 0 and an upper capacity equal to $+\infty$ (or just 1). One can easily see that from any matching of size k one can construct a flow of value k; similarly to any *integer valued* flow of value k corresponds a matching of size k. Since the capacities are in $\mathbb{Z}$, by Corollary 6.2, this means that a maximum flow in

[2]A simple path is one in which no vertex is repeated.

D has the same value as the maximum size of any matching in G. Observe that the upper capacities for the arcs between A and B do not matter, provided they are ≥ 1.

Orientations. In the chapter on matroid intersection, we considered the problem of orienting the edges of an undirected graph $G = (V, E)$ so that the indegree of any vertex v in the resulting digraph is at most $k(v)$. This can be formulated as a maximum flow problem in which we have (i) a vertex for every vertex of G, (ii) a vertex for every edge of G and (iii) 2 additional vertices s and t. Details are left as an exercise.

6.2 Cuts

In this section, we derive an important duality result for the maximum flow problem, and as usual, this takes the form of a minmax relation.

In a digraph $G = (V, A)$, we define a *cutset* or more simply a *cut* as the set of arcs $\delta^+(S) = \{(u, v) \in A : u \in S, v \in V \setminus S\}$. Observe that our earlier notation $\delta^+(v)$ for $v \in V$ rather than $\delta^+(\{v\})$ is a slight abuse of notation. Similarly, we define $\delta^-(S)$ as $\delta^+(V \setminus S)$, i.e. the arcs entering the vertex set S. We will typically identify a cutset $\delta^+(S)$ with the corresponding vertex set S. We say that a cut $\delta^+(S)$ is *an $s - t$ cut* (where s and t are vertices) if $s \in S$ and $t \notin S$.

For an undirected graph $G = (V, E)$, $\delta^+(S)$ and $\delta^-(S)$ are identical and will be denoted by $\delta(S) = \{(u, v) \in E : |\{u, v\} \cap S| = 1\}$. Observe that now $\delta(S) = \delta(V \setminus S)$.

For a maximum flow instance on a digraph $G = (V, E)$ and upper and lower capacity functions u and l, we define the capacity $C(S)$ of the cut induced by S as

$$C(S) = \sum_{e \in \delta^+(S)} u(e) - \sum_{e \in \delta^-(S)} l(e) = u(\delta^+(S)) - l(\delta^-(S)).$$

By definition of a flow x, we have that

$$C(S) \geq \sum_{e \in \delta^+(S)} x_e - \sum_{e \in \delta^-(S)} x_e.$$

We have shown earlier that the net flow out of s is equal to the net flow into t. Similarly, we can show that for any S with $s \in S$ and $t \notin S$ (i.e. the cut induced is an $s - t$ cut), we have that the flow value $|x|$ equals:

$$|x| = \sum_{e \in \delta^+(S)} x_e - \sum_{e \in \delta^-(S)} x_e.$$

This is shown by summing (1) over $u \in S \setminus \{s\}$ together with (2). Thus, we get that for any S with $s \in S$ and $t \notin S$ and any $s - t$ flow x, we have:

$$|x| \leq C(S).$$

Therefore, maximizing over the $s-t$ flows and minimizing over the $s-t$ cuts, we get

$$\max |x| \leq \min_{S: s \in S, t \notin S} C(S).$$

This is weak duality, but in fact, one always has equality as stated in the following theorem. Of course, we need the assumption that the maximum flow problem is feasible. For example if there is an edge with $l(e) > u(e)$ then no flow exists (we will show later that a necessary and sufficient condition for the existence of a flow is that (i) $l(e) \leq u(e)$ for every $e \in E$ and (ii) for any $S \subset V$ with $|S \cap \{s,t\}| \neq 1$, we have $u(\delta^+(S)) \geq l(\delta^-(S))$).

Theorem 6.3 (max $s-t$ flow-min $s-t$ cut) *For any maximum flow problem for which a feasible flow exists, we have that that the maximum $s-t$ flow value is equal to the minimum capacity of any $s-t$ cut:*

$$\max_{\textit{flow } x} |x| = \min_{S: s \in S, t \notin S} C(S).$$

One way to prove this theorem is by using strong duality of linear programming and show that from any optimum dual solution one can derive an $s-t$ cut of that capacity. Another way, and this is the way we pursue, is to develop an algorithm to find a maximum flow and show that when it terminates we have also a cut whose capacity is equal to the flow we have constructed, therefore proving optimality of the flow and equality in the minmax relation.

Here is an algorithm for finding a maximum flow. Let us assume that we are given a feasible flow x (if $u(e) \geq 0$ and $l(e) \leq 0$ for all e, we could start with $x = 0$). Given a flow x, we define a *residual graph* G_x on the same vertex set V. In G_x, we have an arc (i,j) if (i) $(i,j) \in E$ and $x_{ij} < u((i,j))$ or if (ii) $(j,i) \in E$ and $x_{ji} > l((j,i))$. In case (i), we say that (i,j) is a *forward* arc and in case (ii) it is a *backward* arc. If both (i) and (ii) happen, we introduce two arcs (i,j), one forward and one backward; to be precise, G_x is thus a multigraph. Consider now any directed path P from s to t in the residual graph; such a path is called an *augmenting path*. Let P^+ denote the forward arcs in P, and P^- the backward arcs. We can modify the flow x in the following way:

$$x'_e = \begin{cases} x_e + \epsilon & e \in P^+ \\ x_e - \epsilon & e \in P^- \\ x_e & e \notin P \end{cases}$$

This is known as pushing ϵ units of flow along P, or simply augmenting along P. Oberve that flow conservation at any vertex u still holds when pushing flow along a path. This is trivial if u is not on the path, and if u is on the path, the contributions of the two arcs indicent to u on P cancel each other. To make sure the resulting x' is feasible (satisfies the capacity constraints), we choose

$$\epsilon = \min \left(\min_{e \in P^+} (u(e) - x_e), \min_{e \in P^-} x_e - l(e) \right).$$

By construction of the residual graph we have that $\epsilon > 0$. Thus, pushing ϵ units of flow along an augmenting path provides a new flow x' whose value $|x'|$ satisfy $|x'| = |x| + \epsilon$. Thus the flow x was not maximum.

Conversely, assume that the residual graph G_x does not contain any directed path from s to t. Let $S = \{u \in V : \text{there exists a directed path in } G_x \text{ from } s \text{ to } u\}$. By definition, $s \in S$ and $t \notin S$ (otherwise there would be an augmenting path). Also, by definition, there is no arc in G_x from S to $V \setminus S$. This means that, for $e \in E$, if $e \in \delta^+(S)$ then $x_e = u(e)$ and if $e \in \delta^-(S)$ then $x_e = l(e)$. This implies that

$$C(S) = \sum_{e \in \delta^+(S)} u(e) - \sum_{e \in \delta^-(S)} l(e) = u(\delta^+(S)) - l(\delta^-(S)) = \sum_{e \in \delta^+(S)} x_e - \sum_{e \in \delta^-(S)} x_e = |x|.$$

This shows that the flow x is maximum and there exists an $s-t$ cut of the same capacity as $|x|$.

This almost proves Theorem 6.3. Indeed, as long as there exists an augmenting path, we can push flow along it, update the residual graph and continue. Whenever this algorithm stops, *if it stops*, we have a maximum flow and a corresponding minimum cut. But maybe this algorithm never stops; this can actually happen if the capacities might be irrational and the "wrong" augmenting paths are chosen at every iteration. To complete the proof of the max flow min cut theorem, we can simply use the linear programming formulation of the maximum flow problem and this shows that a maximum flow exists (in a linear program, the max is a real maximum (as it is achieved by a vertex) and not just a supremum which may not be attained). Starting from that flow x and constructing its residual graph G_x, we get that there exists a corresponding minimum $s-t$ cut of the same value.

6.2.1 Interpretation of max flow min cut

The max $s-t$ flow min $s-t$ cut theorem together with integrality of the maximum flow allows to derive several combinatorial min-max relations.

Bipartite matchings. Consider for example the maximum bipartite matching problem and its formulation as a maximum flow problem given in section 6.1.1. We said that for the arcs between A and B we had flexibility on how we choose $u(e)$; here, let us assume we have set them to be equal to $+\infty$ (or any sufficiently large integer). Consider any set $S \subseteq (\{s\} \cup A \cup B)$ with $s \in S$ (and $t \notin S$). For $C(S)$ to be finite there cannot be any edge $(i,j) \in E$ between $i \in A \cap S$ and $j \in B \setminus S$. In other words, $N(A \cap S) \subseteq B \cap S$, i.e. if we set $C = (A \setminus S) \cup (B \cap S)$ we have that C is a vertex cover. What is the capacity $C(S)$ of the corresponding cut? It is precisely $C(S) = |A \setminus S| + |B \cap S|$, the first term corresponding to the arcs from s to $A \setminus S$ and the second term correspodning to the arcs between $B \cap S$ and t. The max $s-t$ flow min $s-t$ cut theorem therefore implies that there exists a vertex cover C whose cardinality equals the size of the maximum matching. We have thus rederived König's theorem. We could also derive Hall's theorem about the existence of a perfect matching.

Arc-disjoint paths. For the problem of teh maximum number of arc-disjoint paths between s and t, the max $s-t$ flow min $s-t$ cut theorem can be interpreted as Menger's theorem:

Theorem 6.4 *In a directed graph $G=(V,A)$, there are k arc-disjoint paths between s and t if and only if for all $S \subseteq V \setminus \{t\}$ with $s \in S$, we have $|\delta^+(S)| \geq k$.*

6.3 Efficiency of Maximum Flow Algorithm

The proof of the max $s-t$ flow min $s-t$ cut theorem suggests a simple augmenting path algorithm for finding the maximum flow. Start from any feasible flow and keep pushing flow along an augmenting in the residual graph until no such augmenting path exists. The main question we address now is how many iterations does this algorithm need before terminating.

As mentioned earlier, if the capacities are irrational, this algorithm may never terminate. In the case of integral capacities, if we start from an integral flow, it is easy to see that we always maintain an integral flow and we will always be pushing an integral amount of flow. Therefore, the number of iterations is bounded by the maximum difference between the values of two flows, which is at most $\sum_{e \in \delta(s)} (u(e) - l(e))$. This is finite, but not polynomial in the size of the input (which depends only logarithmically on the capacities u and l).

Shortest augmenting path variant. Edmonds and Karp proposed a variant of the augmenting path algorithm which is guaranteed to terminate in a polynomial number of iterations depending only on $n = |V|$ and $m = |E|$. No assumptions on the capacities are made, and the algorithm is even correct and terminates for irrational capacities.

The idea of Edmonds and Karp is to always find in the residual graph a *shortest* augmenting path, i.e. one with the fewer number of arcs. Given a flow x, consider the residual graph G_x. For any vertex v, let $d(v)$ denote the distance (number of arcs) from s to v in G_x. The shortest augmenting path algorithm is to select a path $v_0 - v_1 - \cdots - v_k$ in the residual graph where $v_0 = s$, $v_k = t$ and $d(v_i) = i$.

The analysis of the algorithm proceeds as follows. Let P be a shotest augmenting path from s to t in G_x and let x' be the resulting flow after pushing as much flow as possible along P. Let d' be the distance labels corresponding to $G_{x'}$. Observe that only reverse arcs (i,j) along P (thus satisfying $d(i) = d(j)+1$) may get introduced in $G_{x'}$. Therefore, after augmentation, we have that $d(j) - d(i) \leq 1$ for every arc $(i,j) \in E_{x'}$. Summing these inqualities along the edges of any path P' in $G_{x'}$ from s to $j \in V$, we get that $d(j) \leq d'(j)$ for any $j \in V$. In particular, we have that $d(t) \leq d'(t)$. As distance labels can never become greater than $n-1$, we have that the distance to t can only increase at most $n-1$ times. But $d'(t)$ can also be equal to $d(t)$. In this case though, the fact that an arc of P is saturated means that there is one fewer arc (i,j) with $d(j) = d(i)+1$ in $G_{x'}$ than in G_x. Thus after at most m such iterations, we must have a strict increase in the distance label of t. Summarizing, this means that the number of augmentations is at most $m(n-1)$. The time it takes to build the residual graph and to find an augmenting path in it is at most $O(m)$

time. This means that the total running time of the shortest augmenting path algorithm is at most $O(m^2 n)$. This can be further improved but this is not the focus of these notes.

6.4 Minimum cuts

From now on, we assume that we have only upper capacities u and no lower capacities l ($l(e) = 0$ for all e). The minimum $s - t$ cut problem that we have solved so far corresponds to:

$$\min_{S: s \in S, t \notin S} u(\delta^+(S)).$$

If our graph $G = (V, E)$ is undirected and we would like to find the minimum $s - t$ cut, i.e.

$$\min_{S: s \in S, t \notin S} u(\delta(S)),$$

we can simply replace every edge e by two opposite arcs of the same capacity and reduce the problem to finding a minimum $s - t$ cut in a directed graph. As we have just shown, this can be done by a maximum flow computation.

Now, consider the problem of finding the *global* minimum cut in a graph. Let us first consider the directed case. Finding the global mincut (or just the mincut) means finding S minimizing:

$$\min_{S: \emptyset \neq S \neq V} u(\delta^+(S)).$$

This problem can be reduced to $2(n - 1)$ maximum flow computations (where $n = |V|$) in the following way. First we can arbitrarily choose a vertex $s \in V$ and s will either be in S or in $V \setminus S$. Thus, for any $t \in V \setminus \{s\}$, we solve two maximum flow problems, one giving us the minimum $s - t$ cut, the other giving us the minimum $t - s$ cut. Taking the minimum over all such cuts, we get the global mincut in a directed graph.

To find the minimum cut problem in an undirected graph, we do not even need to solve two maximum flow problems for each $t \in V \setminus \{s\}$, only one of them is enough. Thus the global minimum cut problem in an undirected graph can be solved by computing $n - 1$ maximum flow problems. The fastest maximum flow algorithms currently take slightly more than $O(mn)$ time (for example, Goldberg and Tarjan's algorithm [1] take $O(mn \log(n^2/m))$ time). Since we need to use it $n - 1$ times, we can find a mincut in $O(mn^2 \log(n^2/m))$ time. However, these $n - 1$ maxflow problem are related, and Hao and Orlin [2] have shown that it is possible to solve *all* of them in $O(mn \log(n^2/m))$ by modifying Goldberg and Tarjan's algorithm. Thus the minimum cut problem can be solved within this time bound.

We will now derive an algorithm for the mincut problem which is not based on network flows, and which has a running time slightly better than Hao and Orlin's. The algorithm is due to Stoer and Wagner [6], and is a simplification of an earlier result of Nagamochi and Ibaraki [5]. We should also point out that there is a randomized algorithm due to Karger and Stein [4] whose running time is $O(n^2 \log^3 n)$, and a subsequent one due to Karger [3] that runs in $O(m \log^3 n)$.

We first need a definition. Define, for any two sets $A, B \subseteq V$ of vertices,

$$u(A:B) := \sum_{i \in A, j \in B} u((i,j)).$$

The algorithm is described below. In words, the algorithm starts with any vertex, and build an ordering of the vertices by always adding to the selected vertices the vertex whose total cost to the previous vertices is maximized. The cut induced by the last vertex in the ordering is considered, as well as the cuts obtained by recursively applying the procedure to the graph obtained by shrinking the last two vertices. (If there are edges from a vertex v to these last two vertices then we substitute those two edges with only one edge having capacity equal to the sum of the capacities of the two edges.) The claim is that the best cut among the cuts considered is the overall mincut. The formal description is given below.

MINCUT(G)
- ▷ Let v_1 be any vertex of G
- ▷ $n = |V(G)|$
- ▷ $S = \{v_1\}$
- ▷ **for** $i = 2$ **to** n
  - ▷ let v_i the vertex of $V \setminus S$ s.t.
  - ▷ $c(S : \{v\})$ is maximized (over all $v \in V \setminus S$)
  - ▷ $S := S \cup \{v_i\}$
- ▷ **endfor**
- ▷ **if** $n = 2$ **then** return the cut $\delta(\{v_n\})$
- ▷ **else**
  - ▷ Let G' be obtained from G by shrinking v_{n-1} and v_n
  - ▷ Let C be the cut returned by MINCUT(G')
  - ▷ Among C and $\delta(\{v_n\})$ return the smaller cut (in terms of cost)
- ▷ **endif**

Figure 6.1 illustrates how the algorithm works on an example. The analysis is based on the following crucial claim.

Claim 6.5 *$\{v_n\}$ (or $\{v_1, v_2, ..., v_{n-1}\}$) induces a min (v_{n-1}, v_n)-cut in G. (Notice that we do not know in advance v_{n-1} and v_n.)*

From this, the correctness of the algorithm follows easily. Indeed, the mincut is either a (v_{n-1}, v_n)-cut or not. If it is, we are fine thanks to the above claim. If it is not, we can assume by induction on the size of the vertex set that it will be returned by the call MINCUT(G').

Proof: Let $v_1, v_2, ..., v_i, ..., v_j, ..., v_{n-1}, v_n$ be the sequence of vertices chosen by the algorithm and let us denote by A_i the sequence $v_1, v_2, ..., v_{i-1}$. We are interested in the cuts that

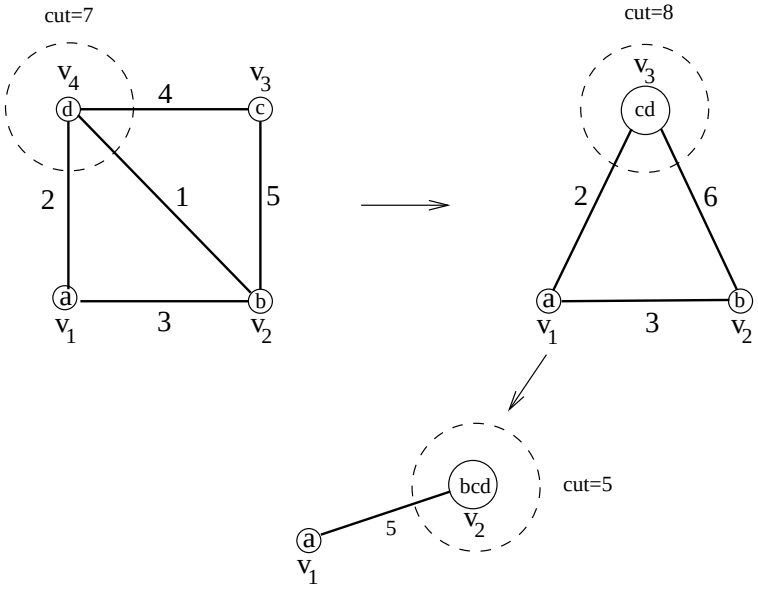

Figure 6.1: Illustration of the mincut algorithm.

separate v_{n-1} and v_n. Let C be any set such that $v_{n-1} \in C$ and $v_n \notin C$. Then we want to prove that the cut induced by C satisfies

$$u(\delta(C)) \geq u(\delta(A_n)).$$

Let us define vertex v_i to be critical with respect to C if either v_i or v_{i-1} belongs to C but not both. We claim that if v_i is critical then

$$u(A_i : \{v_i\}) \leq u(C_i : A_i \cup \{v_i\} \setminus C_i)$$

where $C_i = (A_i \cup \{v_i\}) \cap C$.

Notice that this implies that $u(\delta(C)) \geq u(\delta(A_n))$ because v_n is critical. Now let us prove the claim by induction on the sequence of critical vertices.

Let v_i be the first critical vertex. Then

$$u(A_i : \{v_i\}) = u(C_i : A_i \cup \{v_i\} \setminus C_i)$$

Thus the base of the induction is true.

For the inductive step, let the assertion be true for critical vertex v_i and let v_j be the next (after v_i) critical vertex. Then

$$\begin{aligned} u(A_j : \{v_j\}) &= u(A_i : \{v_j\}) + u(A_j \setminus A_i : \{v_j\}) \\ &\leq u(A_i : \{v_i\}) + u(A_j \setminus A_i : \{v_j\}) \\ &\leq u(C_i : A_i \cup \{v_i\} \setminus C_i) + u(A_j \setminus A_i : \{v_j\}) \\ &\leq u(C_j : A_j \cup \{v_j\} \setminus C_j), \end{aligned}$$

the first inequality following from the definition of v_i, the second inequality from the inductive hypothesis, and the last from the fact that v_j is the next critical vertex. The proof is concluded observing that A_n induces the cut $\{v_1, v_2, \cdots, v_{n-1}\} : \{v_n\}$. $\triangle$

The running time depends on the particular implementation. Using Fibonacci heaps we can implement each iteration in $O(m + n \log n)$ time and this yields a total running time of $O(mn + n^2 \log n)$.

6.5 Minimum T-odd cut problem

Given a graph $G = (V, E)$ with nonnegative edge capacities given by u and an even set T of vertices, the minimum T-odd cut problem is to find S minimizing:

$$\min_{S \subset V : |S \cap T| \text{ odd}} u(\delta(S)).$$

We'll say that S is T-odd if $|S \cap T|$ is odd. Observe that if S is T-odd, so is $V \setminus S$ and vice versa.

We give a polynomial-time algorithm for this problem. We won't present the most efficient one, but one of the easiest ones. Let $ALG(G, T)$ denote this algorithm. The first step of $ALG(G, T)$ is to find a minimum cut having at least one vertex of T on each side:

$$\min_{S \subset V : \emptyset \neq S \cap T \neq T} u(\delta(S)).$$

This can be done by doing $|T| - 1$ minimum $s - t$ cut computations, by fixing one vertex s in T and then trying all vertices $t \in T \setminus \{s\}$, and then returning the smallest cut S obtained in this way.

Now, two things can happen. Either S is a T-odd cut in which case it must be minimum and we are done, or S is T-even (i.e. $T \cap S$ has even cardinality). If S is T-even, we show in the lemma below that we can assume that the minimum T-even cut A is either a subset of S or a subset of $V \setminus S$. Thus we can find by recursively solving 2 smaller minimum T-odd cut problems, one in the graph $G_1 = G/S$ obtained by shrinking S into a single vertex and letting $T_1 = T \setminus S$ and the other in the graph $G_2 = G/(V \setminus S)$ obtained by shrinking $V \setminus S$ and letting $T_2 = T \setminus (V \setminus S) = T \cap S$. Thus the algorithm makes two calls, $ALG(G_1, T_1)$ and $ALG(G_2, T_2)$ and returns the smallest (in terms of capacity) T-odd cut returned.

At first glance, it is not obvious that this algorithm is polynomial as every call may generate two recursive calls. However, letting $R(k)$ denote an upper bound on the running time of $ALG(G, T)$ for instances with $|T| = k$ (and say $|V| \leq n$), we can see that

1. $R(2) = A$, where A is the time needed for a minimum $s - t$ cut computation,
2. $R(k) \leq \max_{k_1 \geq 2, k_2 \geq 2, k = k_1 + k_2} \left((k-1)A + R(k_1) + R(k_2) \right)$.

By induction, we can see that $R(k) \leq k^2 A$, as this is true for $k = 2$ and the inductive step is also satisfied:

$$\begin{aligned} R(k) &\leq \max_{k_1 \geq 2, k_2 \geq 2, k = k_1 + k_2} \left((k-1)A + k_1^2 A + k_2^2 A \right) \\ &\leq (k-1)A + 4A + (k-2)^2 A \\ &= (k^2 - 3k + 7)A \\ &\leq k^2 A, \end{aligned}$$

for $k \geq 4$. Thus, this algorithm is polynomial.

We are left with stating and proving the following lemma.

Lemma 6.6 *If S is a minimum cut among those having at least one vertex of T on each side, and $|S \cap T|$ is even then there exists a minimum T-odd cut A with $A \subseteq S$ or $A \subseteq V \setminus S$.*

Proof: Let B be any minimum T-odd cut. Partition T into T_1, T_2, T_3 and T_4 as follows: $T_1 = T \setminus (B \cup S)$, $T_2 = (T \cap S) \setminus B$, $T_3 = T \cap B \cap S$, and $T_4 = (T \cap B) \setminus S$. Since by definition of B and S we have that $T_1 \cup T_2 \neq \emptyset$, $T_2 \cup T_3 \neq \emptyset$, $T_3 \cup T_4 \neq \emptyset$ and $T_4 \cup T_1 \neq \emptyset$, we must have that either T_1 and T_3 are non-empty, or T_2 and T_4 are non-empty. Possibly replacing B by $V \setminus B$, we can assume that T_1 and T_3 are non-empty.

By submodularity of the cut function, we know that

$$\sum_{e \in \delta(S)} u(e) + \sum_{e \in \delta(B)} u(e) \geq \sum_{e \in \delta(S \cup B)} u(e) + \sum_{e \in \delta(S \cap B)} u(e). \tag{3}$$

Since $T_1 \neq \emptyset$ and $T_3 \neq \emptyset$, both $S \cup B$ and $S \cap B$ separate vertices of T. Furthermore, one of them has to be T-even and the other T-odd, as $|(S \cap B) \cap T| + |(S \cup B) \cap T| = |T_2| + 2|T_3| + |T_4| = |S \cap T| + |B \cap T|$ is odd. Thus, one of $S \cup B$ and $S \cap B$ has to have a cutvalue no greater than the one of B while the other has a cut value no greater than the one of S. This means that either $S \cap B$ or $S \cup B$ is a minimum T-odd cut. $\triangle$

References

[1] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem", *Journal of the ACM*, **35**, 921–940, 1988.

[2] X. Hao and J.B. Orlin, "A faster algorithm for finding the minimum cut in a graph", *Proc. of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, 165–174, 1992.

[3] D. Karger, "Minimum cuts in near-linear time", *Proc. of the 28th STOC*, 56–63, 1996.

[4] D. Karger and C. Stein, "An $\tilde{O}(n^2)$ algorithm for minimum cuts", *Proc. of the 25th STOC*, 757–765, 1993.

[5] H. Nagamochi and T. Ibaraki, "Computing edge-connectivity in multigraphs and capacitated graphs", *SIAM Journal on Discrete Mathematics*, **5**, 54–66, 1992.

[6] M. Stoer and F. Wagner, "A simple mincut algorithm", *Proc. of ESA94*, Lecture Notes in Computer Science, **855**, 141–147, 1994.