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

Handout 17
November 17th, 2005

Lecture notes on the mincut problem

1 Minimum Cuts

In this lecture we will describe an algorithm that computes the minimum cut (or simply mincut) in an undirected graph. A cut is defined as follows.

Definition 1 *Given a graph $G = (V, E)$ and a subset S of V, the cut $\delta(S)$ induced by S is the subset of edges $(i, j) \in E$ such that $|\{i, j\} \cap S| = 1$.*

That is, $\delta(S)$ consists of all those edges with exactly one endpoint in S.

Given an undirected graph $G = (V, E)$ and for each edge $e \in E$ a nonnegative cost (or capacity) c_e, the cost of a cut $\delta(S)$, is the sum of the costs of the edges in the cut, that is

$$c(\delta(S)) = \sum_{e \in \delta(S)} c_e.$$

The *minimum cut problem* (or mincut problem) is to find a cut of minimum cost. If all costs are 1 then the problem becomes the problem of finding a cut with as few edges as possible.

Cuts are often defined in a different, not completely equivalent, way. Define a *cutset* to be a set of edges whose removal disconnects the graph into at least two connected components. *Minimal* cutsets (a minimal cutset C is a cutset such that any proper subset of C is not anymore a cutset) can be seen to correspond to cuts $\delta(S)$ for which the subgraphs induced by S and $V - S$ are connected. Observe that only minimal cutsets can be of minimum cost (among all cutsets) and that only cuts $\delta(S)$ for which both S and $V - S$ induce connected components can be of minimum cost (among all cuts) since the costs are assumed to be nonnegative. For this reason, the problem of finding a cutset of minimum cost is equivalent to the problem of finding a cut $\delta(S)$ of minimum cost, namely the mincut problem. From now on, we will only look at cuts $\delta(S)$ (and not cutsets).

An important variant of the mincut problem is often considered. This is the problem of finding the minimum cost cut separating two given two vertices s and t. A cut $\delta(S)$ is said to separate s and t if only one of them belongs to S. We refer to this problem as the minimum (s, t)-cut problem.

As seen in lecture, the minimum (s, t)-cut problem can be solved by means of network flow algorithms. Indeed it can be reduced to a max flow problem. Given a source s and a sink t of the graph G, we have seen that

$$\text{MAX FLOW}(s, t) = \min_{S: s \in S, t \notin S} c(\delta(S)).$$

Notice that this result relates the value of the maximum flow from s to t and the value of the minimum (s,t)-cut It does not specify any relationship between the minimum (s,t)-cut itself (meaning the edges composing the cut) and the way the maximum flow can be pushed into the graph. However, given a maximum flow, it is easy to obtain the corresponding (s,t)-mincut. If you look at the residual graph corresponding to the maximum flow, the set S of vertices reachable from S will induce a minimum cut. By definition of the residual graph (and properties of maximum flows), the cost of this cut is equal to the value of the maximum flow and thus it is a min (s,t)-cut. On the other hand, the knowledge of a min (s,t)-cut does not help in finding the actual maximum flow (not just its value but the flow on every edge). Indeed, consider the following example. Let C^* be the min (s,t)-cut value in $G=(V,E)$ and let us consider the graph $G'=(V',E')$ where $V'=\{s'\}\cup V$ and $E'=\{(s',s)\}\cup E$ with $c(s',s)=C^*$. Then a possible minimum (s',t)-cut is $\delta(\{s'\})$. However this does not give any more information on how the flow can be pushed from s' to t (than just its value C^*). So far no algorithm that finds a minimum (s,t)-cut without using a reduction to the max flow problem has been discovered. We will see that for general mincuts (ot separating two given vertices), the situation is different.

How can we find a minimum cut in an undirected graph? One possibility is to choose a vertex s and compute the min (s,t)-cuts for every $t \in V-\{s\}$, and choose the cut of minimum cost among all the cuts obtained. 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 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.

In this lecture, we will 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 of vertices of the graph,

$$c(A:B) := \sum_{i\in A, j\in B} c_{i,j}$$

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 1 illustrates how the algorithm works on an example.

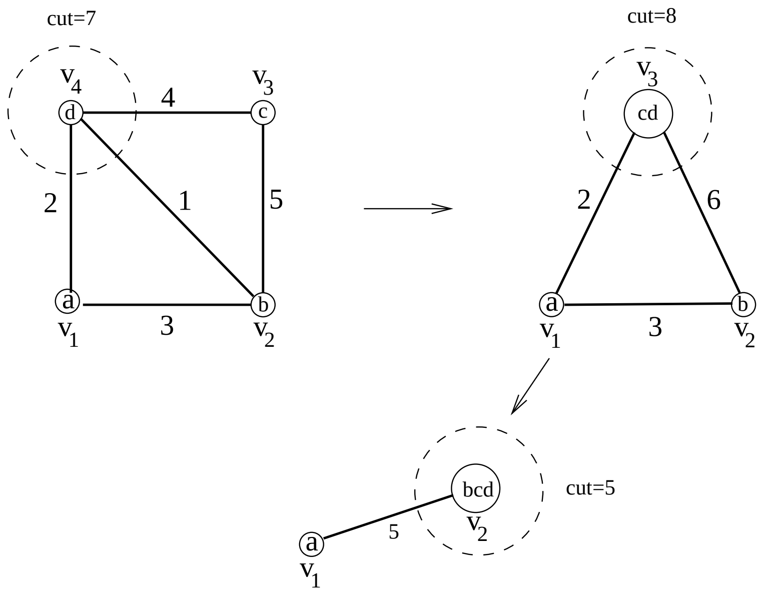

Figure 1: Illustration of the mincut algorithm.

The analysis is based on the following crucial claim.

Claim 1 *$\{v_n\}$ (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 $\textsc{mincut}(G')$.

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 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

$$c(\delta(C)) \geq c(\delta(A_n))$$

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

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

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

Notice that this implies that $c(\delta(C)) \geq c(\delta(A_n))$ 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

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

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

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

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 $O(mn + n^2 \log n)$.

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.