Global Min-Cut

In a graph $G = (V, E)$ with $n$ nodes and $m$ edges we wish to find a global min-cut $(A, B)$, that is, a partitioning $V = A \cup B$ such that the number of cut edges (those edges between $A$ and $B$) is minimized. Motivations include the assessment of reliability of networks, finding clusters in graphs, and efficient hierarchical computation of distances in graphs.

We can easily reduce the problem to Minimum Cut, by trying all possible pairs of sources and sinks $s, t \in V$. But since flow and cut algorithms are somewhat sophisticated, you may be pleased to learn an extremely simple randomized algorithm that solves the Global Min-Cut problem as well. However this comes with a price: Success is no longer guaranteed. We will get a correct solution “only” with high probability.

Randomized algorithms should not be confused with average-case analysis. All randomness is with the algorithms, while nothing is assumed about the probability distribution of inputs. The analysis results (expected time, probability of a correct solution, etc.) will hold for every instance, not only averaged on many instances.

For simplicity we discuss only the basic randomized algorithm for Global Min-Cut, although faster algorithms are known. In the following we have to allow graphs with parallel (multiple) edges. The algorithm works as follows. In every step, choose an edge $e = (u, v)$ at random and contract it. Contraction means: shrink $e$, identify $u, v$ (merge them into a new vertex), and delete all edges that have been parallel to $e$ (they would be loops at the new vertex). Iterate this step until two nodes remain. This two-node graph represents a cut, in the obvious sense. The whole procedure is repeated a certain number of times from scratch, and finally we output the smallest cut found in this way.

It may seem that this algorithm has nothing to do with the problem. It just repeatedly contracts random edges. However, the intuition is that a small cut has a chance not to be affected by these random contractions, thus being preserved in the end. Still, the analysis which has to confirm this intuition is not so obvious. It uses a clever combination of several elementary tools from probability theory.

Consider any global min-cut $(A, B)$. Let $F$ denote the set of its cut edges, and $k := |F|$. After $j$ steps of the algorithm, clearly the contracted graph has $n - j$ nodes. Moreover, every node has degree at least $k$, since otherwise the node and its complement set would already form a global min-cut smaller than $k$, a contradiction. Hence at least $k(n - j)/2$ edges still exist after $j$ steps. Therefore, the probability that unfortunately some of the $k$ edges in $F$ is contracted in the next step is at most $2/(n - j)$. That means, our specific cut $(A, B)$ is returned with probability at least $\prod_{j=0}^{n-3}(1 - 2/(n - j)) = \prod_{j=0}^{n-3}((n - j - 2)/(n - j)) = 2/n(n - 1)$ after the contraction procedure. (However, think carefully: Why is it correct to multiply the probabilities, although the events are certainly not independent?) This is a small probability, but we repeat this $O(m)$-time contraction procedure sufficiently often: Each run fails with probability $1 - 2/n(n - 1)$, but a simple calculation shows that some of $O(n^2)$ runs succeeds, subject to a small constant failure probability. We can make this failure probability arbitrarily small by increasing the hidden constant factor in $O(n^2)$. (Note the superficial similarity to approximation schemes.)
This is an example of a so called Monte Carlo algorithm: It runs in polynomial time in the worst case, but the result can be wrong with some (small) probability. This failure probability can be reduced exponentially by repeated runs. By way of contrast, a Las Vegas algorithm gives always the correct result, but only an expected time bound is given, and the running time can be higher in the worst case.

Introduction to Approximation Algorithms: Minimum Vertex Cover

Given \( G = (V, E) \), we want to have a minimum cardinality vertex cover. A vertex cover is a subset \( U \subseteq V \) of nodes so that for every edge \((u, v) \in E\), either \( u \) or \( v \), or both, are part of \( U \). This problem is NP-hard, so unlikely to be solved in polynomial time. We naturally shoot for approximations.

A very tempting idea to attack this problem is the greedy algorithm: we pick the vertex with the largest degree, remove it and its incident edges from the graph and repeat this process. It is known that this idea does not work—it does not give us constant approximation. (It should be emphasized that the greedy algorithm is often a very effective strategy. It conforms to our intuition, is easy to implement, and sometimes is simply the best possible approximation algorithm possible. But for this time, greedy is bad).

Consider the following idea. Find a maximal matching \( M \subseteq E \) (a matching \( M \) is maximal if you cannot add one more edge \( e \not\in M \) into \( M \) while still having a matching). Then return all nodes in \( M \) as the solution. (Why is this solution valid?)

This is our first algorithm. It is a 2-approximation algorithm and as simple as could be. It can be implemented in polynomial time (in fact linear time). Now let us think why the solution is 2-approximation. Observe the following fact: given an edge \( e = (u, v) \in M \), in the optimal solution, either \( u \) or \( v \) has to be included; on the other hand, our solution picks both \( u \) and \( v \). So our solution is at most twice bigger than the optimal.

We mention in passing that it may be hoped that a more sophisticated algorithm can do better than 2-approximation. As it turns out, this simple algorithm is already the best possible (based on some widely-believed conjecture in complexity theory).

Center Selection

Let \( S \) be a set of \( n \) sites \( s_1, s_2, \ldots, s_n \) in a metric space equipped with a distance function \( \text{dist} \), and \( k \) a given number. The goal is to select a set \( C \) of \( k \) centers among the \( n \) sites so as to minimize the maximum distance of a site to the nearest chosen center. To be more formal, let \( \text{dist}(s, C) = \min_{c \in C} \text{dist}(s, C) \). \( C \) forms a \( r \)-cover if \( \forall s_i, \text{dist}(s_i, C) \leq r \). The minimum \( r \) for which \( C \) is an \( r \)-cover is the \textit{radius} of \( C \). We want a set \( C \) of \( k \) centers so that \( C \) has the smallest possible radius.

Let us remark what metric space is. In a metric space, the triangle inequality is satisfied, that is, given any three sites \( a, b, c \), \( \text{dist}(a, c) \leq \text{dist}(a, b) + \text{dist}(b, c) \). A metric space arises naturally (e.g. Euclidean distance) and in fact many optimization problems
become a lot easier to solve in such a space. The following algorithm critically makes use of the fact that we are given a metric space.

Assume for the moment that a little bird comes and tell us the optimal radius \( r_{\text{opt}} \) (but not the solution itself). As we will see, this would make the problem much easier. (Of course, later we have to drop this assumption.) We say that a center \( c \) “covers” a site \( s \), with covering radius \( r_{\text{opt}} \), if \( \text{dist}(c, s) \leq r_{\text{opt}} \).

Let us make some observation before we embark on the formal algorithm. Consider any uncovered site \( s \). How do we find a nearby center \( c \) to cover \( s \) when we design our approximation algorithm? Here is a lazy and crazy idea. Why not just choose \( s \) itself as a center? One may worry that in the optimal solution, some other cite \( c \neq s \) is chosen to be a center to cover \( s \) and \( c \) also covers a lot of other sites \( s' \). If we shift the center from \( c \) to \( s \), then \( s' \) is no longer covered. The important observation here is that \( s' \) cannot be too far away from \( s \) due to our metric space assumption, namely \( \text{dist}(s, s') \leq \text{dist}(c, s) + \text{dist}(c, s') \leq 2r_{\text{opt}} \). So if we allow the radius of \( s \) to be \( 2r_{\text{opt}} \), then all sites \( s' \) originally covered by \( c \) will be covered by \( s \) as well.

The above observation suggests the following algorithm. Assume that we guess (rightly or wrongly) that \( r \) is the optimal radius. Choose an arbitrary site as a center and remove it, along with all other sites within radius \( 2r \) of this site. Repeat this process until no more site is left. Let us call the chosen centers \( s_1, s_2, \ldots \). In the end, if at most \( k \) sites are chosen as centers, we apparently have a solution with radius \( r \). But what if we end up choosing more than \( k \) centers? Let us prove that in this case the guess \( r \) has to be a wrong guess.

Consider any chosen center \( s_i \). In the optimal solution, there must exist a center \( c^* \) that covers \( s_i \), i.e., \( \text{dist}(s_i, c^*) \leq r \). Could it happen that another chosen center \( s_j \neq s_i \) is also covered by \( c^* \), namely, \( \text{dist}(s_j, c^*) \leq r \)? If this happens, then \( d(s_i, s_j) \leq d(s_i, c^*) + d(s_j, c^*) \leq 2r \), a contradiction (the chosen centers must be at least \( 2r \) away from one another by the algorithm). So if we choose more than \( k \) centers in the algorithm the optimal solution must have also more than \( k \) centers, a contradiction. We can safely conclude that \( r \) is a wrong guess.

The final question: how do we guess \( r \)? One may think about some binary search. But here is a quick solution. There are only \( n \) sites. The optimal solution \( r \) has to be the distance between some two sties. There can be only \( \binom{n}{2} \) possibilities. Thus our running time is polynomial.