CS265/CME309: Randomized Algorithms and Probabilistic Analysis
Lecture #2: Karger's Min-Cut Algorithm, Coupon Collector, and Quicksort with Random Pivot.

Gregory Valiant*

November 14, 2019

1 Karger's Min-Cut Algorithm

Today, we begin with an incredibly elegant randomized algorithm for finding the minimum cut in a graph, due to David Karger from 1993 [1]. Given a graph, the min-cut problem asks us to partition the vertices into two sets, so as to minimize the number of edges that cross the partition (i.e. that have one endpoint in each set). This fundamental problem (and its many variants, including partitioning into $k > 2$ sets), have many applications, including clustering webpages and social graphs, as well as documents (say two documents have an edge between them if one references the other, or we could have a weighted edge corresponding to the Jaccard similarity between the sets of words used in the documents, for example....if you haven't come across Jaccard similarity before, maybe do a quick wikipedia search.). Similar problems are also used to segment images, assign computation to processors, etc.

In contrast to polynomial identity testing (from lecture 1), where we do not know of any efficient (polynomial time) deterministic algorithm, min-cut *does* have an efficient deterministic algorithm, via solving a *max-flow* problem. Still, the randomized algorithm we present below is so clean and elegant, that it is worth including in the first week of this course. Additionally, with several tweaks (one of which you will investigate on the homework this week), the runtime of this randomized algorithm does become competitive with the best deterministic algorithms.

1.1 High-Level Intuition

Karger's min-cut algorithm starts with the original graph, and iteratively reduces the number of vertices via a series of *edge contractions*. In each step, an edge is chosen uniformly at

random, and then the two endpoints of that edge are merged into a single vertex, and all edges are preserved (we will allow multiple parallel edges between two vertices) except the 'self-loop' that is created by the merge. The algorithm proceeds iteratively until there are only two vertices left—call them u_1 and u_2—at which point the algorithm returns the partition (i.e. the cut) where all the vertices that were merged into u_1 are in one set, and all the vertices that were merged into u_2 are in the other set.

To see one intuition for why this algorithm might be expected to perform well, imagine that the original graph consists of two disconnected components. In this case, the above algorithm will (with probability 1) return the correct min-cut corresponding to these two disconnected components. Now imagine that the graph consists of two components (each of which has quite a lot of internal edges), where the two components are connected via a single edge (or, more generally, relatively few edges). The algorithm will be successful provided, at *every* step of the algorithm, it avoids selecting one of the edges that cross between the components. (Why? Well if the algorithm contract an edge that crosses between the components, then at least one of the vertices of the graph will have been merged into the "wrong" side of the cut.) If there is just a single edge crossing between these components, then we certainly don't expect it to be picked early in the algorithm (since there are so many other edges to choose from), and we will be able to argue that there will be a reasonable chance that we never contract that edge.

1.2 The Algorithm

We begin by formally defining an edge *contraction*.

Definition 1.1 Given a graph $G = (V, E)$ with n vertices $V = \{v_1, \ldots, v_n\}$, and an edge $e \in E$ that connects vertices v_i, v_j, the graph resulting from *contracting* edge e will have $n - 1$ vertices, namely $V \setminus \{v_i\}$, and edge set defined as follows: for every edge $e' \in E$ that does not have v_i as an endpoint, e' is an edge of the new graph. For every edge $e' \in E$ that connects v_i, v_k for $k \neq j$, we add the edge (v_j, v_k) to the edge set.

The following figure illustrates whats going on: we merge the two endpoints of the edge, keeping all edges except self-loops.

Figure 1: Example of a graph before and after contracting the orange edge.

We are now ready to formally state the algorithm.

Algorithm 1
MIN-CUT ALGORITHM
Given a graph $G = (V, E)$ with $|V| = n$ vertices:

1. For $i = 1$ to $n - 2$:
   - Choose one of the remaining edges of G uniformly at random, and contract that edge, so that G now has $n - i$ vertices.
2. Return the partition corresponding to the final 2-vertex graph (each of these two vertices represents a subset of V corresponding to all the vertices that eventually became merged).

At this point, you might be wondering how efficiently one can actually perform an edge contraction. Naively, this might require a fair bit of book-keeping, as much as the degree of the vertices being contracted (which could be as large as n). Still, there are some clever data structures can be used to help with this, and also a clever . For the remainder of these notes, we'll ignore how we actually implement a contraction, and instead focus on understanding the probability that the algorithm returns the minimum cut.

Theorem 1.2 *The probability that the above algorithm returns the minimum cut is at least* $\frac{2}{n(n-1)} \leq 2/n^2$.

The proof of the above theorem relies on the following easy lemma which argues that, at some intermediate stage of the algorithm, as long as we haven't yet contracted an edge that crosses the minimum cut of the original graph, then that minimum cut will also be a minimum cut of the graph we currently have.

Lemma 1.3 *If we consider a minimum cut, corresponding to* $S_1, S_2 \subset V$*, and contract an edge* (u, v) *that does* not *cross the cut, then* S_1, S_2 *will be a minimum cut of the new graph resulting from the contraction.*

Proof: For any cut in the new graph (after the contraction) that cuts s edges, that cut corresponds to a partition in the previous graph that also cuts s edges. Hence the number of edges cut by the smallest cut cannot decrease when we contract an edge. Finally, if we contract an edge that does not cross from S_1 to S_2, then partition S_1, S_2 still exists in the new graph, and cuts the same number of edges as before, and hence must still be a min-cut in the new graph. ■

Proof of Theorem 1.2. Although there might be more than one minimum cut, we will actually prove that, for any minimum cut, C, the probability the algorithm returns C is at least $\frac{2}{n(n-1)}$. By Lemma 1.3, the algorithm will return C if, and only if, each of the $n - 2$ edge contractions contract an edge that does not cross the cut C. Let E_i denote the event

that we do not contract an edge crossing C in the ith step of the algorithm. We have the following:

$$Pr[\text{output } C] \;=\; Pr[E_1] \cdot Pr[E_2|E_1] \cdot Pr[E_3|E_1, E_2] \cdot \ldots \cdot Pr[E_{n-2}|E_1, E_2, \ldots, E_{n-3}].$$

Letting k denote the number of edges crossing C in the original graph, we trivially have that $Pr[E_1] = 1 - \frac{k}{\text{total number edges}}$. Since C is, by assumption, a minimum cut, the degree of every vertex must be at least k, which implies that the total number of edges must be at least $nk/2$, because each of the n vertices have degree at least k, so the sum of degrees of the vertices will be at least nk, but this double-counts all the edges, hence the factor of 2. Hence we have that

$$Pr[E_1] \geq 1 - \frac{k}{nk/2} = 1 - \frac{2}{n} = \frac{n-2}{n}.$$

From Lemma 1.3, conditioned on $E_1, \ldots, E_{i-1}$, we now that C is a minimum cut of the graph before the ith contraction, and hence

$$Pr[E_i|E_1, E_2, \ldots, E_{i-1}] = 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}.$$

Combining these terms we conclude

$$Pr[\text{output } C] = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)},$$

where the last equality follows from observing that all numerators and denominators cancel except the denominators of the first two terms, and numerators of the last two terms. ■

Should we be happy with a probability of success of $\approx 2/n^2$? Trivially, if we simply repeat the above algorithm $t = cn^2/2$ times, and return the smallest cut that was found in any of the t runs, then the probability of failure becomes at most $(1 - \frac{2}{n^2})^{cn^2/2} \leq e^{-\frac{2}{n^2}} \cdot \frac{cn^2}{2} = e^{-c}$, where we used the trick from Lecture 1 where we noted that $1 - x \leq e^{-x}$. So, this implies that if we want a probability of success of, say, 0.9, we would need to perform $O(n^2 \cdot n)$ edge contractions—n for each of the $O(n^2)$ runs of the algorithm.

How can we do better? One approach to improving this is based on the following intuition: suppose we were told that a given run of the algorithm was not successful. If we needed to guess which iteration destroyed the minimum cut, we would probably guess that it was one of the later iterations. After all, the probability that our first contraction preserves the minimum cut is at least $\frac{n-2}{n} \approx 1$, whereas, even if everything has gone perfectly, the probability (according to our pessimistic calculations) that the very last contraction is successful might only be 1/3. This motivates the following idea: rather than repeating the entire algorithm, including all the work we have done during the first few iterations (which probably did not destroy the minimum cut), why not just re-do the last few edge contractions (with freshly chosen random choices for the edges to contract)? The last problem on this week's problem set explores this intuition in more detail.

2 Analysis of Quicksort with a Random Pivot

Many of you might have seen this in CS161. I covered it very quickly (≈ 5 minutes) in class, largely to illustrate the points that 1) linearity of expectation is extremely powerful, and 2) when analyzing the expectation of a random variable, it is often a good idea to represent that random variable as the sum of other random variables that are easy to analyze, and apply linearity of expectation.

Recall the recursive Quicksort algorithm for sorting a list of n numbers. For clarity, we describe the algorithm and its analysis assuming that the n numbers are all distinct (no repetitions), though the algorithm and analysis naturally extend to the general case.

Algorithm 2
QUICKSORT (WITH RANDOM PIVOT)
Given set/list of n distinct numbers, $S = (x_1, \ldots, x_n)$:

1. If $|S| = 0$ return the empty list.
2. Otherwise, select i uniformly at random from $\{1, \ldots, n\}$.
3. Compare every element of S to x_i, forming two sets, $S_<$ and $S_>$ consisting respectively of the number less than x_i, and the numbers that are greater than x_i.
4. Return the list corresponding to the concatenation of $Quicksort(S_<), (x_i), Quicksort(S_>)$.

The above algorithm, in the worst case, might require almost $n^2/2$ comparisons if the pivot that is chosen at every step of the recursion happens to be the minimum element of the set. Nevertheless, as we argue below, in expectation, the number of comparisons is at most $2n \log n + O(n)$, which is extremely good.

Theorem 2.1 *The expected runtime of the above algorithm is at most* $2n \log n + O(n)$.

Proof: To compute the expected number of comparisons, we will simply apply linearity of expectation, and then analyze the expected number of times that every pair of inputs is compared. For convenience, assume that the input set contains the numbers $z_1 < z_2 < \ldots z_n$. First note that for any pair z_i, z_j, we either compare them 0 times, or once, since if we do compare them at some iteration of the recursion, that means that z_i or z_j was a pivot, and we will never compare that pair again, because the pivot is not included in the sets $S_<$ or $S_>$. Given this, by linearity of expectation, we have

$$E[\#\text{ comparisons}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ is compared to } z_j \text{ at any point during the algorithm}].$$

To analyze this, we just need to think about the probability that z_i and z_j ever get compared during the algorithm.

One slick way of analyzing this is as follows: at every recursive execution of the algorithm, z_i and z_j will both end up in the same set (either $S_<$ or $S_>$) until the first time that an element of the set $R_{i,j} = z_i, z_{i+1}, \ldots, z_{j-1}, z_j$ is chosen as the pivot. After the first time that happens, z_i and z_j will be split up, and will never be able to be compared to each other. Hence, the probability they are compared is exactly equal to the probability that, the first time a number in $R_{i,j}$ is chosen, it happens to be either z_i or z_j. Hence this probability is exactly $\frac{2}{|R_{i,j}|} = \frac{2}{j-i+1}$. The rest is just some tedious calculations:

$$\begin{aligned}
E[\#\text{ comparisons}] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^{n} \frac{2}{k}(n+1-k) \\
&= 2(n-1) + (n+1)\sum_{k=2}^{n} \frac{2}{k} \\
&< 2(n-1) + 2(n+1)(1+\log n) = 2n\log n + O(n).
\end{aligned}$$

To get from the second-to-last line to the last line, we used the fact that $\sum_{i=1}^{n} \frac{1}{i}$ is between $\log n$ and $1 + \log n$, since $\int_{x=1}^{n} \frac{1}{x} = \log n$, and $\int_{x=1}^{n} \frac{1}{x} < \sum_{i=1}^{n} \frac{1}{i} < \int_{x=1}^{n+1} \frac{1}{x}$. ■

References

[1] David R Karger. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. In *SODA*, volume 93, pages 21–30, 1993.