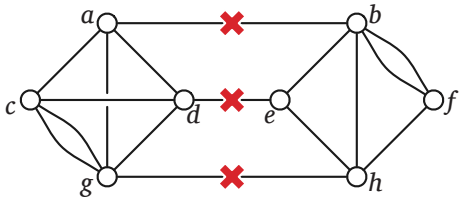
> **Jaques:** *But, for the seventh cause; how did you find the quarrel on the seventh cause?*
> **Touchstone:** *Upon a lie seven times removed:–bear your body more seeming, Audrey:–as thus, sir. I did dislike the cut of a certain courtier's beard: he sent me word, if I said his beard was not cut well, he was in the mind it was: this is called the Retort Courteous. If I sent him word again 'it was not well cut,' he would send me word, he cut it to please himself: this is called the Quip Modest. If again 'it was not well cut,' he disabled my judgment: this is called the Reply Churlish. If again 'it was not well cut,' he would answer, I spake not true: this is called the Reproof Valiant. If again 'it was not well cut,' he would say I lied: this is called the Counter-cheque Quarrelsome: and so to the Lie Circumstantial and the Lie Direct.*
> **Jaques:** *And how oft did you say his beard was not well cut?*
> **Touchstone:** *I durst go no further than the Lie Circumstantial, nor he durst not give me the Lie Direct; and so we measured swords and parted.*
>
> — William Shakespeare, *As You Like It*, Act V, Scene 4 (1600)

13 Randomized Minimum Cut

13.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph[1] G representing a communications network like the UIUC telephone system, the Facebook social network, the internet, or Al-Qaeda. In order to disrupt the network, an enemy agent plans to remove some of the edges in this multigraph (by cutting wires, placing police at strategic drop-off points, or paying street urchins to 'lose' messages) to separate it into multiple components. Since his country is currently having an economic crisis, the agent wants to remove as few edges as possible to accomplish this task.

More formally, a *cut* partitions the nodes of G into two nonempty subsets. The *size* of the cut is the number of *crossing edges*, which have one endpoint in each subset. Finally, a *minimum* cut in G is a cut with the smallest number of crossing edges. The same graph may have several minimum cuts.



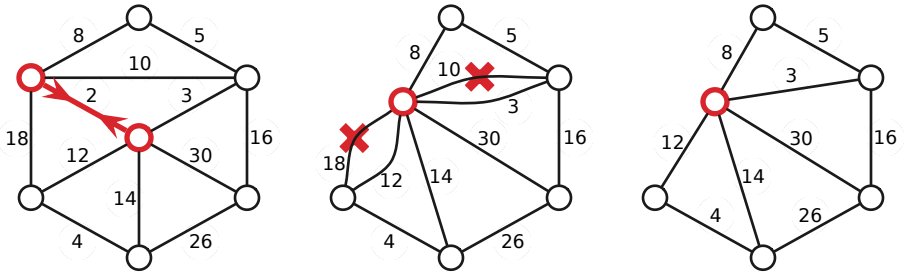
A multigraph whose minimum cut has three edges.

This problem has a long history. The classical deterministic algorithms for this problem rely on *network flow* techniques, which are discussed in another lecture. The fastest such algorithms (that we will discuss) run in $O(n^3)$ time and are fairly complex; we will see some of these later in the semester. Here I'll describe a relatively simple randomized algorithm discovered by David Karger when he was a Ph.D. student.[2]

[1] A multigraph allows multiple edges between the same pair of nodes. Everything in this lecture could be rephrased in terms of simple graphs where every edge has a non-negative weight, but this would make the algorithms and analysis slightly more complicated.

[2] David R. Karger*. Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.

Karger's algorithm uses a primitive operation called ***edge contraction***. Suppose u and v are vertices that are connected by an edge in some multigraph G. To contract the edge $\{u, v\}$, we create a new node called uv, replace any edge of the form $\{u, w\}$ or $\{v, w\}$ with a new edge $\{uv, w\}$, and then delete the original vertices u and v. Equivalently, contracting the edge shrinks the edge down to nothing, pulling the two endpoints together. The new contracted graph is denoted $G/\{u, v\}$. We don't allow self-loops in our multigraphs; if there are multiple edges between u and v, contracting any one of them deletes them all.



A graph G and two contracted graphs $G/\{b, e\}$ and $G/\{c, d\}$.

Any edge in an n-vertex graph can be contracted in $O(n)$ time, assuming the graph is represented as an adjacency list; I'll leave the precise implementation details as an easy exercise.

The correctness of our algorithms will eventually boil down the following simple observation: For any cut in $G/\{u, v\}$, there is cut in G with exactly the same number of crossing edges. In fact, in some sense, the 'same' edges form the cut in both graphs. The converse is not necessarily true, however. For example, in the picture above, the original graph G has a cut of size 1, but the contracted graph $G/\{c, d\}$ does not.

This simple observation has two immediate but important consequences. First, contracting an edge cannot decrease the minimum cut size. More importantly, contracting an edge increases the minimum cut size if and only if that edge is part of *every* minimum cut.

13.2 Blindly Guessing

Let's start with an algorithm that tries to *guess* the minimum cut by randomly contracting edges until only two vertices remain.

```
GuessMinCut(G):
  for i ← n downto 2
      pick a random edge e in G
      G ← G/e
  return the only cut in G
```

Because each contraction requires $O(n)$ time, this algorithm runs in $O(n^2)$ time. Our earlier observations imply that as long as we never contract an edge that lies in every minimum cut, our algorithm will actually guess correctly. But how likely is that?

Suppose G has only one minimum cut—if it actually has more than one, just pick your favorite—and this cut has size k. Every vertex of G must lie on at least k edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least kn. Since every edge is incident to exactly two vertices, G must have at least $kn/2$ edges. That implies that if we pick an edge in G uniformly at random, the probability of picking an edge in the minimum cut is at most $2/n$. In other words, the probability that we don't screw up on the very first step is at least $1 - 2/n$.

Once we've contracted the first random edge, the rest of the algorithm proceeds recursively (with independent random choices) on the remaining $(n-1)$-node graph. So the overall probability $P(n)$ that GUESSMINCUT returns the true minimum cut is given by the recurrence

$$P(n) \geq \frac{n-2}{n} \cdot P(n-1)$$

with base case $P(2) = 1$. We can expand this recurrence into a product, most of whose factors cancel out immediately.

$$P(n) \geq \prod_{i=3}^{n} \frac{i-2}{i} = \frac{\prod_{i=3}^{n}(i-2)}{\prod_{i=3}^{n} i} = \frac{\prod_{j=1}^{n-2} j}{\prod_{i=3}^{n} i} = \boxed{\frac{2}{n(n-1)}}$$

13.3 Blindly Guessing Over and Over

That's not very good. Fortunately, there's a simple method for increasing our chances of finding the minimum cut: run the guessing algorithm many times and return the smallest guess. Randomized algorithms folks like to call this idea ***amplification***.

```
KargerMinCut(G):
  mink ← ∞
  for i ← 1 to N
      X ← GuessMinCut(G)
      if |X| < mink
          mink ← |X|
          minX ← X
  return minX
```

Both the running time and the probability of success will depend on the number of iterations N, which we haven't specified yet.

First let's figure out the probability that KARGERMINCUT returns the actual minimum cut. The only way for the algorithm to return the wrong answer is if GUESSMINCUT fails N times in a row. Since each guess is independent, our probability of success is at least

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^N \leq 1 - e^{-2N/n(n-1)},$$

by The World's Most Useful Inequality $1 + x \leq e^x$. By making N larger, we can make this probability arbitrarily close to 1, but never equal to 1. In particular, if we set $\boldsymbol{N = c\binom{n}{2}\ln n}$ for some constant c, then KARGERMINCUT is correct with probability at least

$$1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

When the failure probability is a polynomial fraction, we say that the algorithm is correct *with high probability*. Thus, KARGERMINCUT computes the minimum cut of any n-node graph in $\boldsymbol{O(n^4 \log n)}$ time with high probability.

If we make the number of iterations even larger, say $N = n^2(n-1)/2$, the success probability becomes $1 - e^{-n}$. When the failure probability is exponentially small like this, we say that the algorithm is correct with *very* high probability. In practice, very high probability is usually overkill; high probability is enough. (Remember, there is a small but non-zero probability that your computer will transform itself into a kitten before your program is finished.)

13.4 Not-So-Blindly Guessing

The $O(n^4 \log n)$ running time is actually comparable to some of the simpler flow-based algorithms, but it's nothing to get excited about. But we can improve our guessing algorithm, and thus decrease the number of iterations in the outer loop, by exploiting the observation we made earlier:

> *As the graph shrinks, the probability of contracting an edge in the minimum cut increases.*

At first the probability is quite small, only $2/n$, but near the end of execution, when the graph has only three vertices, we have a $2/3$ chance of screwing up!

A simple technique for working around this increasing probability of error was developed by David Karger and Cliff Stein.[3] Their idea was to group the first several random contractions a "safe" phase, so that the cumulative probability of screwing up is relatively small and a "dangerous" phase, which is much more likely to screw up.

The safe phase shrinks the graph from n nodes to about $n/2$ nodes, using a sequence of about $n/2$ random contractions.[4] Following our earlier analysis, the probability that *none* of these safe contractions touches the minimum cut is at least

$$\prod_{i=n/2+1}^{n} \frac{i-2}{i} = \frac{(n/2)(n/2-1)}{n(n-1)} = \frac{n-2}{4(n-1)} \approx \frac{1}{4}.$$

To get around the danger of the dangerous phase, we use amplification: we run the dangerous phase *four times* and keep the best of the four answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a recursion tree with degree 4, which expands as we get closer to the base case, instead of a single path. More formally, the algorithm looks like this:

```
CONTRACT(G, m):
  for i ← n downto m
      pick a random edge e in G
      G ← G/e
  return G
```

```
BETTERGUESS(G):
  if n < 1000
      use brute force
  else
      H ← CONTRACT(G, n/α)
      X1 ← BETTERGUESS(H, α)
      X2 ← BETTERGUESS(H, α)
      X3 ← BETTERGUESS(H, α)
      X4 ← BETTERGUESS(H, α)
      return min{X1, X2, X3, X4}
```

At first glance, it may look like we are performing exactly the same BETTERGUESS computation four times in a row, but remember that CONTRACT (and therefore BETTERGUESS) is randomized. Each recursive call to BETTERGUESS contracts a different, independently chosen random set of edges. Thus, X_1, X_2, X_3, and X_4 are almost always four completely different cuts!

Why four recursive calls, and not two or six? We're facing a tradeoff between the speed of the algorithm and its probability of success. More recursive calls makes the algorithm more likely to succeed but slower; fewer recursive calls makes the algorithm faster but *considerably* more likely to fail. Four recursive calls turns out to be the right balance, both guaranteeing a fast algorithm and a reasonably large success probability.

[3] David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

[4] I'm deliberately simplifying the analysis here to expose more intuition. More formally, if we contract from n to $\lceil n/2 \rceil + 1$ nodes, the probability that no minimum cut edge is contracted is strictly greater than $1/2$.

BETTERGUESS correctly returns the minimum cut unless (1) none of the edges of the minimum cut are CONTRACTed and (2) *all four* recursive calls return the incorrect cut for H. Let $P(n)$ denote the probability that BETTERGUESS returns a minimum cut of an n-node graph. Then for each index i, the probability that X_i is the minimum cut of H is $P(n/2)$. Because the four recursive calls are independent, we obtain the following recurrence for $P(n)$, with base case $P(n) = 1$ for all $n \le 8$.

$$\begin{aligned}
P(n) &= \Pr[\text{mincut in } H \text{ is mincut in } G] \cdot \Pr[\text{some } X_i \text{ is mincut in } H] \\
&\ge \frac{1}{4}\big(1 - \Pr[\text{no } X_i \text{ is mincut in } H]\big) \\
&\ge \frac{1}{4}\big(1 - \Pr[X_1 \text{ is not mincut in } H]^4\big) \\
&\ge \frac{1}{4}\Big(1 - \big(1 - \Pr[X_1 \text{ is mincut in } H]\big)^4\Big) \\
&\ge \frac{1}{4}\left(1 - \left(1 - P\left(\frac{n}{2}\right)\right)^4\right) \\
&= \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4
\end{aligned}$$

Using a series of transformations, Karger and Stein prove that $P(n) = \Omega(1/\log n)$. I've included a proof at the end of this note.

For the running time, we get a simple recurrence that is easily solved using recursion trees:

$$T(n) = O(n^2) + 4T\left(\frac{n}{2}\right) = O(n^2 \log n)$$

So all this splitting and recursing has slowed down the guessing algorithm slightly, but the probability of failure is *exponentially* smaller!

Let's express the lower bound $P(n) = \Omega(1/\log n)$ explicitly as $P(n) \ge \alpha/\ln n$ for some constant α. (Karger and Stein's proof implies $\alpha > 2$). If we call BETTERGUESS $N = c\ln^2 n$ times, for some new constant c, the overall probability of success is at least

$$1 - \left(1 - \frac{\alpha}{\ln n}\right)^{c\ln^2 n} \ge 1 - e^{-(c/\alpha)\ln n} = 1 - \frac{1}{n^{c/\alpha}}.$$

By setting c sufficiently large, we can bound the probability of failure by an arbitrarily small polynomial function of n. In other words, we now have an algorithm that computes the minimum cut with high probability in only $\boldsymbol{O(n^2 \log^3 n)}$ time!

*13.5 Solving the Karger-Stein recurrence

Recall the following recursive inequality for the probability that BETTERGUESS successfully finds a minimum cut of an n-node graph:

$$P(n) \ge \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4$$

Let $\overline{P}(n)$ be the function that satisfies this recurrence with equality; clearly, $P(n) \ge \overline{P}(n)$.

We can solve this rather ugly recurrence for $\overline{P}(n)$ through a series of functional transformations. First, consider the function $p(k) = \overline{P}(2^k)$; this function satisfies the recurrence

$$p(k) = \frac{1}{4} - \frac{1}{4}(1 - p(k-1))^4,$$

Next, define $d(k) = 1/p(k)$; this new function obeys the reciprocal recurrence

$$d(k) = \frac{4}{1-\left(1-\frac{1}{d(k-1)}\right)^4} = \frac{4\,d(k-1)^4}{d(k-1)^4-(d(k-1)-1)^4}$$

Straightforward algebraic manipulation[5] implies the inequalities

$$x < \frac{4x^4}{x^4-(x-1)^4} \le x+3$$

for all $x \ge 1$. Thus, as long as $d(k-1) \ge 1$, we have the simple recursive inequalities

$$d(k-1) < d(k) \le d(k-1)+3.$$

Our original base case $P(2) = 1$ implies the base case $d(1) = 1$. It immediately follows by induction that $d(k) \le 3k-2$, and therefore

$$P(n) \ge \overline{P}(n) = p(\lg n) = \frac{1}{d(\lg n)} \ge \frac{1}{3\lg n - 2} = \Omega\left(\frac{1}{\log n}\right),$$

as promised. Whew!

Exercises

1. (a) Suppose you had an algorithm to compute the minimum spanning tree of a graph in $O(m)$ time, where m is the number of edges in the input graph.[6] Use this algorithm as a subroutine to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

   (b) Describe and analyze an algorithm to compute the *maximum-weight edge* in the minimum spanning tree of a graph with m edges in $O(m)$ time. *[Hint: We can compute the median edge weight in $O(m)$ time. How do you tell quickly if that's too big or too small?]*

   (c) Use your algorithm from part (b) to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

2. Suppose you are given a graph G with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.

   (a) Describe and analyze an algorithm to select a random edge of G, where the probability of choosing edge e is proportional to the weight of e.

   (b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that GuessMinCut returns a minimum-weight cut is still $\Omega(1/n^2)$.

   (c) What is the running time of your modified GuessMinCut algorithm?

[5] otherwise known as Wolfram Alpha

[6] In fact, there *is* a randomized algorithm—due to Philip Klein, David Karger, and Robert Tarjan—that computes the minimum spanning tree of any graph in $O(m)$ *expected* time. The fastest deterministic algorithm known in 2015 runs in $O(m\alpha(m))$ time, where $\alpha(\cdot)$ is the inverse Ackermann function.

3. Prove that GUESSMINCUT returns the *second* smallest cut in its input graph with probability $\Omega(1/n^3)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. Consider the following generalization of the BETTERGUESS algorithm, where we pass in a real parameter $\alpha > 1$ in addition to the graph G.

```
BetterGuess(G, α):
  if n < 1000
      use brute force
  else
      H ← Contract(G, n/α)
      X1 ← BetterGuess(H, α)
      X2 ← BetterGuess(H, α)
      X3 ← BetterGuess(H, α)
      X4 ← BetterGuess(H, α)
      return min{X1, X2, X3, X4}
```

Assume for this question that the input graph G has a unique minimum cut.

(a) What is the running time of the modified algorithm, as a function of n and α? *[Hint: Consider the cases $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ separately.]*

(b) What is the probability that CONTRACT$(G, n/\alpha)$ does not contract any edge in the minimum cut in G? Give both an exact expression involving both n and α, and a simple approximation in terms of just α. *[Hint: When $\alpha = 2$, the probability is approximately $1/4$.]*

(c) Estimate the probability that BETTERGUESS(G, α) returns the minimum cut in G, by adapting the solution to the Karger-Stein recurrence. *[Hint: Consider the cases $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ separately.]*

(d) Suppose we iterate BETTERGUESS(G, α) until we are guaranteed to see the minimum cut with high probability. What is the running time of the resulting algorithm? For which value of α is this running time minimized?

(e) Suppose we modify BETTERGUESS(G, α) further, to make k recursive calls instead of four. What is the best choice of α, as a function of k? What is the resulting running time? What is the best choice for k and α?