16.1 Karger's algorithm

We are given an undirected, unweighted graph $G = (V,E)$ with $|V| = n, |E| = m$. A *cut* of G is a partition of the vertices into two non-empty sets. The weight $w(C)$ of a cut $C = (S,T)$ is the number of edges crossing the cut (i.e. $w(C) = |\{e = (u,v) \in E : u \in S, v \in T\}|$. A *minimum cut* of G is a cut achieving the minimum weight over all cuts of G. The algorithmic problem we consider is thus as follows: given G, compute a minimum cut of G.

One motivation for studying minimum cut is that it is some measure of the reliability of a network. Imagine that each vertex represents a network switch, and edges corresond direct links between switches. Let us assume our graph is connected (which hopefully our communication network is!). Then the weight of a minimum cut answers the following question: *what is the minimum number of links that need to go down in our network to make our graph disconnected, i.e. so that not every switch can still talk to every other switch*? The minimum cut doesn't deal with switches themselves failing (i.e. vertex failures), but that's a problem we won't consider today.

There is an obvious brute-force algorithm for the minimum cut problem: try all cuts then take the one with the minimum weight. There are $2^{n-1} - 1$ cuts to try. To see this, write $V = \{1,\ldots,n\}$ and put cuts in correspondence with length-$(n-1)$ binary strings. The ith bit (1-indexed) is 1 if vertex $i+1$ is on the same side of the cut as vertex 1. There are 2^{n-1} such strings, but the all-1s string is not allowed since that would mean one side of the cut would be empty.

A more efficient Monte Carlo algorithm, due to David Karger (a Harvard alum '89!), is the following. It is known as Karger's contraction algorithm. In what follows, G may be a multigraph (i.e. there may be multiple parallel edges between the same two vertices; even if G doesn't start this way, it may become this way in later recursive calls).

```
procedure CONTRACT($G = (V,E)$)
    if $|V| = 2$ then output the only cut
    else
        pick a random edge $e$
        contract the edge $e$ to form $G' = (V',E')$
        return CONTRACT($G'$)
end
```

What does it mean to *contract* an edge $e = (u,v)$? It means we remove the vertices (u,v) from the graph and insert a new vertex w. Any edge in G of the form (u,x) for $x \neq v$ is replaced by the edge (w,x). Similarly any edge of the form (v,x) for $x \neq u$ is replaced by (w,x). All edges of the form (u,v) are removed. Figure 16.1 gives an example of obtaining some G' by contracting an edge in G. Essentially when we contract (u,v), we are saying that we are promising to place u,v on the same side of the final cut we output. Thus when $|V| = 2$ in the last level of recursion, the two vertices actually represent the vertices placed into the two sides of the cut.

Lemma 16.1 *Let C be some minimum cut of G with $w(C) = k$. Then the probability that Karger's contraction algorithm outputs C is at least $1/\binom{n}{2}$.*

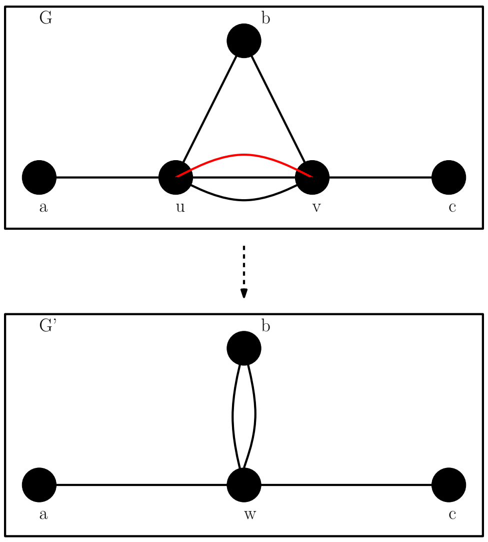

Figure 16.1: Contracting one of the (u,v) edges highlighted in red to merge u,v into w.

Proof: We say C survives the ith level of recursion if no edge in C is contracted in recursive level i. Then C is output as long as C survives every level. Thus

$$\mathbb{P}(C \text{ is output}) = \mathbb{P}(C \text{ survives every recursive level}) = \prod_{i=0}^{|V|-2} \mathbb{P}(C \text{ survives level } i | C \text{ survived levels } 0,1,\ldots,i-1)$$

where $\mathbb{P}(A|B)$ is the conditional probability of event A, given that event B occurred. Let m_i be the number of edges in our current graph G_i in recursive level i (note this graph is a multigraph: there can be parallel edges between vertices). Then the probability C survives recursive level i conditioned on it having already survived up until this point is $1-|C|/m_i = 1-k/m_i$. Since C has survived up until this point, the minimum cut of G_i is also still k. Thus the minimum degree of any vertex in G_i is at least k, since otherwise the cut separating a low-degree vertex from everyone else would have smaller cut value. Since m_i is just half the sum of all degrees, we have $m_i \ge n_i k/2 = (n-i)k/2$. Thus $1-k/m_i \ge 1-2/(n-i)$. Thus

$$\mathbb{P}(C \text{ is output}) \ge \left(1-\frac{2}{n}\right)\cdot\left(1-\frac{2}{n-1}\right)\cdot\ldots\cdot\frac{1}{3} = \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdot\frac{n-5}{n-3}\cdot\ldots\cdot\frac{3}{5}\cdot\frac{2}{4}\cdot\frac{1}{3} = \frac{2}{n(n-1)}. \quad (16.1)$$

due to cancellation. ■

You might ask: can we improve the $1/\binom{n}{2}$ success probability? The answer is no as the lemma is stated, since if there are t minimum cuts then one of them will be output with probability at most $1/t$. We can obtain $t=\binom{n}{2}$ by letting G be the cycle on n vertices (it has $\binom{n}{2}$ minimum cuts: namely choose any 2 of its n edges to cut).

Now, $1/\binom{n}{2}$ success probability seems low, but we can alleviate this by running the algorithm $\binom{n}{2}$ times and outputting the smallest weight cut we ever find. Then the probabilty that we never see a particular minimum cut is at most $(1-1/\binom{n}{2})^{\binom{n}{2}} \le 1/e$, using our favorite approximation $1+x\le e^x$ for all real x. Thus if we repeat this $\lceil \ln(1/P)\rceil$ times for some $0<P<1/2$, the probability of never seeing a minimum cut is at most $e^{-\ln(1/P)}=P$.

We also leave it as an exercise to show that the contraction algorithm above can be implemented to run in time $O(n^2)$.

16.1.1 Karger-Stein

Shortly after Karger's contraction algorithm described above, Karger and Stein developed a variant with improved running time. The key observation comes from staring at (16.1). Notice that in the first round of contraction the probability that C survives is pretty good: at least $1-2/n$. Meanwhile toward the end, the probability C survives starts getting much farther away from one: we can only say it is at least $1/3$ in the last iteration.

The idea of Karger and Stein was to run contraction not until we get down to two vertices, but rather until we get down to t vertices. Then the probability that C has still survived up until this point, by (16.1), is at least $t(t-1)/(n(n-1))$. By picking $t = \lceil n/\sqrt{2}\rceil + 1$, we have that the probability that C survives after contracting down to t vertices is at least $1/2$. The Karger-Stein algorithm, henceforth known as KS, is then as follows:

```
procedure KS($G = (V, E)$)
    if $|V| < 6$ then try all $2^{|V|-1}$ cuts and return the smallest one
    else
      $t \leftarrow \lceil n/\sqrt{2}\rceil + 1$
      Obtain $G_1$ by contracting $G$ $n-t$ times to obtain $t$ vertices
      Obtain $G_2$ by contracting $G$ $n-t$ times to obtain $t$ vertices (using independent randomness from last step)
      $C_1 \leftarrow KS(G_1)$
      $C_2 \leftarrow KS(G_2)$
      return the smaller of the two cuts $C_1$ and $C_2$
end
```

The reason we make $|V| < 6$ a special case is that $\lceil n/\sqrt{2}\rceil + 1 \geq n$ for $n < 6$, and thus the recursive calls in the else statement would not actually shrink the problem size for $n < 6$.

First, what is the running time of the above algorithm? We mentioned that the original contraction algorithm can be implemented to run in time $O(n^2)$. Thus the running time $T(n)$ of KS satisfies the recurrence

$$T(n) \leq 2T(\lceil n/\sqrt{2}\rceil + 1) + O(n^2).$$

The master Master theorem then implies $T(n) = O(n^2 \log n)$.

The key benefit though will be that KS has much better success probability. Whereas the vanilla contraction algorithm had success probability at least $1/\binom{n}{2}$, we will now show that KS has success probability at least $\Omega(1/\log n)$.

Lemma 16.2 *KS outputs a minimum cut with probability* $\Omega(1/\log n)$.

Proof: Let p_k be a lower bound on the probability that KS returns a minimum cut on its input at the kth level of recursion, where $k = 0$ corresponds to the base case $n < 6$. Then clearly $p_0 = 1$ is a valid lower bound since we try all cuts. What about p_{k+1}? By our choice of t, the probability that G_1 still contains a minimum cut of G is at least $1/2$ (and similarly for G_2). Thus the probability that C_1 is a minimum cut of G is at least $(1/2) \cdot p_k$. The same is true for C_2. Since G_1 and G_2 are independent, the probability that *neither* C_1 nor C_2 is a minimum cut of G is thus at most $(1-(1/2)p_k)^2$. Therefore, a valid lower bound on the probability that KS returns a minimum cut at the $(k+1)$st level of recursion is

$$p_{k+1} = 1 - (1-(1/2)p_k)^2 = p_k - (1/4)p_k^2. \tag{16.2}$$

Define new variables z_k by $p_k = 4/(z_k+1)$. Then $z_0 = 3$ and a substitution into (16.2) shows that $z_{k+1} = 1 + z_k + 1/z_k$. Induction on k then implies that $z_k \geq k$ for all $k \geq 0$. Then, given this, induction on k then implies $z_k \leq 3 + 2k$ for all $k \geq 0$. Notice the highest level of recursion of interest is some $k^* = O(\log n)$, since in each recursive call n decreases by a factor of roughly $\sqrt{2}$ (which can only happen $\log_{\sqrt{2}} n$ times). Thus $z_{k^*} = O(\log n)$; remembering the definition of z_k, this implies p_{k^*}, the success probability at the root of the recursion tree, is $\Omega(1/\log n)$. ■

Given the above lemma, as with the analysis for the vanilla contraction algorithm, we see that it suffices to run KS $\Theta(\log n \log(1/P))$ times to have success probability $1-P$. Since the KS running time itself is $O(n^2 \log n)$, taht implies that the total running time to obtain success probability $1-P$ is $O(n^2(\log n)^2 \log(1/P))$.

16.2 2SAT

Previously, we used the implication graph and strongly connected components to determine whether there exists a solution to 2SAT. Here, we will show another possible way to solve the 2SAT problem.

Recall that the input to 2SAT is a logical expression that is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_1}).$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied– that is, there is at least one true literal in each clause. For example, the assignment $x_1 = T, x_2 = F, x_3 = F, x_4 = T$ satisfies the 2SAT formula above.

Here is a simple randomized solution to the 2SAT problem. Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied or you get tired of flipping coins.

In the example above, when we begin with all variables set to F, the clause $(x_1 \vee x_2)$ is not satisfied. So we might randomly choose to set x_1 to be T. In this case this would leave the clause $(x_4 \vee \overline{x_1})$ unsatisfied, so we would have to flip a variable in the clause, and so on.

Why would this algorithm tend to lead to a solution? Let us suppose that there is a solution, call it S. Suppose we keep track of the number of variables in our current assignment A that match S. Call this number k. We would like to get to the point where $k = n$, the number of variables in the formula, for then A would match the solution S. How does k evolve over time?

At each step, we choose a clause that is unsatisfied. Hence we know that A and S disagree on the value of at least one of the variables in this clause– if they agreed, the clause would have to be satisfied! If they disagree on both, then clearly changing either one of the values will increase k. If they disagree on the value one of the two variables, then with probability 1/2 we choose that variable and make increase k by 1; with probability 1/2 we choose the other variable and decrease k by 1.

Hence, intuitively, in the "worst case", k behaves like a *random walk*– it either goes up or down by 1, randomly. This leaves us with the following question: if we start k at 0, how many steps does it take (on average, or with high probability) for k to stumble all the way up to n, the number of variables?

We can formalize this idea of "worst case" as follows: let y be a random walk on $[0,n]$, and suppose that k and y are changing in parallel and sharing the same random coins. In particular, y goes left or right with probability $1/2$, whereas k either does the same as y (i.e. the clause chosen disagrees on one variable) or goes right (i.e. the clause chosen disagrees onb oth variables). The only exception is if either variable is at 0, in which case we immediately move it to 1 *before* simulating the next step.

Then, X and Y be random variables. X will be the number of steps in our simulation before we find a satisfying assignment (assuming one exists), plus the number of times k is moved from 0 to 1. Y will be the number of steps for y to reach n, plus the number of times that y is moved from 0 to 1.

We see that X simulates the number of rounds for our algorithm to find a satisfying assignment. Then, we want to say that $\mathbb{E}X \leq \mathbb{E}Y$ so that we can use $\mathbb{E}Y$ to get an upper bound on $\mathbb{E}X$.

Why is this true? In particular, we can check by induction on the number of steps 0 that $k \geq y$. The base case is simply our starting scenario, where $k = y = 1$ (after the move). For the inductive step, assume that $k \geq y$ at the end of the previous step. Then, if k moves left, so must y.

- If neither is moved right from 0 immediately after the next step, then we immediately obtain that $k \geq y$ at the end of that step.
- Otherwise, for a move right from 0 at the end of the next step to be possible, at least one of k and y must have been equal to 1 in the previous step.
  - If $k = 1$, then $y = 1$ as well because of our inductive hypothesis. If k moves left, so must y, and they both end up at 1 again after having been moved away from 0. If k moves right, then $k \geq y$ at the end of the next step regardless of whether y moves right or left.
  - If $y = 1$ but $k \neq 1$, then if both move right it still holds that $k \geq y$. Otherwise, if k moves left, then k will end up back at 1 (after having been moved there from 0). Meanwhile, because k was at least 2, it must be at least 1 after a move.

In all cases, we see that $k \geq y$ still after another step.

Hence, because $k \geq y$ after any number of steps, then k must reach n before y or at the same time. As a result, for any fixed set of coin flips, $X \leq Y$, so when we take the expectation over all coin flips, it must therefore be true that $\mathbb{E}X \leq \mathbb{E}Y$.

Now, note that Y is just equal to the number of steps for a random walk to get from 0 to n, where we remove the automatic move from 0 to 1 (but if $y = 0$, it is still only allowed to move right).

We can now check that $\mathbb{E}Y = n^2$. In fact, we claim that the average amount of time to walk from i to n is $n^2 - i^2$. Note that the time average time $T(i)$ to walk from i to n is given by:

$$\begin{aligned} T(n) &= 0 \\ T(i) &= \frac{T(i-1)}{2} + \frac{T(i+1)}{2} + 1,\ i \geq 1 \\ T(0) &= T(1) + 1. \end{aligned}$$

These equations completely determine $T(i)$, and our solution satisfies these equations!

Hence, on average, we will find a solution in at most n^2 steps. (We might do better– we might not start with all of our variables wrong, or we might have some moves where we must improve the number of matches!)

We can run our algorithm for say $100n^2$ steps, and report that no solution was found if none was found. This algorithm might return the wrong answer– there may be a truth assignment, and we have just been unlucky. But most of the time it will be right: Markov's inequality gives that if we run the algorithm for cn^2 steps for $c \geq 1$, then it will be wrong at most $1/c$ of the time.