CSE525: Randomized Algorithms and Probabilistic Analysis 04/04/2013

Lecture 2

Lecturer: Anna Karlin *Scribe: Sam Hopkins, James Youngquist*

1 Derandomizing via the Method of Conditional Expectation

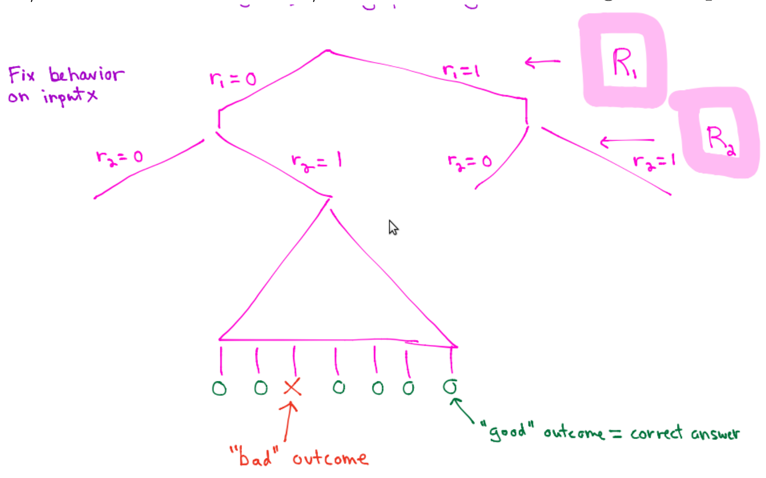
[This section follows the presentation in Vadhan, *Pseudorandomness*.]

Consider a randomized algorithm A that uses m random bits, which we consider to be random variables $R_1, \ldots, R_m$, and let us suppose that A is successful at least $2/3$ of the time—that is, for every input x,

$$\Pr_{R_1,\ldots,R_m}[A(x, R_1, \ldots, R_m) \text{ is "good"}] \geq \frac{2}{3}.$$

Suppose we want to derandomize A—that is, give a deterministic variant of A which succeeds with probabilty 1 on every input. Sometimes we can do this using the *method of conditional expectation*.

We can think of A as a binary tree which, given x, branches on the sampled value of each random bit R_i in turn. Paths in this tree correspond to different possible random strings $R_1, \ldots, R_m$ that could be sampled by A, and leaf nodes at level $m+1$ are labeled by outputs. The fact that A succeeds with probability at least $2/3$ means that at least $2/3$s of the leaf nodes are good outputs for the input x.



The idea is now to produce a deterministic algorithm that traverses this tree from the root to a leaf, which, when encountering the branch at level i, (corresponding to the point at which the algorithm uses R_i) chooses a direction (i.e. a value for R_i) which leads to a good output. What criterion can we use to choose a good direction?

Where $r_1, \ldots, r_i \in \{0, 1\}$ corresponds to a partial root-leaf path in the tree terminating at node n, let $P(r_1, \ldots, r_i)$ be the fraction of leaves in the subtree below n which are good outputs. Formally,

$$\begin{aligned} P(r_1, \ldots, r_n) &= \Pr[A(x, R_1, \ldots, R_m) \text{ is good} | R_1 = r_1, \ldots, R_i = r_i] \\ &= \frac{1}{2}P(r_1, \ldots, r_i, 0) + \frac{1}{2}P(r_1, \ldots, r_i, 1) \end{aligned}$$

where the last equality follows because the subtrees below n's children have equally-many nodes. It follows immediately that there is a choice $r_{i+1} \in \{0, 1\}$ for R_{i+1} so that $P(r_1, \ldots, r_{i+1}) \geq P(r_1, \ldots, r_i)$. To find a good path in the tree, then, it suffices at each branch to pick such an $r \in \{0, 1\}$. Then at the end of the walk,

$$P(r_1, \ldots, r_m) \geq P(r_1, \ldots, r_{m-1}) \geq \ldots \geq P(r_1) \geq \Pr[A(x, R_1, \ldots, R_m) \text{ is good}] \geq \frac{2}{3}.$$

Since $P(r_1, \ldots, r_m)$ is either 0 or 1, it must be 1.

In order to make this work, we need to be able to deterministically and efficiently calculate $P(r_1, \ldots, r_i)$. This cannot always be done, but it will actually work for our randomized Max-Cut algorithm.

1.1 Derandomizing Max-Cut

Let's recall our simple randomized algorithm for Max-Cut. Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$, the algorithm is as follows:

```
1. Let $S, \overline{S} = \emptyset$.
2. for $i := 1$ to $m$:
    (a) Let $R_i$ be chosen uniformly at random from $\{0, 1\}$.
    (b) if $R_i = 1$ then put $v_i$ in $S$ else put $v_i$ in $\overline{S}$.
3. Output the cut $(S, \overline{S})$.
```

We can derandomize this using the method of conditional expectations. Instead of maximizing the expected number of good outputs at each branching point, we will maximize the expected size of the resulting cut at each branching point i, where the expectation is over the random bits $R_{i+1}, \ldots, R_m$ which have yet to be set.

We now treat S as a random variable, dependent on the random variables R_i. Let $cut(S, \overline{S})$ (also a random variable) denote the size of the $(S, \overline{S})$ cut. Define

$$C(r_1, \ldots, r_i) = E[cut(S, \overline{S}) | R_1 = r_1, \ldots, R_i = r_i].$$

Note that

$$C(\emptyset) = E[cut(S, \overline{S})] = \frac{|E|}{2}.$$

By the preceeding analysis, we if we can compute $C(r_1, \ldots, r_i)$ efficiently, we can give a deterministic algorithm which finds a cut of size at least $|E|/2$.

Let us consider the state of the world after choice i is made. Let $S_i = \{v_j | j \leq i, R_j = 1\}$ and $\overline{S_i} = \{v_j | j \leq i, R_j = 1\}$ and $U_i = \{v_{i+1}, \ldots, v_n\}$. Observe first that

$$C(r_1, \ldots, r_i) = |cut(S_i, \overline{S}_i)| + \frac{1}{2}|\text{edges with at least one endpoint in } U_i|.$$

The first term appears because edges already in the cut will not ever be removed, and the second because edges since edges not already in the cut have a 1/2-chance of ending up in the cut.

In order to make a choice at branch point i, we need to compare $C(r_1, \ldots, r_i, 0)$ with $C(r_1, \ldots, r_i, 1)$. By the above,

$$C(r_1, \ldots, r_{i+1}) = |cut(S_{i+1}, \overline{S}_{i+1})| + \frac{1}{2}|\text{edges with at least one endpoint in } U_{i+1}|.$$

However, the quantity $|$edges with at least one endpoint in $U_{i+1}|$ does not depend on R_{i+1}—no matter what, v_{i+1} is being removed from U_{i+1}. So in order to make a choice for r_{i+1}, it suffices to maximize $cut(S_{i+1}, \overline{S}_{i+1})$. Slightly more formally,

$$cut(S_{i+1}, \overline{S}_{i+1}) = cut(S_i, \overline{S}_i) + \begin{cases} |\text{edges from } v_{i+1} \text{ to } S_i| \text{ if } r_{i+1} = 0 \\ |\text{edges from } v_{i+1} \text{ to } \overline{S}_i| \text{ if } r_{i+1} = 1 \end{cases}$$

To maximize, simply pick the larger of the two possibilites in the right-hand term—this is the greedy algorithm!

Corollary 1. *The greedy algorithm for Max-Cut finds a cut of size at least $|E|/2$.*

2 Randomized Min-Cut

Given an undirected multigraph $G = (V, E)$, a Min-Cut of that graph is a partioning of the vertices into two sets, S and $\bar{S}$, with the least number of edges connecting the vertices of those sets of all possible cuts of G. Equivalently, a Min-Cut is a minimal set of edges needed to disconnect G. There could be more than one Min-Cut for a given graph.

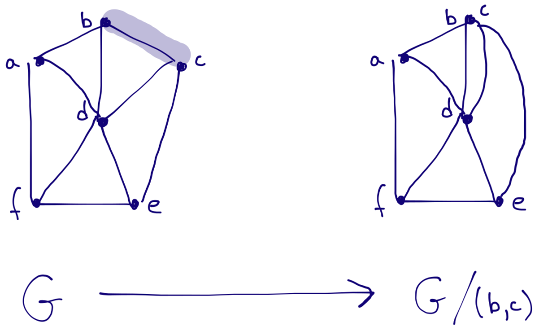
There exist deterministic algorithms for finding the Min-Cut. A naive ways is find $\min_c\{c|c \text{ is an } s{-}t \text{ cut}\}$ with runtime $\tilde{O}(n^2m)$. The faster Hao-Orlin algorithm [1] can do it in $\tilde{O}(nm)$.

Here we examine a particular idea for building a randomized Min-Cut algorithm, first using it naively and then with some additional cleverness to get an algorithm with runtime $\tilde{O}(n^2)$ and probability of failure of $O1/n^k)$.

(Note that the $\tilde{O}$ notation indicates that constant and logarithmic factors are dropped from consideration.)

2.1 A First Try

The algorithmic heart of the idea is this: given a graph G, in general the Min-Cut is a "small" set of edges in G. Thus, an edge selected uniformly at random in G is unlikely to be in the Min-Cut, so if we decide to force that edge out of the cut (by forcing its endpoint vertices to be on the same side of the cut) we are unlikely to have made much of an error. Thus, we will simply contract edges (merging adjacent vertices) until only two vertices remain, forming a graph G' with two "supernodes". Note that we allow for there to be more than one edge between two vertices after a contraction operation. Once again, the key observation is that when we contract an edge that does not belong to some chosen Min-Cut, the resulting post-contraction graph is smaller, but has the same Min-Cut as G.



The last two remaining vertices represent a partitioning of G – all the vertices in S have been collapsed and likewise for $\bar{S}$. The remaining edges in G' form a cut in G.

```
input  : $G$ a graph
while $G$ has more than 2 vertices do
    Choose an edge $(u, v)$ uniformly at random;
    $G \leftarrow G/(u, v)$;
end
output: unique cut defined by contracted graph
```
Algorithm 1: Random Min-Cut 1

[1] http://dl.acm.org/citation.cfm?id=139439

2.2 Analysis of the Simple Algorithm

As the size of the contracted graph decreases, it becomes ever more likely that we do err and contract away an edge in the Min-Cut.

Lemma 2. *A particular Min-Cut is returned by Algo 1 with prob* $\geq 1/\binom{n}{2}$.

Proof Consider a particular Min-Cut with c edges in it.

$$P(\text{select an edge in this cut}) = \frac{c}{m}$$

. We claim this is $\leq \frac{2}{n}$. Since the size of the Min-Cut is c and the edges indicent to any particular vertex form a cut, every vertex must have degree at least c. Hence, the graph has at least $\frac{nc}{2}$ edges.

When r vertices remain, there must be at least $\frac{rc}{2}$ edges left, again because the minimum degree of G is c and contraction operations do not ever decrease degree. The probability of choosing a edge in our fixed cut when r vertices remain, assuming that the cut has not already been contracted away, is thus at most

$$\frac{c}{\frac{rc}{2}} = \frac{2}{r}.$$

So we get that

$$\begin{aligned} P(\text{find Min-Cut}) &= P(\text{never contract Min-Cut edge}) \\ &= P(\text{don't on } 1^{st} \text{ contraction}) \cdot P(\text{don't on } 2^{nd} | \text{didn't on } 1^{st}) \cdots \\ &\leq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{3}\right) \\ &= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\ &= \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \geq \frac{1}{n^2} \end{aligned}$$

■

To boost the probability of finding a Min-Cut, we simply repeat the algorithm and return the smallest cut found across many iterations. If we want inverse-polynomial probability of failure, then we repeat $n^2 \ln n$ times. Since each trial is independent of the others,

$$P(\text{no iteration returns Min-Cut}) \leq \left(1 - \frac{1}{n^2}\right)^{n^2 \ln n}$$

From the most useful approximation ever (see appendix), we get that

$$\left(1 - \frac{1}{n^2}\right)^{n^2 \ln n} \leq e^{-\ln n} = \frac{1}{n}$$

and the total running time is $O(n^4 \ln n)$.

Unfortunately, this running time does not beat the deterministic algorithms referenced above, but this bound already yields nice results (see Lecture 3 for some nice combinatorial corollaries).

2.3 Improving the Runtime

Is it possible to improve this algorithm to do better than the deterministic algorithms mentioned above?

Observe that the probability of failure is much higher in later iterations than at the beginning because the ratio of "good" edges to "bad" is much higher. That is, we're more likely to pick a Min-Cut edge as the number of edges decreases. However, in the method given above, we repeat the low-chance-of-failure stages just as much as the high-chance-of-failure ones. We would like to fix this, repeating the high-failure stages more than the low-failure ones.

We will employ a recursive branching algorithm that chooses the best Min-Cut of two recursive calls on a reduced graph. Because each contraction step chooses a random edge, the ith contraction step will be repeated some number of times exponential in i, so most of the repetitions happen in lower levels of the call graph where there are fewest vertices remaining in the graph. See Algo. 2

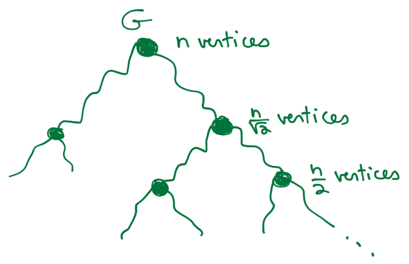
The question is how to decide how much to contract before branching and recursing. If we contract from n to k vertices (which we will write $n \to k$), the probability we contract an edge in the Min-Cut is

$$P(\text{bad contraction when } n \to k) = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{k+1}{k+3}\right)\left(\frac{k}{k+2}\right) \approx \frac{k^2}{n^2}$$

So if we choose $k = n/\sqrt{2}$ then $P(\text{no failure for } n \to k) \leq 1/2$. Branching after $n \to n/\sqrt{2}$ will yield an algorithm with easily-boostable success probability. Here's the algorithm:

```
input : $G$ a graph
input : $n$ number of vertices in $G$
if $G$ has 2 vertices then
    return: weight of (unique) Min-Cut
end
else
    $G_1 \leftarrow$ Contract $G(n \to \frac{n}{\sqrt{2}})$ vertices ;
    RecursiveContract($G_1$, $\frac{n}{\sqrt{2}}$) ;
    $G_2 \leftarrow$ Contract $G(n \to \frac{n}{\sqrt{2}})$ vertices ;
    RecursiveContract($G_2$, $\frac{n}{\sqrt{2}}$) ;
    return: $\max(G_1, G_2)$
end
```

Algorithm 2: Recursive Contract



2.4 Analysis of the Improved Algorithm

What is the running time of Recursive Contract? What is its probability of success?

Running Time Recursive Contract's running time clearly satisfies the recurrence

$$T(n) \leq 2(n^2 + T(n/\sqrt{2})).$$

It follows easily from the master theorem that the runtime is therefore $O(n^2 \log n) = \tilde{O}(n^2)$. Thus the branching has slowed down the algorithm slightly, but hopefully the error probability is such that we need many fewer repitions to have an appropriate success rate.

Probability of Success Recursive Contract induces a perfect binary tree where internal nodes are branching points and leaves represent cuts of G which are attempts at min-cuts. Fix a particular min cut.

For an internal node n, define p_n to be the probability that there is a leaf node corresponding to our fiexed min cut in the subtree rooted at n, given that no edge of that min cut has been collapsed when the algorithm reaches node n. By symmetry, if p_n and $p_{n'}$ are of the same depth then $p_n = p_{n'}$, so define p_d to be p_n for n of depth d.

By definition, $p_0 = 1$ (where the 0-depth nodes are the leaves). By our above analysis, the probability of contracting a good edge (i.e. one in our fixed min cut) in one contraction step is at most $1/2$, so the probability that one branch of a computation at level d succeeds is at least $1 - \frac{1}{2}p_{d-1}$. Hence, the probability of success at at level d is given by

$$p_d = 1 - \Pr[\text{both children fail}] = 1 - \left(1 - \frac{1}{2}p_{d-1}\right)^2.$$

Claim 3.

$$p_d \geq \frac{1}{d+1}.$$

Proof The proof is by induction. In the base case, $p_0 = 1 \geq 1/2$. For the induction step, we can expand our previous analysis of p_d as follows:

$$\begin{aligned} p_d &= 1 - \Pr[\text{both children fail}] = 1 - \left(1 - \frac{1}{2}p_{d-1}\right)^2 \\ &= 1 - (1 - p_{d-1} + \frac{p_{d-1}^2}{4}) \\ &= p_{d-1} - \frac{p_{d-1}^2}{4}. \end{aligned}$$

Then by induction this bounded from below by

$$\frac{1}{d} - \frac{1}{4d^2}$$

which, because for all $d \geq 1$ it is the case that $(4d^2)^{-1} \leq (d(d+1))^{-1}$, is greater than

$$\frac{1}{d} - \frac{1}{d(d+1)} = \frac{1}{d}\left(\frac{d+1-1}{d+1}\right) = \frac{1}{d+1}$$

as desired. ■

It remains to find (an upper bound on) the total depth of the tree. At depth i in the tree, there are $n/(2^{i/2})$ nodes left in the graph. We will get an upper bound on the depth if we think of the algorithm as running till there is just one node left—that is, we want $2^{d/2} = n$. This occurs when $d = 2\log n$.

Thus, the probability of success is at least $p_{2\log n} = \Omega(1/\log n)$. This is exponentially better than our naive (non-branching) contraction algorithm, so boosting it is much easier.

Suppose we want to get failure probability ϵ. How many times to we have to run Recursive Contract? That is, we need k such that

$$\left(1 - \frac{1}{\log n}\right)^k \leq \epsilon.$$

By the most useful approximation ever, this occurs when

$$e^{-k/\log n} \le \epsilon$$

which (dropping a constant, since this will all go in to a big-O analysis) occurs when

$$\frac{k}{\log n} \ge \log \epsilon^{-1},$$

or, rewriting, when

$$k \ge \log \epsilon^{-1} \log n.$$

If we want inverse polynomial error, for example $\epsilon = 1/n$, then we can take $k = (\log n)^2$. If we want error at most n^{-c}, we just get an additional multiplicative factor of c.

Thus, we have proved:

Theorem 4 (Karger, Karger/Stein)**.** [2] *There is a randomized Monte Carlo algorithm for min-cut that runs in time $\tilde{O}(n^2)$. It fails to output a min cut with probability $O(n^{-k})$.*

What is a Monte Carlo algorithm? Read on:

Monte Carlo vs Las Vegas Algorithms This is bizarre terminology for the following notions. Randomness in an algorithm can affect both the runtime and the correctness. We distinguish between two kinds of randomized algorithms: *Monte Carlo* algorithms have fixed (deterministic) runtime but a small probability of failure, and *Las Vegas* algorithms always give the correct answer but have a random variable for their running time.

Note that we can convert a Las Vegas algorithm which runs in time $f(n)$ with high probability into a Monte Carlo algorithm by stopping after $f(n)$ time and guessing randomly if no answer has been output. However, there is no (known) general way to convert a Monte Carlo algorithm to a Las Vegas one, the issue being that there is no known way to check the correctness of the answer output by the Monte Carlo algorithm.

3 Appendix: $1 - x \le e^{-x}$ a.k.a. the Most Useful Approximation Ever

We offer in this section a proof of

Theorem 5. *For all $x \in \mathbb{R}$, $1 - x \le e^{-x}$.*

Proof Note that since the second derivative of e^x is just e^x which is everywhere positive, e^x is everywhere convex. Thus, for any point x_0 it lies above its linear approximation at x_0. That is,

$$e^x \ge e^{x_0} + e^{x_0}(x - x_0).$$

With $x_0 = 0$ this gives

$$e^x \ge 1 + (x - 0) = 1 + x.$$

A trivial inversion of sign yields

$$e^{-x} \ge 1 - x$$

as desired. ■

Corollary 6. *For all $x, y \in \mathbb{R}$, $(1 - x)^y \le e^{-xy}$.*

[2]http://dl.acm.org/citation.cfm?id=234534