ORIE 4520: Stochastics at Scale **Fall 2015**

Unit 1

Instructor: Sid Banerjee (sbanerjee@cornell.edu) *Date: August 26*

This unit covers lectures 1 through 5

1.1 Topics in this unit

In this first set of lectures, we revised some basic probabilistic tools, and saw how they can be used for designing fast algorithms for some applications.

Tools and concepts

- (Use of asymptotic notation – O, Ω and Θ)
- Linearity of expectation
- Indicator random variables
- Law of total probability (and the 'Principle of deferred decisions')

Models and Applications

- Balls and Bins - Here, we looked at a bunch of different problems:
  - Filling all bins (the coupon-collector problem)
  - Collisions (the birthday paradox)
  - (Later lecture: Maximum loaded bin)
- Matrix multiplication checking (Freivald's algorithm)
- Sorting (Quicksort)
- Max-Cut (1/2 approximation via sampling)
- Min-Cut (the basic CONTRACT algorithm, and the FASTCUT algorithm)

1.2 Basic tools from probability

The most important thing you learnt in this lesson was the use of the linearity of expectations + indicator random variables.

Proposition 1 (Linearity of Expectation). *Given* any arbitrary collection *of random variables* $X_1, X_2, \ldots, X_n$*, we have:*

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

Definition 1 (Indicator Random Variable). *For any random event* E*, the random variable* $X = \mathbb{1}_{\{E\}}$ [1] *is called the* indicator random variable *of* E*. In particular, we have:*

$$\mathbb{E}[X] = \mathbb{P}[E]$$

[1] $\mathbb{1}_{\{E\}}$ is a common notation for '1 if E is true, 0 otherwise.

The other very useful identity that we will use repeatedly is the law of total probability:

Proposition 2 (Law of Total Probability). *Given $\{B_i; i = 1, 2, 3, \ldots\}$, a finite or countably infinite partition of a sample space Σ (i.e., B_i are* mutually exclusive *events whose union is the entire sample space), then for any event A in the same probability space:*

$$\mathbb{P}[A] = \sum_i \mathbb{P}[A|B_i]\mathbb{P}[B_i]$$

The most common way in which we will use this is via the *principle of deferred decisions* – essentially, if an event is determined by several random variables, then it is often easier to analyze the probability of the event by fixing some of the variables, while allowing the rest to be random (in particular, if we can show that the probability is the same, or has the same bound, irrespective of the values of the fixed variables). The events B_i will now correspond to all the different possibilities for the fixed variables.

1.3 Verifying matrix multiplication

The first randomized algorithm we studied was for testing matrix multiplication. Given 3 $n \times n$ matrices A, B and C, we want to check if:

$$AB = C?$$

Intuitively, we need $\Omega(n^2)$ time at least, as that is the time it takes to even look at every element of C. The 'brute-force' way to check is by verifying that $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$ for each index (i, j) – the best we can say about this, however, is that it takes n^3 time (there are n^2 indices (i, j), and for each we are multiplying 2 vectors, which requires n operations). There are more sophisticated ways of doing matrix multiplication – however even then, for the current fastest algorithm (by Coppersmith and Winograd), the best we can say is that it is $O(n^{2.3728659})$ – still some way off from the lower bound.

However, using randomization, we can get around this!

1.3.1 Freivalds' Algorithm

To make the argument easier, we henceforth work in base 2 – i.e., we want to check if $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} \pmod 2$.

For the runtime, note that each matrix-vector product takes $O(n^2)$ time, and then comparing y and z takes $O(n)$ time – thus the algorithm runs in $O(n^2)$ time. Moreover, it is clear that if $AB = C$, then the algorithm will definitely output TRUE. The main thing now is to understand when the fails, i.e., when it returns $TRUE$ even though $AB \neq C$.

Theorem 1.1.

$$\mathbb{P}[\textit{FREIVALDS}(A, B, C) \textit{ returns } \textit{TRUE}\,|\, AB \neq C] \leq \frac{1}{2}$$

Algorithm 1 FREIVALDS(A, B, C)

1: Pick random n-bit vector. Formally – pick $\sigma \in \{0,1\}^n$ u.a.r (*uniformly at random*)
2: Compute $z = C\sigma$, and $y = A(B\sigma)$ (Note: this involves 3 matrix-vector products)
3: **if** $y = z$ **then**
4: **return** TRUE
5: **else**
6: **return** FALSE
7: **end if**

Proof: Let $D = AB - C$ – note that for a given σ, FREIVALDS(A, B, C) returns TRUE iff $D\sigma = 0$. Thus, to prove the theorem, it is sufficient to show that $\mathbb{P}[D\sigma = 0 \pmod 2 | D \neq 0] \leq 1/2$. We drop the (mod 2) notation, remembering that we are working in base 2.

Suppose indeed that $D \neq 0$ but for the chosen σ, we have $D\sigma = 0$. The main idea behind bounding the probability of this event is the use of the *principle of deferred decisions*. First, since $D \neq 0$, we know that there exists at least one index (i, j).s.t. $D_{ij} \neq 0$. Moreover, we also have that $\sum_{k=1}^{n} D_{ik}\sigma_k = 0$, which we can re-write as:

$$\sigma_j = \frac{-\sum_{k=1}^{j-1} D_{ik}\sigma_k + \sum_{k=j+1}^{n} D_{ik}\sigma_k}{d_{ij}} \tag{1.1}$$

Remember that we Now the idea behind deferred decisions is that *once we fix the realizations* of $(\sigma_1, \sigma_2, \ldots, \sigma_{j-1}, \sigma_{j+1}, \ldots, \sigma_n)$, then there is only a single possible value that σ_j must take, as given by Equation 1.1. However, since each $\sigma_i = Bernoulli(1/2)$ i.i.d. [2], this happens with probability $1/2$.

Finally, since D could have multiple non-zero entries, the above condition is only necessary (and not sufficient) to ensure that $D\sigma = 0$ – thus $\mathbb{P}[D\sigma = 0 \pmod 2 | D \neq 0] \leq 1/2$ $\square$

A few final notes about this algorithm:

- It matches our lower bound! (at least, in an *order-wise* sense, i.e., ignoring constants).
- Algorithms like Freivalds, where the runtime is bounded in the worst-case, but the output is 'correct' only with some probability, are called *Monte Carlo* algorithms. Moreover, Freivalds is said to have a *one-sided error* – it is always correct if $AB = C$, but errs only if $AB \neq C$.
- A Las Vegas algorithm can be repeated multiple times independently in order to improve the probability. In particular, if an algorithm is successful with probability at least p, then $\frac{c \ln n}{p}$ independent runs are sufficient to guarantee success with probability at least $1/n^c$ (see homework 2).

The Wikipedia entry on Freivalds' algorithm is a good place to learn more on this.

[2] Convince yourself that picking each $\sigma_i = Bernoulli(1/2)$ i.i.d. is the same as picking $\sigma \in \{0,1\}^n$ u.a.r.

1.4 Sorting

In our next example, we will see a different style of randomized algorithm, where we always get the correct answer, but the runtime may vary depending on the randomness.

The problem we consider is that of sorting an array. We are given an array $S = \{x_1, x_2, \ldots, x_n\}$ of length n, of all distinct elements. Our aim is to output the array sorted in ascending order – i.e., we want $\{S_1, S_2, \ldots, S_n\}$, where S_i is the i^{th}-largest element in S.

Before we present an algorithm, we first try to understand what sort of runtime guarantee we want. First, note that the array $\{S_1, S_2, \ldots, S_n\}$ can be permuted in $n!$ ways – thus, sorting the array S is essentially the same as identifying which permutation it is. Moreover, if we can only compare between elements, the best we can hope to do is use each comparison to reduce the set of feasible permutations. Thus the number of comparisons we need to identify the correct permutation is $\Omega(\log n!) = \Omega(n \log n)$ (see homework 1). We now see one way to achieve this bound.

1.4.1 Randomized Quicksort

The algorithm we will study is called Quicksort, first proposed by C.A.R. Hoare in 1959.

Algorithm 2 QUICKSORT(S)

1: Pick element $p \in S$ u.a.r. as the *pivot*
2: Compute $S_\ell = \{x \in S | x < p\}$, and $S_h = \{x \in S | x > p\}$
3: **return** $\{\text{QUICKSORT}(S_\ell), p, \text{QUICKSORT}(S_h)\}$

Few notes:

- This is a *recursive algorithm* – we split the array and use QUICKSORT on each sub-array.
- To split an array of size n above a pivot, we need n comparisons.
- To get some intuition as to why it gives the desired running time, suppose that we always could pick the median of S as the pivot. Then clearly $|S_\ell| \approx |S_h| \approx |S|/2$, and thus we have that the runtime of QUICKSORT T_n for an n-length array satisfies: $T_n = 2T_{n/2} + O(n)$ In homework 1, we show that the solution to this is $O(n \log n)$ – essentially, the total number of iterations is bounded by $\log_2 n$, and in each iteration, the number of comparisons is the same as the sum of sizes of each sub-array. However, since the sub-arrays at any stage partition S, we have that the number of comparisons at any stage of the iteration tree is $O(n)$.
- Choosing the pivot randomly is important. Suppose instead we chose the first element of each array as the pivot. Now if the array S was actually sorted in descending order, then at each stage, we will do n comparisons to get $S_\ell = \phi$ and $S_h = S \setminus \{x_1\}$. Thus the runtime is $n + (n-1) + \ldots + 1 = O(n^2)$.

As we mentioned before, the output of QUICKSORT is always the sorted array. Our goal is to show that in addition, the runtime is as promised.

Theorem 1.2. *If the pivot is chosen independently and u.a.r. for each sub-array, then:*

$$\mathbb{E}[T_n] = O(n \log n)$$

Proof: To calculate the expected runtime, we will use a slick argument based on indicator r.v.s and linearity of expectation. For indices $1 \leq i \leq j \leq n$, we define:

$$X_{ij} = \begin{cases} 1 & \text{if } S_i \text{ and } S_j \text{ are ever compared during the algorithm} \\ 0 & \text{otherwise} \end{cases}$$

One way to measure the running time is to keep track of every comparison that is made – thus, we have that $T_n = \sum_{1 \leq i \leq j \leq n} X_{ij}$, and hence $\mathbb{E}[T_n] = \sum_{1 \leq i \leq j \leq n} \mathbb{E}[X_{ij}]$ by linearity of expectation. We now need the following intermediate result:

Proposition 3. *$X_{ij} = 1$ IFF the first pivot chosen from among the elements $\{S_i, S_{i+1}, \ldots, S_j\}$ is S_i or S_j.*

Proof (of proposition): We prove this by the principle of deferred decisions. Initially, both S_i and S_j are in the array – we now have the following cases:

- Whenever we pick a pivot which is $< S_i$ or greater than S_j, both S_i and S_j end up in the same sub-array.
- If we pick S_i or S_j as the first pivot in $\{S_i, S_{i+1}, \ldots, S_j\}$, then the two are compared.
- If however we pick one of $\{S_{i+1}, \ldots, S_{j-1}\}$, then the two are separated, and hence never compared in future.

□

Using the above proposition, we see that $\mathbb{E}[X_{ij}] = \mathbb{P}[X_{ij} = 1] = \frac{2}{j-i+1}$. Now we have:

$$\begin{aligned}
\mathbb{E}[T_n] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2 \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{1}{k} = 2 \sum_{k=2}^{n} \sum_{i=1}^{n-k+1} \frac{1}{k} \\
&= 2 \sum_{k=2}^{n} \frac{n+1}{k} - 2 \sum_{k=2}^{n} 1 \\
&= 2\left((n+1)H_n - n - 1 - (n-1)\right) = 2\left((n+1)H_n - 2n\right) \\
&= O(n \log n)
\end{aligned}$$

□

Although this argument is really slick, one problem with knowing the expected running time alone is that it does not guarantee any bounds on the true running time. Later, when we learn the Markov and Chebyshev bounds, we will see that the above technique plus these bounds allow us to show that $\mathbb{P}[T_n > (1+\epsilon)\mathbb{E}[T_n]]$ is indeed small (in fact, with some effort, we can use Chebyshev to show it is $o(n)$) – thus QUICKSORT does truly achieve $O(n \log n)$ running time. However, after we learn the Chernoff bound, we will see an alternate way of analyzing QUICKSORT which gives tighter bounds, and (I think!) more intuition.

1.5 Randomized Graph Algorithms

Finally, before we shift to talking about tail inequalities, we'll look at two more examples where randomization is used in designing graph algorithms. In particular, we will focus on the Min-Cut and Max-Cut problems. First though, some notation:

- We will denote a graph as $G(V, E)$, where $V = \{1, 2, \ldots, n\}$ is the set of vertices, and E is the set of edges connecting vertices. We usually use $n = |V|$ and $m = |E|$ as the number of vertices and edges respectively.
- Another way to represent a graph is in terms of its *adjacency matrix* A, which is an $n \times n$ $\{0,1\}$-matrix, with $A_{ij} = 1$ iff $(i,j) \in E$ (i.e., i and j are connected by an edge).
- Unless stated otherwise, we will usually consider *undirected, unweighted, simple* graphs: undirected means that edges have no direction, and hence $A_{ij} = A_{ji}$ (or $A = A^T$), unweighted means that A has only $\{0,1\}$ values, and simple means that there are no self loops or multiple edges (i.e., the diagonals of A are 0 and all other values are in $\{0,1\}$. However, we will occasionally deal with:
  - Directed graphs – where edges have directions; thus (i,j) and (j,i) are different edges, or equivalently, A_{ij} and A_{ji} may have different values.
  - Multigraphs – where a pair of nodes may have multiple edges between them; thus A_{ij} can now take values in $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$.
  - Weighted graphs – where edges have associated weights. In such cases, we usually use $A_{ij} = \mathbb{1}_{\{w_{ij} \neq 0\}}$ (i.e., is the weight on edge (i,j) non-0), and use a separate *weight matrix* W to encode the weights.
- For a given node v in an undirected graph, we will use $d(v) = \sum_{u \in V} \mathbb{1}_{\{(u,v) \in E\}}$ to denote the *degree* of the node, i.e., the number of edges incident on v (or edges for which v is an end-point). For a weighted undirected graph, we will use w_v to denote the total weight of edges incident on v. For a directed graph, we will need to distinguish between *in-degree* and *out-degree*.
- A *cut* $(C, \overline{C})$ is a partition of the vertices of the graph – i.e., $C \cap \overline{C} = \phi$ and $C \cup \overline{C} = V$.
- We will define $E(C, \overline{C}) = \{(i,j) \in E | i \in C, j \in \overline{C}\}$ to denote the edges *in the cut* $(C, \overline{C})$, and $\delta(C, \overline{C}) = |E(C, \overline{C})|$ as the *cut-weight* of the cut $(C, \overline{C})$.

Given this notation, the two problems we want to look at are the following:

- **Max-Cut:** Find cut $(C, \overline{C})$ so as to *maximize* $\delta(C, \overline{C})$.
- **Min-Cut:** Find $(C, \overline{C})$ so as to *minimize* $\delta(C, \overline{C})$.

The two problems, on first glance, seem very similar. Moreover, the brute-force way of solving either is the same – consider all possible non-trivial (i.e., non empty) subsets $C \subset V$, and find the one with the maximum/minimum cut. Note that there are $2^{n-1}-1$ such subsets (there are 2^n-2 ways to pick a non-trivial subset of V, and each partition has two subsets) – thus this will take $\Omega(2^n)$ time. Using more clever algorithms, we know how to solve Min-Cut much faster than that – in particular, there are deterministic algorithms that solve it in $O(n^3)$ time. Surprisingly, however, Max-Cut is an NP-complete problem – we do not know how to do much better than brute force. The best we can do if we insist on a polynomial-time algorithm is to guarantee that although we may not find the true Max-Cut, we can still find a cut whose number of edges is within a constant factor of the true Max-Cut – such a guarantee is called an *approximation ratio*.

For us, what is more interesting is that the best algorithm for both problems (fastest for Min-Cut, best approximation guarantee for Max-Cut) use randomization in very clever ways!

1.5.1 (Simple) Randomized Max-Cut

One intuition behind using randomization to find the Max-Cut of a graph is that if we pick random edges in the graph, then these edges are likely to be in the maximum cut. Creating a cut by picking random edges, though, is tricky, as once we pick some edges, then the other edges are constrained (as their end points may already have been assigned to C or $\overline{C}$. However, it is easy to create a cut by randomly assigning nodes to C to $\overline{C}$, as follows:

Algorithm 3 RANDMAXCUT(G)

1: **for** Each node $v \in V$ **do**
2: Assign v to C with probability 1/2, else assign to $\overline{C}$.
3: **end for**

Clearly the above algorithm runs in time $O(n)$ – surprisingly, it also gives a decent approximation guarantee:

Theorem 1.3. *Let $(C, \overline{C})$ be the cut returned by RANDMAXCUT(G). Then:*

$$\mathbb{E}[\delta(C, \overline{C})] \geq \frac{1}{2} \textit{Max-Cut}(G)$$

Proof: We will actually prove the following stronger statement: $\mathbb{E}[\delta(C, \overline{C})] = \frac{m}{2}$. Since the Max-Cut can have at most m edges, this will give us what we need.

For every edge $(i, j) \in E$, let X_{ij} denote the indicator r.v. that is 1 if $(i, j) \in E(C, \overline{C})$. Then, we have $\delta(C, \overline{C}) = \sum_{(i,j)\in E} X_{ij}$, and thus by linearity of expectation, $\mathbb{E}[\delta(C, \overline{C})] = \sum_{(i,j)\in E} \mathbb{E}[X_{ij}]$. Now to compute $\mathbb{E}[X_{ij}] = \mathbb{P}[X_{ij} = 1]$, we use the principle of deferred decisions. For any edge (i, j), suppose we first assign node i to either C or $\overline{C}$ – then in order for (i, j) to be in the cut, we must assign node j to the other set, which happens with probability 1/2. Thus, we get that $\mathbb{E}[\delta(C, \overline{C})] = \sum_{(i,j)\in E} 1/2 = m/2$. □

Before moving on to Min-Cut, a bit of history: the Max-Cut problem was shown to be NP-Complete in the 1970s, and the simple algorithm given above (as well as simple greedy procedures with the same approximation guarantee) were known since about the same time. It took more than 20 years before the above bound was improved in the mid-90s by Michel Goemans and David Williamson, who gave an algorithm with an approximation ratio of 0.878 – recent work has suggested that this can not be improved any further. The Goemans-Williamson algorithm is based on a technique called *semidefinite programming relaxation* followed by a beautiful *randomized rounding* procedure. Presenting it is just outside the scope of this class, but if you are interested (especially in the rounding ideas), there are excellent lecture notes available online on this topic.

1.5.2 Randomized Min-Cut: A First Attempt

Note: My presentation here is mostly based on Section 10.2 in Motwani and Raghavan (MR). Also, the Wikipedia entry for the Karger-Stein algorithms is really well written!

We now turn to the Min-Cut problem. In contrast to Max-Cut, this is actually solvable in polynomial time – the fastest-known deterministic algorithm takes $O(n^3)$ time [3]. However, a remarkable algorithm by David Karger and Clifford Stein in 1996 uses a clever randomization technique to find the Min-Cut of a graph G in $O(n^2 \log^{O(1)} n)$ (i.e., $O(n \log^{\alpha} n)$ for some constant α – this is often denoted as $O(n\text{polylog}(n))$ or $\widetilde{O}(n)$).

Before presenting the Karger-Stein algorithm, we first need to understand a simpler randomized Min-Cut finding procedure, which is a subroutine for the Karger-Stein algorithm. The main idea behind a randomized algorithm for Max-Cut is that unlike in Max-Cut, a randomly picked edge is likely to *not belong to the minimum cut* – thus, the end-nodes of a randomly picked edge are likely in the same side of the cut. One way to use this is to 'contract' a randomly picked edge (i, j) into a super node $\{i, j\}$. In the process, we may create multiple edges between nodes (for example, if both i and j are connected to a node k, then contracting (i, j) will create a parallel edge between the super node $\{i, j\}$ and k). Thus, it makes sense to define this algorithm for a multigraph G. Moreover, note that any cut in the graph formed after any sequence of edge contractions is a cut in the original graph. If we contract edges until we are left with two super nodes, then we have a cut – however, we can also stop when we have $t \geq 2$ nodes left, in which case the resulting graph has $2^{t-1} - 1$ possible cuts. We will define the algorithm for the case of multigraphs, and for stopping with t nodes left, as this will be useful later when we use it as a primitive for Karger-Stein.

Algorithm 4 CONTRACT(G, t)

1: **while** Number of nodes in $G \geq t$ **do**
2: Pick uniform random edge, and contract its endpoints.
3: **end while**

[3] More specifically, $O(nm + n^2 \log n)$ time – see this Wikipedia article for details.

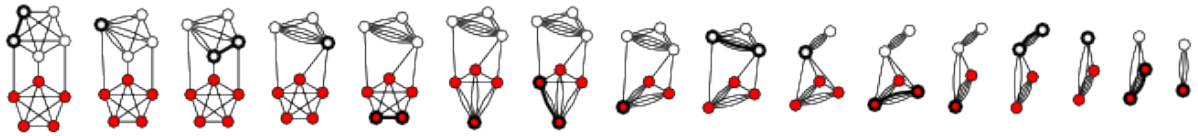

Figure 1.1. A successful run of the CONTRACT algorithm. Image credit: *"Single run of Karger's Mincut algorithm" by Thore Husfeldt - Created in python using the networkx library for graph manipulation, neato for layout, and TikZ for drawing. Licensed under CC BY-SA 3.0 via Commons - Wikipedia link*

Figure 1.1 shows a successful execution of CONTRACT$(G, 2)$, wherein after contracting down to 2 nodes, the algorithm actually retrieves the Min-Cut of the graph. For any given cut $(C, \overline{C})$, it is easy to see that the cut is preserved after executing CONTRACT(G, t) if the $n - t$ edges picked for contracting *do not include any edge in* $E(C, \overline{(C)})$. Now to understand the performance of CONTRACT(G, t), we need to bound its running time, and the probability that it preserves a minimum cut. We will not discuss the running time here, except to note that as long as $n - t = \Theta(n)$, then the running time is $O(n^2)$. For the probability that a chosen Min-Cut of the graph is preserved by CONTRACT(G, t), we have the following:

Theorem 1.4. *Given any minimum cut* $(C, \overline{C})$ *of graph* G*, we have:*

$$\mathbb{P}[\textit{CONTRACT}(G, t) \textit{ preserves } (C, \overline{C})] \geq = \frac{t(t-1)}{n(n-1)}.$$

Proof: Suppose we are given any multigraph G with n nodes and with a minimum cut $(C, \overline{C})$ with k edges. Then for any vertex of the G, we have that $d(v) \geq k$ (else it is smaller than the minimum cut, which is a contradiction. This implies that the total number of edges satisfies $m \leq nk/2$. Using this, we have that:

$$\mathbb{P}[\text{Randomly picked edge lies in } E(C, \overline{C})] \leq \frac{k}{nk/2} = \frac{2}{n}.$$

Note that this is true for any multigraph of size n.

Returning to the CONTRACT algorithm, in order for the minimum cut $(C, \overline{C})$ to be preserved, we require that no edge in $E(C, \overline{C})$ is picked in the $n - t$ random edge selections. Thus, we have:

$$\begin{aligned}
\mathbb{P}[\text{CONTRACT}(G, t) \text{ preserves } (C, \overline{C})] &\geq \Pi_{i=0}^{n-t-1} \left(1 - \frac{2}{n-i}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{t-1}{t+1}\right) \\
&= \frac{t(t-1)}{n(n-1)}
\end{aligned}$$

□

We also get the following corollary setting $t = 2$:

Corollary 1.5. *A single CONTRACT$(G, 2)$ returns a minimum cut of G in $O(n^2)$ time with probability at least $\frac{2}{n(n-1)} = O\left(\frac{1}{n^2}\right)$.*

In order to find the true minimum-cut with probability at least $1/n$, we can do $O(n^2 \log n)$ independent runs of $CONTRACT(G, 2)$ – this overall needs $O(n^4 \log n)$ runtime.

1.5.3 Randomized Min-Cut: The Karger-Stein Algorithm

The above corollary shows that using CONTRACT$G, 2$ repeatedly is not sufficient to get a better running time than the best deterministic algorithm. To understand what went wrong, note that the initial edge contractions preserved the minimum cut with fairly high probability (for example, the i^{th} contract preserved it with probability $1 - 2/(n - i + 1) \sim 1$ as long as $i = o(n)$). On the other hand, the last few edge contractions actually had a fairly low probability of not picking an edge in $E(C, \overline{C})$ (for the last step, its as low as $1/3$).

One option is to use CONTRACTG, t for a much larger t (for example, $t = n^\alpha$ for some $\alpha \in (0, 1]$, and then switch to using a deterministic algorithm on the t-node multigraph. Doing this carefully does in fact improve the running time – in the homework, you will show that this can achieve an $O(n^{8/3})$ running time for finding the minimum cut with probability at least $1/2$.

Next, note that since CONTRACTG, t preserves the minimum cut with probability $\Omega(t^2/n^2)$, in order to preserve it with a constant probability, we need to stop at $t = \alpha n$ for some constant n. In particular, if we choose $t = \lceil 1 + n/\sqrt{2} \rceil$, then CONTRACT$G, t$ preserves the minimum cut with probability at least $1/2$. The main idea behind the Karger-Stein algorithm is to use two independent runs of CONTRACTG, t with the above value of t, and then recurse on each subproblem. The reason this works is that doing multiple runs of CONTRACT(G, t) boosts the probability of preserving the minimum cut, while the cost of repeating on a smaller graph is much less than that of repeating on the original large graph. Following the notation in MR, we refer to this as the FASTCUT algorithm:

Algorithm 5 FASTCUT(G)

1: If $n = |G| \leq 6$, then find the min-cut deterministically.
2: Set $t = \lceil 1 + n/\sqrt{2} \rceil$.
3: Generate independent subgraphs G_1, G_2, using CONTRACT(G, t).
4: Execute FASTCUT(G_1) and FASTCUT(G_2).
5: Return the smaller of the two cuts.

We check $n \geq 6$ to ensure $t \geq 2$. Now we have the following result:

Theorem 1.6. *FASTCUT(G) runs in $O(n^2 \log n)$ time, and outputs a minimum cut with probability $\Omega(1/\log n)$.*

Proof: For the running time, note that we have:

$$T(n) \leq 2T(\alpha n) + O(n^2)$$

where $\alpha = \Theta(1)$ is a constant which is $\geq 1/\sqrt{2}$ (it is roughly $1/\sqrt{2}$). Thus, after k iterations, we get 2^k subproblems, each of size $\alpha^k n$ – the total overhead for the subproblems at this stage is $O(2^k \alpha^{2k} n^2) = O(n^2)$. Moreover, after $O(\log n)$ iterations, the subgraphs are of size ≤ 6 – thus the running time is $O(n^2 \log n)$.

Next, note that for our choice of t, the minimum cut $(C, \overline{C})$ of G survives each contraction with probability at least $1/2$. $\square$