CS 170: Efficient Algorithms and Intractable Problems Spring 2020

Lecture Notes — April 16, 2020

Profs. Alessandro Chiesa and Jelani Nelson

Randomized Algorithms

So far in this class we have talked about *deterministic algorithms*, which do not use randomness as a resource to guide their decision-making. In this lecture we discuss *randomized algorithms*. A randomized algorithm $\mathcal{A}$ is one which generates random bits to guide its execution. Alternatively, $\mathcal{A}$ can be seen as a regular (deterministic) algorithm but that takes two inputs: the usual input x, as well as a uniformly random string $r \in \{0,1\}^R$ for some R. Since r is random, the behavior of $\mathcal{A}$ may differ on different runs on the same input x due to different random strings r being randomly generated. We generally categorize randomized algorithms into two classes:

Las Vegas: These are algorithms for which correctness is guaranteed, but the runtime $t(x,r)$ on input x with randomness r is a random variable. We are then usually concerned with understanding the runtime distribution of $\mathcal{A}$ on an input x for random r. When we talk about $T(n)$ being the expected runtime of $\mathcal{A}$ on inputs x of size n, we still mean worst-case analysis in the following form:

$$T(n) = \max_{|x|=n} \mathop{\mathbb{E}}_{r} t(x,r).$$

Note the *input* is not random (that case is usually referred to as "average-case analysis") but rather is worst case, since we take the max runtime over all inputs x of a fixed length n. The only source of randomness is the random string r used by $\mathcal{A}$. An example of a Las Vegas algorithm, which we cover in this lecture in more detail and have discussed earlier in the semester, is QuickSort. QuickSort always sorts the input correctly, but its runtime is a random variable. That random variable can be as bad as $\approx n^2$ in the worst case, but its expectation for *any* input array is $O(n \log n)$.

Monte Carlo: A Monte Carlo algorithm is one where its runtime is always bounded by some desired bound (with probability 1), but its *correctness* is a random variable. That is, if we let $\Delta(x,r)$ be the so-called "indicator random variable" for the event that $\mathcal{A}(x,r)$ returns an incorrect value ($\Delta(x,r) = 1$ when $\mathcal{A}(x,r)$ is incorrect and $\Delta(x,r) = 0$ otherwise), then $\mathbb{E}\,\Delta(x,r) = \mathbb{P}(\Delta(x,r) = 1) = \delta > 0$ for some positive δ. A good example of a Monte Carlo procedure is polling. Suppose there is an election with two candidates A and B. Now let us suppose x is a list of the n people eligible to vote, together with the name of the candidate they plan to vote for. We would like a procedure which, given x, outputs the percentage of voters voting for A, and we say that the procedure is correct if it outputs a number that is off from the true percentage by say at most five additive percentage points. The straightforward solution is to iterate over every person in x, see who they voted for, and tally votes. This is time-consuming, since a real implementation of this idea would involve talking to every person in the population (if the vote were for California governor, we would have to call tens of millions of Californians!). Instead, the usual way this is done is via *polling*: we use a source of randomness r to sample a random subset of the population

(say, a thousand people), then we call only those sampled voters and ask them who they plan to vote for. Note the "runtime" here is always bounded, by 1000 (we call 1000 people). However whether or not we manage to give a good estimate of the percentage of voters voting for A is a random event, which depends upon our random sample from the population (which itself depended on r). We may be unlucky and randomly select 1000 people who happen to all be voting for A thereby skewing our estimate, and thus $\Delta(x, r) > 0$.

1 QuickSort

We briefly discussed QuickSort [2] earlier in this semester when covering linear time median-finding/selection. Recall the problem description: we must sort an input array $A[1 \dots n]$. Without loss of generality we can assume A has distinct entries, i.e. $A[i] \neq A[j]$ for all $i \neq j$, since we can replace the ith entry of $A[i]$ with the pair $(A[i], i)$ for each i then sort lexicographically. The pseudocode for QuickSort is then below.

Algorithm QuickSort($A[1 \dots n]$):

if $i = 1$, **then** return A
else:

1. pick a uniformly random pivot $\in \{1, \dots, n\}$
2. $L \leftarrow \{i : A[i] < A[\text{pivot}]\}$
3. $R \leftarrow \{i : A[i] > A[\text{pivot}]\}$
4. **return** [QuickSort(L), A[pivot], QuickSort(R)]

Note that steps 2. and 3. to obtain L, R can be done in $O(n)$ time by a for loop through A.

For any fixed input A, the runtime of QuickSort is a random variable which could be as bad as $\approx n^2$ if the pivot is, for example, always randomly chosen to correspond to the smallest element in A. In the other extreme, if the pivot is randomly always chosen as the median element, we would have the recurrence $T(n) = 2T(n/2) + O(n)$, which we know solves to $T(n) = O(n \log n)$. If $t(A, r)$ is the runtime of QuickSort(A) with randomness r, what is then the expected runtime $\mathbb{E}_r\, t(A, r)$?

The key insight is that the runtime $t(A, r)$ is, up to a constant factor, equal to the number of comparisons performed by QuickSort. This is because for every recursive call on some array of size, say, ℓ, the total time spent at that level of recursion (steps 1 through 3 above) is $\Theta(\ell)$, whereas the number of comparisons is also $\ell - 1 = \Theta(\ell)$. Furthermore, for any pair of indices $i \neq j \in \{1, \dots, n\}$, $A[i]$ is compared to $A[j]$ either 0 or 1 times throughout the entire run of the algorithm; this is because for two items to be compared, one of them had to have been the pivot, in which case the non-pivot item is then put into either L or R and never has a chance to be compared with the pivot again. Thus if we define an *indicator random variable* $X_{i,j}$ which is 1 if the ith smallest element is ever compared with the jth smallest and 0 otherwise, we have that

$$t(A, r) \leq C \sum_{i<j} X_{i,j}, \text{ for some constant } C \tag{1}$$

Now we remember *linearity of expectation*:

Claim 1. *If X, Y are random variables and $a, b \in \mathbb{R}$, then*

$$\mathbb{E}[aX + bY] = a\,\mathbb{E}\,X + b\,\mathbb{E}\,Y.$$

Applying linearity of expectation to (1), we have

$$\mathop{\mathbb{E}}_{r} t(A,r) \leq \mathop{\mathbb{E}}_{r}\left[C\sum_{i<j} X_{i,j}\right] = C\sum_{i<j}\mathop{\mathbb{E}}_{r} X_{i,j} = C\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathop{\mathbb{E}}_{r} X_{i,j}, \tag{2}$$

noting that $X_{i,j}$ depends on the randomness in r (r is the source of randomness used to pick the pivots). Let us now turn our attention to bounding $\mathbb{E}\, X_{i,j}$. What is the probability that the ith and jth smallest elements (call them a_i, a_j) are ever compared? If we trace the QuickSort recursion tree from the root, both elements start off being in the same recursive subproblem (the root). Then when the root recursion node picks a random pivot, if that pivot is either $< a_i$ or $> a_j$ then a_i and a_j either both go to L or both go to R (i.e. stay in the same recursive subproblem). Otherwise, if the pivot is in $S = \{a_i, a_{i+1}, \ldots, a_j\}$, then they do not go to the same recursive subproblem. Thus what decides whether $X_{i,j}$ equals 0 or 1 is the following: it is equal to 1 iff the first time a pivot is picked in the set S, it is picked to be either a_i or a_j; otherwise $X_{i,j} = 0$ since a_i, a_j will be sent separately to L and R and never have a chance to be compared again. Thus $\mathbb{P}(X_{i,j} = 1) = 2/(j-i+1)$. Plugging this into (2),

$$\mathop{\mathbb{E}}_{r} t(A,r) \leq C\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1}. \tag{3}$$

Looking carefully at the summation on the right-hand side of (3), we see that the number of times $2/n$ appears is once (when $i=1$), $2/(n-1)$ appears twice (when $i=1,2$), $2/(n-2)$ appears three times (when $i=1,2,3$), etc. In general, $2/(n-k+1)$ appears k times for $k = 1, 2, \ldots, n-1$. Thus

$$\begin{aligned}
\mathop{\mathbb{E}}_{r} t(A,r) &\leq C\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\
&\leq 2C\sum_{k=1}^{n-1}\frac{k}{n-k+1} \\
&\leq 2Cn\sum_{k=1}^{n-1}\frac{1}{n-k+1} \text{ (since } k \leq n) \\
&= 2Cn\sum_{t=2}^{n}\frac{1}{t} \\
&\leq 2Cn\int_{1}^{n}\frac{1}{t}\mathrm{d}t \\
&= 2Cn\ln n.
\end{aligned}$$

We thus have that the expected runtime is $O(n\log n)$, as desired.

One may not wish to only bound the *expected* runtime of an algorithm, but to also say its runtime is low with high probability. A statement of this form is true for QuickSort, though we will not prove the strongest possible such statement here. We will instead recall Markov's inequality.

Claim 2 (Markov's inequality)**.** *Let Z be a nonnegative random variable. Then*

$$\forall \lambda > 0,\ \mathbb{P}(Z > \lambda) < \frac{\mathbb{E}\, Z}{\lambda}.$$

We have shown that for Z denoting the runtime of QuickSort (which is a random variable), $\mathbb{E}\, Z < Cn \log n$ for some constant $C > 0$. Since Z is clearly nonnegative (algorithms do not run in negative time!), we can apply Markov's inequality to say that $\mathbb{P}(Z > 200Cn \log n) < 1/100$. Thus its running time is not only $O(n \log n)$ in expectation, but also with 99% probability.

2 Freivalds' Algorithm

In this and the next subsection, we consider Monte Carlo algorithms. The problem we consider now is that of verifying computation. Imagine we have some problem we would like to solve on a big input, but we have limited computational resources locally. Thus we upload the input to the cloud (say AWS) to have some big company use its heavy duty compute power to solve our problem for us. The cloud then returns the answer to us. This though presents at least one problem: how can we ensure that the answer returned to us is correct? Even if the cloud company is not malicious, there may be bugs somewhere in its system causing us to get wrong outputs. Thus what we would like is a way to *verify* results much more efficiently than redoing the entire computation ourselves.

Freivalds' algorithm [1] specifically focuses on verifying matrix-matrix multiplication. That is, we have two matrices $A, B \in \mathbb{R}^{n \times n}$ and would like to know $C = A \times B$. We could of course multiply these matrices ourselves in $\Theta(n^3)$ time (or faster using Strassen or more recent algorithms), but if we upload A, B to the cloud which returns some C to us, is there a way to verify that $C = A \times B$ much faster than doing the multiplication ourselves from scratch?

It turns out that the answer is *yes*, using randomization. Freivalds' algorithm does the following simple thing: pick a uniformly random $x \in \{0,1\}^n$ and check whether $ABx = Cx$. Note this check can be implemented in $O(n^2)$ time instead of $O(n^3)$ time via associativity, since $ABx = A(Bx)$; that is, we can do two matrix-vector multiplications Bx, then $A(Bx)$, instead of ever having to multiply $A \times B$.

Claim 3. *If $C = A \times B$, then $\mathbb{P}_x(ABx = Cx) = 1$. Else, $\mathbb{P}_x(ABx = Cx) \le 1/2$.*

Proof. The first half of the claim is obvious. For the second half, for the sake of the analysis define $D := AB - C$. Then $D \neq 0$. Since it is not zero, in particular its ith column D_i is nonzero for some i. Now suppose $ABz = Cz$ for some particular $z \in \{0,1\}^n$, i.e. $Dz = 0$. Then if $z' = z$ but with the ith entry flipped, we have $Dz' \neq 0$ since $Dz' = D(z \pm e_i) = Dz \pm De_i = \pm D_i \neq 0$, where e_i is the ith standard basis vector (0 everywhere with a 1 in the ith entry). Now let us pair all the 2^n elements of $\{0,1\}^n$ into 2^{n-1} pairs, where each pair differs in only the ith entry; e.g. if $n = 3, i = 2$, the $2^{3-1} = 4$ pairs would be: $(0,0,0)$ paired with $(0,1,0)$, $(1,0,0)$ paired with $(1,1,0)$, $(0,0,1)$ paired with $(0,1,1)$ and $(1,0,1)$ paired with $(1,1,1)$. Then we see that in each pair (z, z'), at most one of Dz or Dz' can equal zero, but not both. Thus $|\{z \in \{0,1\}^n : Dz = 0| \le 2^{n-1}$. Thus $\mathbb{P}_x(ABx = Cx) = \mathbb{P}_x(Dx = 0) \le 2^{n-1}/2^n = 1/2$. □

If we want to reduce our probability of returning an incorrect answer to some desired failure probability p, we can for example set $N = \lceil \log_2 1/p \rceil$ and pick N random vectors $x_1, \ldots, x_n \in \{0,1\}^n$ independently. We then declare $A \times B = C$ iff $ABx_i = Cx_i$ for *all* $i = 1, \ldots, N$. Then the probability we fail is at most $1/2^N \le p$. Our runtime would then be $\Theta(n^2 \log(1/p))$.

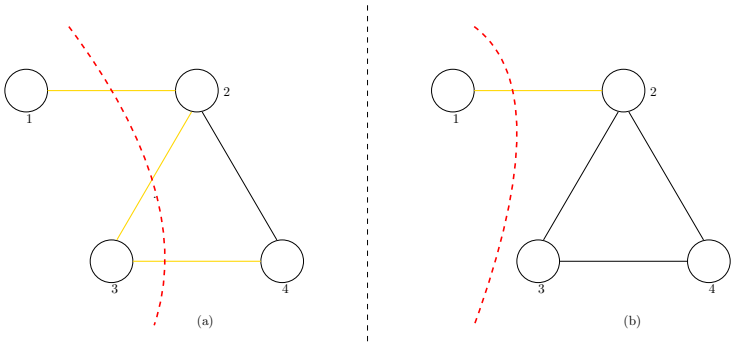

Figure 1: Consider the graph G above on four vertices. Panel (a) denotes the cut $S = \{1, 3\}$, $\bar{S} = \{2, 4\}$, which we see has size 3 (we have highlighted the three edges crossing the cut in yellow). The dashed red line visually separates S from $\bar{S}$, and the cut edges are those crossing it. Meanwhile (b) denotes the cut $S = \{1\}$, $\bar{S} = \{2, 3, 4\}$, which has size 1 and is in fact the minimum cut of G.

3 Karger's Global Minimum Cut Algorithm

The last algorithm we cover is one due to Karger for the global minimum cut problem [3]. As Karger has several algorithms for this same problem, this particular one is often called the Contraction algorithm for reasons that will be clear shortly.

First, the problem: we are given a (possibly weighted) undirected graph G and would like to find a *cut* of smallest size. A cut is a partition of the vertices V into two non-empty sets $S, \bar{S}$. The *size* of the cut is then defined to be $\sum_{(u,v)\in E\cap(S\times\bar{S})} w(u, v)$. If the weights are all one, then the size is just $|E \cap (S \times \bar{S})|$, i.e. the number of edges that cross from one side of the cut to the other. See Figure 1 for an example. In this lecture we focus exclusively on the unweighted case (though it is possible to adapt the Contraction algorithm to the weighted case as well). We will assume the graph is connected, so in particular $m \geq n - 1$, since otherwise we could find a cut of size 0 using depth- or breadth-first search (let S be one connected component and $\bar{S}$ be everything else).

We first note that *maximum flow* can be used to solve global mincut deterministically. In particular, note that the global minimum cut has to separate vertex 1 from *some* other vertex t, though we do not know a priori which one. We thus set $s = 1$ and loop over all possibilities for $t \in \{2, \ldots, n\}$. For each possibility we compute the s-t minimum cut as discussed in earlier lectures on the maximum flow problem. In an unweighted graph the capacities are all 1, so the value of the max flow/min cut is at most $n - 1$ (the cut $S = \{1\}$ has size equal to the degree of vertex 1, which is at most $n - 1$, so the min cut size can only be $n - 1$ or smaller). Thus each s-t min cut computation takes $O(mn)$ time using Ford-Fulkerson. Since we do n s-t mincut computations, the total time is $O(mn^2)$.

We now describe and analyze Karger's simpler Contraction algorithm, which does not rely on flows.

Algorithm Contraction(G):

for $i = 1, \ldots, |V(G)| - 2$:
1. pick a uniformly random edge e
2. **contract**(e)

return the cut specified by the remaining two supervertices

We explain the pseudocode in the above algorithm then provide an example for extra clarity in Figure 2. In step 2., the **contract** operation on an edge $e = (u, v)$ removes e, u, v from the graph and adds a new merged "supervertex" which we call uv. Also, all edges that used to point into either u or v remain (other than edges between u and v, which are removed), but instead point into uv. If a vertex had edges to both u *and* to v, then both those edges remain and both point to uv, and thus over the course of the execution of the algorithm we may have multi-edges (multiple parallel edges between the same two vertices).

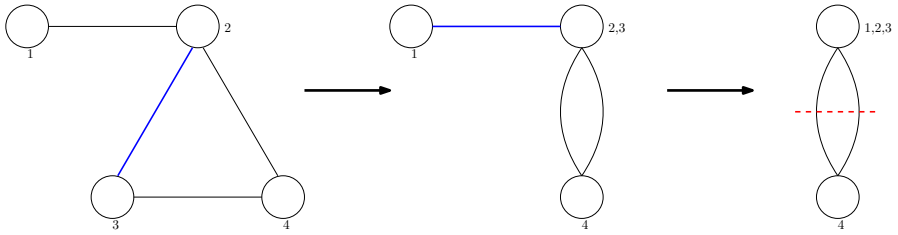

Figure 2: We show one run of the Contraction algorithm on the same 4-vertex graph from Figure 1. As $n = 4$, there are $n - 2 = 2$ iterations of the algorithm. For each of these iterations, we pick a random edge (highlighted in blue) to contract. The next panel then shows the resulting graph after contraction. In this run, we first contracted $(2, 3)$, then next the edge $(1, (2, 3))$ between 1 and the new merged "supervertex" $2, 3$. After $n - 2 = 2$ contractions we are down to two supervertices, which in this example led to the cut $S = \{1, 2, 3\}$ and $\bar{S} = \{4\}$ with size 2.

Lemma 4. *For any graph G, fix a particular global minimum cut $S, \bar{S}$. Then the probability that Contraction(G) return this particular cut is at least $1/\binom{n}{2}$.*

Proof. The contraction algorithm will return $S, \bar{S}$ iff no edge crossing the cut is ever contracted. Let k be the number of edges crossing the cut, and let m_i and n_i denote the number of edges (resp. vertices) in the graph during the ith iteration of the for loop, $i = 1, 2, \ldots, n - 2$ (note n_i is just $n - i + 1$, and m_i is a random variable that depends on what has been contracted so far). Then the probability that we never contract an edge crossing this cut is precisely

$$\prod_{i=1}^{n-2} \left(1 - \frac{k}{m_i}\right). \tag{4}$$

Now we get a handle on the m_i. Note that contracting an edge cannot decrease the minimum cut value. Thus for each intermediate (multi)graph throughout the run of the algorithm, the minimum cut value is always at least k. Thus the vertex degrees are always at least k (else we could take the cut which separates a low-degree vertex from the rest of the graph and obtain a cut of size $< k$). In the ith iteration $\sum_{v=1}^{n_i} d_v = 2m_i$, but we just argued that $\sum_{v=1}^{n_i} d_v \geq \sum_{v=1}^{n_i} k = (n - i + 1)k$. Thus $m_i \geq (n - i + 1)k/2$. Plugging this back into (4), our success probability is at least

$$\prod_{i=1}^{n-2} \left(1 - \frac{2}{n - i + 1}\right) = \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3}.$$

The above is a telescoping product, which equals $1/n \cdot 1/(n-1) \cdot 2 \cdot 1 = 1/\binom{n}{2}$, as desired. □

Note Lemma 4 implies that any graph has at most $\binom{n}{2}$ minimum cuts, since with a graph having Q mincuts it is not possible that each one has probability $> 1/Q$ of being returned (else we would have probabilities of disjoint events sum to more than 1). This is in fact tight: the cycle on n vertices has exactly $\binom{n}{2}$ global mincuts: pick an arbitrary two edges to cut and let $S, \bar{S}$ be the two distinct arcs that are disconnected by removing those two edges. Furthermore, it is possible to implement the Contraction algorithm in $O(m)$ time, but we will not give the details here. It should be clear though that it can be implemented in polynomial time in n.

Now, Lemma 4 does seem a bit problematic: we usually like our success probability to be *large*, but $1/\binom{n}{2}$ is quite a small success probability. This can be fixed by repetition: we will run the algorithm $N = \lceil \binom{n}{2} \cdot \ln 1/p \rceil$ times, independently, and return the smallest cut ever found. To analyze this, we make use of the following useful fact from calculus, using convexity of the function e^x.

Fact 5. *For all $x \in \mathbb{R}$, $1 + x \leq e^x$.*

Proof. Define $f(x) = e^x$. Taylor's theorem gives $e^x = 1+x+f''(y)y^2/2$ for some $y \in [0, x]$. But note that $f''(y) = e^y > 0$ for all $y \in \mathbb{R}$, and thus $f''(y)y^2/2 \geq 0$. Thus $e^x = 1+x+f''(y)y^2/2 \geq 1+x$. □

Now we analyze the idea of repeating the algorithm N times. The probability that we never find a global minimum cut is at most

$$\begin{aligned}\left(1 - 1/\binom{n}{2}\right)^N &\leq (e^{-1/\binom{n}{2}})^N \text{ (Fact 5 with } x = -1/\binom{n}{2}) \\ &= e^{-N/\binom{n}{2}} \\ &\leq p \text{ (by choice of } N).\end{aligned}$$

Thus overall our runtime is $O(mN) = O(mn^2 \log(1/p))$ to succeed with probability $1 - p$. This runtime is comparable to the max flow based approach using Ford-Fulkerson (though slower by a $\log(1/p)$ factor), but is much simpler.

References

[1] Rusins Freivalds. Probabilistic Machines Can Use Less Running Time. *IFIP Congress*, pages 839–842, 1977.

[2] Charles Antony Richard Hoare. Algorithm 64: Quicksort. *Comm. ACM.*, volume 4, number 7, page 321, 1961.

[3] David R. Karger. Global Min-cuts in $\mathcal{RNC}$, and Other Ramifications of a Simple Min-Cut Algorithm. *Proceedings of the 4th Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA)*, pages 21–30, 1993.