CS 388R: Randomized Algorithms Fall 2017

Lecture 1 — August 31, 2017

Prof. Eric Price *Scribe: Garrett Goble, Daniel Brown*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Randomized Algorithms

This course covers using randomness for algorithms.

Cost of Randomness

- chance the answer is wrong (Monte Carlo Algorithm)
- chance the algorithm takes a long time (Las Vegas Algorithm)

Benefits of Randomness

- on average, with high probability gives a faster or simpler algorithm
- some problems require randomness (e.g. Nash Equilibrium)

The main technique is avoiding the worst case or adversarial inputs.

Example Uses

- Fingerprinting: compare two items by hashing down to a small fingerprint using randomness to avoid false positives
- Load balancing by allocating randomly
- Sampling: reducing a large collection of items to a representative subsample
- Symmetry breaking: e.g., network protocols to elect a leader with random tie-breaking
- Probabilistic method: if a random object satisfies a property with non-zero probability, we just proved that such an object exists

2 Types of randomized algorithms

Las Vegas Algorithm (LV):

- Always correct
- Runtime is random (small time with good probability)
- Examples : Quicksort, Hashing

Monte Carlo Algorithm (MC):

- Always bounded in runtime
- Correctness is random
- Examples : Karger's min-cut algorithm

It's easy to see the following:

1. LV $\implies$ MC. Fix a time T and let the algorithm run for T steps. If the algorithm terminates before T, we output the answer, otherwise we output 0.
2. MC does not always imply LV. The implications holds when verifying a solution can be done much faster than finding one. In that case, we test the output of MC algorithm and stop only when a correct solution is found.

3 Notation and General Bounds

$$[n] = \{1, 2, \ldots, n\} \tag{1}$$

3.1 Comparisons

$$f \lesssim g \iff \exists \quad \text{a constant} \quad c > 0 \mid f \leq cg \tag{2}$$
$$f \gtrsim g \iff g \lesssim f \tag{3}$$
$$f \eqsim g \iff f \lesssim g \quad \text{and} \quad g \lesssim f \tag{4}$$

3.2 General Bounds

$$(1 - a) \leq e^{-a} \ \ \forall a > 0 \tag{5}$$

4 Quick Sort

One of the most well known randomized algorithms is Quick Sort. The pseudo code is given in Algorithm **??**.

Algorithm 1 Sort

if $x == []$ **then**
  return []
end if
Choose random $t \in [n]$
return $Sort([x_i | x_i < x_t]) + [x_t] + Sort(x_i | x_i \geq x_t])$

Question: Why should t be random?

What if $t = 1$ always? $\Theta(n^2)$ time for sorted input. Note, we don't need an adversary for bad inputs. In practice it is likely that lists that need to be sorted will consist of a mostly sorted list with a few new entries.

Question: How should we analyze the time complexity?

Let $Z_{ij} :=$ event that ith largest element is compared to the jth largest element at any time during the algorithm. Because the pivot is removed, each comparison can happen at most once.

Time is proportional to the number of comparisons $= \sum_{i<j} Z_{ij}$

$Z_{ij} = 1 \iff$ the first pivot in $\{i, i+1, \ldots, j\}$ is i or j. To see this, note that if the pivot is $< i$ or $> j$, then i and j aren't compared, while if the pivot is $> i$ and $< j$, then the pivot splits i and j into two different recursive branches so they will never be compared.

Thus, we have

$$Pr[Z_{ij}] = \frac{2}{j - i + 1}$$

.

Note that the Z_{ij}'s are complex random variables that are highly dependent on each other; however, we are only interested in expectations of Z_{ij} so we can leverage the linearity of expectation to greatly simplify things:

$$\begin{aligned}
\mathbb{E}[Time] &\lesssim \mathbb{E}\left[\sum_{i<j} Z_{i,j}\right] \\
&= \sum_{i<j} \mathbb{E}\left[Z_{i,j}\right] && \text{(by linearity of expectation)} \\
&= \sum_{i} \sum_{i<j} \mathbb{E}\left[\frac{2}{j-i+1}\right] \\
&= 2 \cdot \sum_{i} \mathbb{E}\left[\frac{1}{2} + \cdots + \frac{1}{n-i+1}\right] \\
&\leq 2 \cdot n \cdot (H_n - 1) \\
&\leq 2 \cdot n \cdot \log n
\end{aligned}$$

$$
\begin{aligned}
E[time] &\approx E[\sum_{i<j} Z_{ij}] \\
&= \sum_{i<j} E[Z_{ij}] && \text{(by linearity of expectation)} \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= 2\sum_{i=1}^{n} \sum_{k=2}^{n-j+1} \frac{1}{k} \\
&\leq 2\sum_{i=1}^{n} (\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) \\
&\leq 2\sum_{i=1}^{n} H_n - 1 && \text{(nth harmonic number, } H_n = \sum_{k=1}^{n} \frac{1}{k}) \\
&\leq 2n \log n
\end{aligned}
$$

5 Min Cuts

Assume we have a graph $G = (V, E)$, where V denotes the vertex set and E the edge set, with $n = |V|$ nodes and $m = |E|$ edges.

Definition The **(s,t) cut** is the set $S \subseteq V$ with $s \in S$, $t \notin S$.

Definition The **cost of a cut** is the number of edges e with one vertex in S and the other vertex not in S.

Definition The **min(s,t) cut** is the (s,t) cut of minimum cost.

Definition The **Global Min Cut** is the min (s,t) cut over all $s, t \in V$.

Question What is the time complexity to complete the global min cut?

Every global cut is an s-t cut for some $s, t \in V$ so we can iterate over all choices of s and t and pick the smallest (s, t) cute. So we can simply run $O(n^2)$ Ford-Fulkersons over all pairs.

The run-time complexity can be reduced by a factor of n by noting that each node must be in one of the two partitioned subsets. Thus, we can just select any node s and compute a $\min(s, t)$ cut for all other vertices and return the cut of smallest cost. This results in $O(n)$ Ford-Fulkersons, resulting in compleixty $O(n \cdot nm) = O(n^2 m)$.

5.1 Algorithm 1

Question: What about with a randomized algorithm?

Algorithm 1 $O(n^2)$

while $n > 2$ **do**
  choose a random edge
  contract into a single vertex
end while

Contracting an edge means that we remove that edge and combine the two vertices into a super-node. We note that self-loops thus formed are removed but any resulting parallel edges are *not* removed. This means at every step, we have a multi-graph without any self-loops.

This algorithm will yield a cut. The minimum cut? Probably not. But maybe! If the graph is somewhat dense, then choosing an edge in the minimum cut is rather unlikely at the beginning...and very likely at the end. We will prove the following:

Lemma: Algorithm 1 succeeds with $\geq \frac{2}{n^2}$ probability

To get started, let's prove the following lemma.

Lemma 1: The chance the algorithm fails in round 1 is $\leq \frac{2}{n}$

Proof: The algorithm "fails" in step i if in that step, you choose to contract an edge in the minimum cut. We know that the chance of failure in the first step is $\frac{OPT}{m}$ where OPT = cost of true min cut = $|E(S, \bar{S})|$. We need a way to bound $|E(S, \bar{S})|$.

For all $u \in V$, let $d(u)$ be the degree of u. Let's use the node degrees to bound OPT.

$$\begin{aligned} OPT = |E(S, \bar{S})| &\leq \min_u d(u) \\ &\leq \frac{1}{n} \sum_{u \in V} d(u) \\ &\leq \frac{2 \cdot m}{n} \end{aligned}$$

Dividing both sides by m we have

$$\frac{|E(S, \bar{S})|}{m} \leq \frac{2}{n} \tag{6}$$

□

We can continue this analysis in each subsequent round. For example, in Round 2, we have the collapsed graph $G' = (V', E')$. Every set $S' \in V'$ corresponds to $S \in V$ and $cost(S') = cost(S)$ since the min-cut doesn't change. So again algorithm will fail with probability $\leq \frac{2}{n-1}$.

Thus, we have

$$\begin{aligned}
\mathbb{P}(\text{fail in } 1^{st} \text{ step}) &\leq \frac{2}{n} \\
\mathbb{P}(\text{fail in } 2^{nd} \text{ step} \mid \text{success in } 1^{st} \text{ step}) &\leq \frac{2}{n-1} \\
&\vdots \\
\mathbb{P}(\text{fail in } i^{th} \text{ step} \mid \text{success till } (i-1)^{th} \text{ step}) &\leq \frac{2}{n+1-i}
\end{aligned}$$

We now proceed with the proof of the original Lemma.

Lemma: Algorithm 1 succeeds with $\geq \frac{2}{n^2}$ probability

Proof: Let $Z_i :=$ the event that an edge from the cut set is picked in round i.

We have that

$$Pr[Z_i | \bar{Z}_1 \cap \bar{Z}_2 \cap \cdots \cap \bar{Z}_{i-1}] \leq \frac{2}{n+1-i}$$

Thus the probability of success is given by

$$Pr(Success) = Pr[\bar{Z}_1 \cap \bar{Z}_2 \cap \cdots \cap \bar{Z}_{n-2}] \geq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \ldots (1 - \frac{2}{3})$$

We can simplify this by turning things into simple fractions to get

$$\begin{aligned}
\mathbb{P}(\text{Success}) &\geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{2}{1} \cdot \frac{1}{1} \\
&= \frac{2}{n(n-1)} \\
&\geq \frac{2}{n^2}
\end{aligned}$$

□

So how do we use this? This algorithm seems pretty likely to fail if run only once.

5.2 Algorithm 2

Question: Can we improve the success rate of Algorithm 1 by running it multiple times?

If we run Algorithm 1 k times and pick the set of min cost, then

$$Pr(failure) \leq (1 - \frac{2}{n^2})^k \leq e^{\frac{-2k}{n^2}} \tag{7}$$

Algorithm 2

$i = 0$
while $i < k$ **do**
  Run Algorithm 1
  $i = i + 1$
end while
pick set of min cost

To get this we use the useful fact that

$$(1 - a) \leq e^{-a} \quad \forall a > 0 \tag{8}$$

Plugging in $2/n^2$ for a and raising both sides to the kth power we get the above result.

What's neat is that now we can set $k = \frac{n^2}{2} \log(\frac{1}{\delta})$ to get $1 - \delta$ success probability.

Question What is the run time of Algorithm 2? The only real computation is the contraction. At every step of the algorithm, we have to contract an edge. This takes at most $\mathcal{O}(n)$ time. We do at most n such contractions. Thus for every run, we need $\mathcal{O}(n^2)$ time and the total time is $O(n^2 \cdot k)$. If we set $k = \frac{n^2}{2} \log(\frac{1}{\delta})$, then we have complexity $O(n^4 \log(\frac{1}{\delta}))$.

We can choose δ to be, say $1/4$, and thus $k = \frac{n^2}{2} \log 4$ and this will ensure that within $\mathcal{O}(n^4)$ time the algorithm succeeds with probability at least $3/4$.

5.3 Algorithm 3

Question: Can we make Algorithm 1 more efficient?

Initial stages of the algorithm are very likely to be correct. In particular, the first step is wrong with probability at most $2/n$. As we contract more edges, failure probability goes up. Moreover, earlier stages take more time compared to later ones.

Idea: Let us redistribute our iterations. Since earlier ones are more accurate and slower, why not do less of them at the beginning, and increasingly more as the number of edges decreases?

Algorithm 3

Repeat twice
    take $n - \frac{n}{\sqrt{2}}$ steps of contraction
    recursively apply this algorithm
take better result

This results in the following runtime:

$$T(n) = O(n^2) + 2T(\frac{n}{\sqrt{2}}) = n^2 \log n$$

The success probability is now given by:

$$\begin{aligned} p(n) &= 1 - (\text{failure probabilty of one branch})^2 \\ &= 1 - (1 - \textit{success in one branch})^2 \\ &= 1 - (1 - (\frac{\frac{n}{\sqrt{2}}}{n})^2 \cdot p(\frac{n}{\sqrt{2}}))^2 \\ &= 1 - \left(1 - \frac{1}{2}\, p\left(\frac{n}{\sqrt{2}}\right)\right)^2 \\ &= p\left(\frac{n}{\sqrt{2}}\right) - \frac{1}{4}\, p\left(\frac{n}{\sqrt{2}}\right)^2 \end{aligned}$$

To solve this recursion, we let $x = \log_{\sqrt{2}} n$. Thus, setting $f(x) = p(n)$, we get

$$f(x) = f(x-1) - f(x-1)^2$$

We observe that setting $f(x) = \frac{1}{x}$ gives:

$$f(x-1) - f(x) = \frac{1}{x-1} - \frac{1}{x} = \frac{1}{x(x-1)} \approx \frac{1}{(x-1)^2} = f(x-1)^2$$

Hence, we have $p(n) = \mathcal{O}\left(\frac{1}{\log n}\right)$.

5.4 Algorithm 4

We can now run Algorithm 3 multiple times.

Alg 4: Repeat algorithm $O(\log n \log \frac{1}{\delta})$ times.