CS 125 Section #10 **Randomized & Approximation algorithms** November 15, 2016

1 Probability

First, recall a couple useful facts from last time about probability:

- Linearity of expectation: $\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$ even when X and Y are not independent.
- Geometric sums: $\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$ (for $0 < |p| < 1$) and $\sum_{i=0}^{n} p^i = \frac{1-p^{n+1}}{1-p}$ (for all p).
- For random numbers X which only take on nonnegative integer values, $\mathbb{E}(X) = \displaystyle\sum_{j=0}^{\infty} \mathbb{P}(X > j)$.
- Geometric random variables: If some event happens with probability p, the expected number of tries we need in order to get that even to occur is $1/p$.
- Markov's inequality: For any nonnegative random variable X and $\lambda > 0$, $\mathbb{P}(X > \lambda \cdot \mathbb{E}X) < \frac{1}{\lambda}$.

Exercise (Fixed points in permutations). *Suppose we pick a uniformly random permutation on n elements. First, how would we do it? And then, what is the expected number of fixed points it has? What is the probability that there is a fixed point?*

Solution.
One way to pick a permutation uniformly at random would be to start with the list $A = \{1, 2, \ldots, n\}$ in sorted order, and then switch $A[i]$ with a uniformly random element among $A[i], \ldots, A[n]$, for $i = 1, 2, \ldots, n-1$. It's easy to see by induction that this gives a uniformly random permutation on the list - as with the first step we pick a uniformly random element for the first position, and then we run the same algorithm on the remaining list.

Now let X stand for the random variable that is the number of fixed points of a random permutation π. Then

$$X = \sum_{i=1}^{n} X_i$$

where X_i is the indicator of the event $\pi(i) = i$. But $\mathbb{E}(X_i) = (n-1)!/n! = 1/n$ by an easy counting argument, so by additivity of expectation,

$$\mathbb{E}(X) = \sum_{i=1}^{n} \mathbb{E}(X_i) = \sum_{i=1}^{n} 1/n = 1$$

To determine the probability, we use principle of inclusion/exclusion. The probability that elements $i_1, ..., i_k$ are fixed is $\binom{n}{k} \cdot \frac{(n-k)!}{n!} = \frac{n!}{k!(n-k)!} \cdot \frac{(n-k)!}{n!} = \frac{1}{k!}$. So the probability that there is at least 1 fixed point is

$$\boxed{\sum_{k=1}^{n} \frac{(-1)^{k+1}}{k!}}.$$

Note that as $n \to \infty$, this tends towards $1/e$.

Exercise (Special Dice)**.** *Consider the following game: there are three dice, each with its faces labeled by integers. You are allowed to choose what you think is the "best" die, after which your opponent will choose another die. Each of you roll yours, and the person who rolls the larger number receives* \$1 *from the other player. Should you always play?*

Solution.
No; suppose that the dice have labels $(2,2,2,2,2,2)$, $(0,0,4,4,4,4)$, $(1,1,1,1,5,5)$. Then, we see that each die is better than the previous one in turn, so the second player to choose will always have an advantage.

Exercise (Coupon Collector)**.** *There are n coupons $C_1, \ldots, C_n$. Every time you buy a magazine, a uniformly random coupon C_i is inside. How many magazines do you have to buy, in expectation, to get all n coupons? How many magazines do you have to buy to get all n coupons with a failure probability of at most P?*

Solution.
a) You need $\Theta(n \log n)$ to get all n coupons.

b) To get all n coupons with probability P, it's enough to buy $\Theta(n \log n \log \frac{1}{P})$.

Proofs:

a) Once you've collected k coupons, buying a new magazine gives you a new coupon with probability $1 - \frac{k}{n} = \frac{n-k}{n}$. Thus you need to flip $\frac{n}{n-k}$ coupons to get the $(k+1)$-st one. Thus the expected number of magazines you have to buy is $\sum_k \frac{n}{n-k} = n \cdot \sum_{t=1}^{n} \frac{1}{t}$ (substitute $t = n - k$). This is the n-th Harmonic number, which we can check is $\Theta(n \log n)$ by an integral.

b) By Markov's inequality, the probability you don't get all coupons after $k \cdot$ (expected number of steps) is at most $\frac{1}{k}$. Thus the probability this happens $k = \log \frac{1}{P}$ times in a row is $2^{-\log \frac{1}{P}} = P$.

Exercise (Election Problem (Challenge))**.** *America is having its next presidential election between two candidates, and for each of the n states you know the probability that candidate 1 wins that state is P_i. For simplicity we'll assume that independent candidates don't exist, so the probability candidate 2 wins is $1 - P_i$.*

Because of the electoral college, state i is worth S_i points: when a candidate wins a state, he/she gets all S_i points. To win, a candidate must get a strict majority of the electoral votes. Assuming a tie isn't possible, how do you efficiently approximate the probability that candidate 1 wins?

Solution.
If the S_i values are small enough, you can actually obtain a direct computation by knapsack.

But what if the S_i values are very large? Let's say you only need a percentage with two digits after the decimal point (e.g., 61.29%).

What if we just simulate the election randomly? For each state we generate a random number between 0 and 1, and then we compute the outcome of the election: if our number is below P_i, candidate 1 wins; else candidate 2 wins. Repeat this some large number of times, and we should get fairly accurate results.

But how many iterations do we need to perform to get it quite this accurate? For a binomial distribution like this the standard deviation is $\sqrt{\frac{p(1-p)}{n}}$ (where p is the final probability), so if we want d digits of precision, the value for n that succeeds with a probability of $\frac{1}{2}$ will be $O(10^{2d})$. In other words, with a few billion iterations, we can be pretty confident that we have four digits of accuracy.

(Fun story: I actually saw this on FiveThirtyEight, and I was disappointed because they used this Monte Carlo simulation when they really should have just solved it deterministically with a knapsack.)

2 Probabilistic Algorithms

We have looked at probailistic algorithsm with bounded running time, but that may be incorrect with a small probability. More specifically:

- **RP** is the class of languages for which there exists a polynomial-time randomized algorithm which always returns correctly for inputs in the language, and which returns correctly at least half the time for inputs not in the language.
- **BPP** is the class of languages for which there exists a polynomial-time ranomized algorithm which returns correctly at least $2/3$ of the time on any input.

Note that the randomness is over the coins of the algorithm, not over the input.

Oftentimes, we can decrease the failure probability by running the algorithm multiple times. For example, if we use our algorithm to check some computation but can be fooled with some probability, we can just run it multiple times. Indeed, this can be used to show that it is sufficient for a **BPP** algorithm to be correct at least $\frac{1}{2} + \frac{1}{p(n)}$ of the time for any polynomial $p(n)$ in the length of the input.

2.1 Hashing

Remember from class that a hash function is a mapping $h : \{0, \dots, n-1\} \to \{0, \dots, m-1\}$. In general, because one of the goals of a hash function is to save space, we will see that $m << n$.

Hashing-based data structures are useful since they ideally allow for constant time operations (lookup, adding, and deletion), although collisions can change that if, for example, n is too large or you use a poorly chosen hash function. (Why?)

In practice, there are several ways to deal with collisions while hashing, such as:

- Chaining: Each bucket holds a set of all items that are hashed to it. Simply add x to this set.
- Linear probing: If $f(x)$ already has an item, try $f(x) + 1, f(x) + 2$, etc. until you find an empty location (all taken mod m).
- Perfect hashing: If the set is known and fixed, generate a hash function for that specific set. However, this does not support insertions and deletions.
- Automatic resizing: Once the number of elements exceeds cm for some constant c, make the hash table larger by doubling m. In practice, a load factor of $c = 0.75$ is fairly common.

2.2 Document Similarity

Consider two sets of numbers, A and B. For concreteness, we will assume that A and B are subsets of 64 bit numbers. We may define the *resemblance* of A and B as

$$\text{resemblance}(A, B) = R(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

The resemblance is a real number between 0 and 1. Intuitively, the resemblance accurately captures how close the two sets are. To calculate this, we could sort the sets and compare, but this is still rather slow, so we'd like to do better. We will do this by computing an approximation to the set resemblance, rather than the exact value.

Let π_1 be some random permutations of the numbers $1, 2, \ldots, 2^{64}$ (that is, it is a bijection $\{1, \ldots, 2^{64}\} \to \{1, \ldots, 2^{64}\}$). Let $\pi_1(A) = \{\pi_1(x) : x \in A\}$.

Exercise. *When does* $\min\{\pi_1(A)\} = \min\{\pi_1(B)\}$*?*

Solution.
This happens only when there is some element x satisfying $\pi_1(x) = \min\{\pi_1(A)\} = \min\{\pi_1(B)\}$. In other words, the element x that is the minimum element in the set $A \cup B$ has to be in the intersection of the sets $A \cap B$.

Exercise. *What is* $Pr[\min\{\pi_1(A)\} = \min\{\pi_1(B)\}]$*?*

Solution.
If π_1 is a random permutation, then every element in $A \cup B$ has equal probability of mapping to the minimum element after the permutation is applies. That is, for all x and y in $A \cup B$,

$$\mathbf{Pr}[\pi_1(x) = \min\{\pi_1(A \cup B)\}] = \mathbf{Pr}[\pi_1(y) = \min\{\pi_1(A \cup B)\}].$$

Thus, for the minimum of $\pi_1(A)$ and $\pi_1(B)$ to be the same, the minimum element must lie in $\pi_1(A \cap B)$. Hence

$$\mathbf{Pr}[\min\{\pi_1(A)\} = \min\{\pi_1(B)\}] = \frac{|A \cap B|}{|A \cup B|}.$$

But this is just the resemblance $R(A, B)$!

This gives us a way to estimate the resemblance. Instead of taking just one permutation, we take many– say 100. For each set A, we preprocess by computing $\min\{\pi_j(A)\}$ for $j = 1$ to 100, and store these values. To estimate the resemblance of two sets A and B, we count how often the minima are the same, and divide by 100. It is like each permutation gives us a coin flip, where the probability of a heads (a match) is exactly the resemblance $R(A, B)$ of the two sets.

If you have some more experience with this type of probability, you might think about how we could bound the accuracy of this method – that is, you want to make some statement of the form "the algorithm is within ϵ of the correct set resemblence with probability at least $1 - \delta$" for some $\epsilon, \delta \in [0, 1]$.

Now, we can use this to think about similarity between documents via a method called *shingling*. That is, divide the document into pieces of consecutive words and hash the results into a set S_D. Once we have the shingles for the document, we associate a document *sketch* with each document. The sketch of a document S_D is a list of say 100 numbers: $(\min\{\pi_1(S_D)\}, \min\{\pi_2(S_D)\}, \min\{\pi_3(S_D)\}, \ldots, \min\{\pi_{100}(S_D)\})$.

Exercise. *How well does this method do?*

More concretely, suppose that we mark two documents as "similar" if their sketches match on at least 90 values. What is the probability that this happens if the set resemblence of the two sets of shingles is r?

Solution.
For each permutation π_i, the probability that two documents A and B have the same value in the ith position of the sketch is just the resemblance of the two documents $R(A, B) = r$. Hence, the probability $p(r)$ that at least 90 out of the 100 entries in the sketch match is

$$p(r) = \sum_{k=90}^{100} \binom{100}{k} r^k (1-r)^{100-k}.$$

What does $p(r)$ look like as a function of r? Notice that $p(r)$ stays very small until r approaches 0.9, and then quickly grows towards 1. This is exactly the property we want our scheme to have– if two documents are not similar, we will rarely mistake them for being similar, and if they are similar, we are likely to catch them!

3 Random Walks

A random walk is an iterative process on a set of vertices V. In each step, you move from the current vertex v_0 to each $v \in V$ with some probability. The simplest version of a random walk is a one-dimensional random walk in which the vertices are the integers, you start at 0, and at each step you either move up one (with probability 1/2) or down one (with probability 1/2).

2-SAT: In lecture, we gave the following randomized algorithm for solving 2-SAT. 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.

We used a random walk with a completely reflecting boundary at 0 to model our randomized solution to 2-SAT. Fix some solution S and keep track of the number of variables k consistent with the solution S. In each step, we either increase or decrease k by one. Using this model, we showed that the expected running time of our algorithm is $O(n^2)$.

Exercise. *This weekend, you decide to go to a casino and gamble. You start with k dollars, and you decide that if you ever have $n \geq k$ dollars, you will take your winnings and go home. Assuming that at each step you either win* \$1 *or lose* \$1 *(with equal probability), what is the probability that you instead lose all your money?*

Solution.
Let P_k be the probability that you win if you currently have \$$k$

Set up a recurrence relation: $P_0 = 0$, $P_n = 1$, $P_k = 1/2P_{k+1} + 1/2P_{k-1}$. It seems like P_k is the average of P_{k-1} and P_{k+1} for all $0 < k < n$, which suggests a linear function some sort. Thus, the linear function that would take $P_0 = 0$ and $P_n = 1$ is $P_k = k/n$, which we confirm to be a correct solution.

Solution: $P_k = k/n$, so you lose all your money with probability $1 - k/n$

4 Approximation algorithms

4.1 k-Approximations

Approximation algorithms help us get around the lack of efficient algorithms for NP-hard problems by taking advantage of the fact that, for many practical applications, an answer that is "close-enough" will suffice. An approximation ratio, k, gives the ratio that bounds the solutions generated by our k-approximation.

The goal is to obtain an approximation ratio as close to 1 as possible.

4.2 Examples of approximation algorithms

1. **Minimum Vertex Cover**: Given a graph $G = (V, E)$, can you find the minimal cardinality set of vertices $S \subseteq V$ such that for all $(u, v) \in E$, we have at least one of $u \in S$ or $v \in S$.

   A 2-approximation algorithm for this problem is relatively straight forward. Just greedily find a $(u, v) \in E$ for which $u \notin S_i$ and $v \notin S_i$, and construct $S_{i+1} = S_i \cup \{u, v\}$. We have $S_0 = \emptyset$. When not such (u, v) exist, return S.

   Furthermore, because this is a 2-approximation, it can be shown that $|S| \leq 2|\text{OPT}|$ where OPT is a minimum set cover.

2. **Maximum Cut**: Given a graph $G = (V, E)$, can you create two sets S, T such that $S \cap T = \emptyset, S \cup T = V$, and the number of $(u, v) \in E$ that cross the cut is maximal?

   For this problem, we've covered two 2-approximation algorithms. The first is to randomly assign each $u \in V$ into either S or T. The second is to deterministically and iteratively improve our maximal cut by moving edges from S to T or vice-versa.

3. **Euclidean Traveling Salesman Problem**: Give a set of points (x_i, y_i) in Euclidean space with l_2 norm, find the tour of minimum length that travels through all cities.

   A 2-approximation algorithm for this problem can be achieved by short-circuiting DFS on minimum spanning tree. Furthermore, the approximation can be improved to $\frac{3}{2}$-approximation. Both require the triangle inequality to hold.

4. **Max Sat**: Find the truth assignment for variables which satisfies the largest number of OR clauses.

   We actually have two approximation algorithms. The first is by random coin-flipping (does best on long clauses). The second is randomized rounding after applying linear programming relaxation (does best on short clauses). We can also combine both for an even better solution.

4.3 Practice problems

Exercise. ***Minimum Set Cover***
We are given inputs to the minimum set cover problem. The inputs are a finite set $X = \{x_1, x_2, ..., x_n\}$*, and a collection of subsets* $\mathcal{S}$ *of* X *such that* $\bigcup_{S\in\mathcal{S}} S = X$*. Find the subcollection* $\mathcal{T} \subseteq \mathcal{S}$ *such that the sets of* $\mathcal{T}$ *cover* X*, that is:*

$$\bigcup_{T\in\mathcal{T}} T = X$$

Recall minimum set cover seeks to minimize the size of the selected subcollection (minimize $|\mathcal{T}|$*). Let* k *be the number of times the most frequent item appears in our collection of subsets,* $\mathcal{S}$*. Find a* k*-approximation algorithm for minimum set cover.*
(Hint: Use linear program relaxation.)

Solution.
Following the hint, we can set this up with an integer linear program. We create decision variables z_S for each $S \in \mathcal{S}$. Then the problem can be phrased as:

$$\begin{aligned} \text{minimize} \quad & \sum_{S\in\mathcal{S}} z_S \\ \text{subject to} \quad & \sum_{S:x_j\in S, S\in\mathcal{S}} z_S \geq 1, \qquad j = 1, ..., n \\ & z_S \in \{0,1\}, \quad S \in \mathcal{S} \end{aligned}$$

Intuitively, $z_S = 1$ if $S \in \mathcal{T}$, otherwise $z_S = 0$. Our goal is to minimize the number of selected subsets subject to the constraint that for each of the x_i items, at least one of the selected subsets must contain that item.

We can then take the above ILP and relax the integer constraint to create the below LP, which is solvable in poly-time:

$$\begin{aligned} \text{minimize} \quad & \sum_{S\in\mathcal{S}} z_S \\ \text{subject to} \quad & \sum_{S:x_j\in S, S\in\mathcal{S}} z_S \geq 1, \qquad j = 1, ..., n \\ & 0 \leq z_S \leq 1, \quad S \in \mathcal{S} \end{aligned}$$

Given the above solution, we look at the achieved value of each z_S for $S \in \mathcal{S}$. If $z_S \geq \frac{1}{k}$, then $S \in \mathcal{T}$, otherwise $S \notin \mathcal{T}$.

The claim is that the above is a k-approximation. For correctness, note that the maximum number of sets S for which $x_j \in S$ is at most k. Therefore, our constraint sums over at most k sets containing item x_j, and this sum must be ≥ 1. Then there must exists at least one set containing x_j for which $z_S \geq \frac{1}{k}$. Therefore, for each item, we take at least one $S \in \mathcal{T}$ such that $x_j \in S$.

To see the k-approximation, let OPT be the optimal solution. Then our algorithm returns a set which is at most kOPT.

Exercise. ***Maximum Independent Set***
Recall that the maximum independent set problem for a graph $G = (V, E)$ *consists of finding a set* $T \subseteq V$

such that for all $u, v \in T$, $(u, v) \notin G$ and $|T|$ is maximal.

Find a $\frac{1}{d+1}$-approximation algorithm, for this problem where d is the maximum degree of any vertex in the graph.

Solution.
Our graph G has $n = |V|$ vertices. There are a total of $n!$ possible orderings of these vertices. Pick one such ordering uniformly at random. Then process each vertex in turn based on the selected ordering. Begin with $S = \emptyset$. Then at each step, if $u \in V$ and for all $(u, v), (v, u) \in E$, $v \notin S$, add v to S. Otherwise, continue, mark v as processed and continue to the next.

First, note that the above algorithm, by construction, returns an independent set (we only add vertices to S which are disjoint). Now we just need to show that the returned independent set $S \subseteq V$ has the property such that $|S| \geq \frac{1}{d+1} OPT$.

Let X_u be an indicator variable for whether or not $u \in S$. Then the expected size of S is given by:

$$\begin{aligned}
\mathbb{E}(|S|) &= \mathbb{E}\left(\sum_{u \in V} X_u\right) \\
&= \sum_{u \in V} \mathbb{E}(X_u) && \text{(linearity of expectation)} \\
&= \sum_{u \in V} \Pr(X_u = 1) && \text{(fundamental bridge)} \\
&= \sum_{u \in V} \frac{1}{n_u + 1} && \text{(n_u is the number of neighbors of vertex u)} \\
&\geq \sum_{u \in V} \frac{1}{d + 1} \\
&= \frac{n}{d + 1} \\
&\geq \frac{OPT}{d + 1} && \text{(OPT is at most $|V|$)}
\end{aligned}$$

Note that we arrive at $\Pr(X_u = 1)$ by noting that vertex u is included in S iff it is the first vertex in its neighborhood to be processed. Given that the process order is selected uniformly at random, the probability u being processed first among its neighborhood is $\frac{1}{n_e+1}$.

From the above, it follows that our algorithm is a $\frac{1}{d+1}$ approximation.