Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 4-cycles

We use $G(n,p)$ to denote the distribution of graphs obtained by taking n vertices and for each pair of vertices i,j placing edge $\{i,j\}$ independently with probability p.

(a) Compute the expected number of edges in $G(n,p)$? **Solution:** By linearity of expectation, $p\binom{n}{2}$ since there are $\binom{n}{2}$ potential edges in the graph.

(b) Compute the expected number of 4-cycles in $G(n,p)$? **Solution:** Each individual 4-cycle has probability p^4 of appearing, since it consists of 4 edges. There are 3 possible 4-cycles on any particular set of 4 vertices, and there are $\binom{n}{4}$ sets of 4 vertices in the graph. So, by linearity of expectation, the answer is $(p^4)(3)\binom{n}{4} = p^4 \cdot \frac{n(n-1)(n-2)(n-3)}{8}$.

(c) Give a polynomial time randomized algorithm that takes in n as input and in poly(n)-time outputs a graph G such that G has no 4-cycles and the expected number of edges in G is $\Omega(n^{4/3})$. **Solution:** Let $\boldsymbol{G} \sim G(n,p)$ for $p = Cn^{-2/3}$; let $\boldsymbol{e}$ be the number of edges and let $\boldsymbol{c}_4$ be the number of 4-cycles in $\boldsymbol{G}$. Let $\boldsymbol{G'}$ be the graph obtained by deleting an edge from every 4-cycle, and output it; this deletion removes at most $\boldsymbol{c}_4$ edges, so the number of edges $\boldsymbol{e'}$ remaining in the graph satisfies: $\boldsymbol{e'} \geq \boldsymbol{e} - \boldsymbol{c}_4$, which means $\mathbf{E}[\boldsymbol{e'}] \geq \mathbf{E}[\boldsymbol{e}] - \mathbf{E}[\boldsymbol{c}_4] = \frac{pn(n-1)}{2} - \frac{p^4 n(n-1)(n-2)(n-3)}{4} \geq \frac{pn^2 - p^4 n^4}{4} = \frac{Cn^{4/3} - C^4 n^{4/3}}{4}$. Choosing C as, say, .5 finishes the proof, and each step takes polynomial time.

2 Universal Hashing

Let $[m]$ denote the set $\{0,1,...,m-1\}$. Recall that a family of functions $\mathcal{H}$ is *universal* if for any $x \neq y, \Pr_{h\sim\mathcal{H}}[h(x) = h(y)] \leq 1/m$. That is, the chance that $h(x) = h(y)$ if we sample h uniformly at random from $\mathcal{H}$ is at most $1/m$.

For each of the following families of hash functions, determine whether or not it is universal. If it is universal, determine how many random bits are needed to choose a function from the family.

(a) $H = \{h_{a_1,a_2} : a_1, a_2 \in [m]\}$, where m is a fixed prime and

$$h_{a_1,a_2}(x_1,x_2) = a_1x_1 + a_2x_2 \mod m$$

Notice that each of these functions has signature $h_{a_1,a_2} : [m]^2 \to [m]$, that is, it maps a pair of integers in $[m]$ to a single integer in $[m]$.

(b) H is as before, except that now $m = 2^k$ for $k > 1$ is some fixed power of 2.

(c) H is the set of all functions $f : [m] \to [m-1]$.

Solution:

(a) The hash function is universal. The universality proof is the same as the one in the textbook (only now we have a 2-universal family instead of 4-universal). To reiterate, assume we are given two distinct pairs of integers $x = (x_1, x_2)$ and $y = (y_1, y_2)$. Without loss of generality, let's assume that $x_1 \neq y_1$. If we chose values a_1 and a_2 that hash x and y to the same value, then

$$a_1x_1 + a_2x_2 \equiv a_1y_1 + a_2y_2 \mod m.$$

We can rewrite this as

$$a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \mod m.$$

Let $c \equiv a_2(y_2 - x_2) \mod m$. Since m is prime and $x_1 \neq y_1$, $(x_1 - y_1)$ must have a unique inverse. So $a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \mod m$ if and only if $a_1 \equiv c(x_1 - y_1)^{-1} \mod m$, which will only happen with probability $1/m$.

We need to randomly pick two integers in the range $[0, \ldots, m-1]$, so we need $2 \log m$ random bits.

(b) This family is not universal. Consider the following inputs: $(x_1, x_2) = (0, 2^{k-1})$ and $(y_1, y_2) = (2^{k-1}, 0)$. We then have

$$h_{\alpha_1,\alpha_2}(x_1, x_2) = 2^{k-1}\alpha_2 \mod 2^k$$

and

$$h_{\alpha_1,\alpha_2}(y_1, y_2) = 2^{k-1}\alpha_1 \mod 2^k$$

Now notice that if α_2 is even (i.e. with probability $1/2$) then $h_{\alpha_1,\alpha_2}(x_1, x_2) = 0 \mod 2^k$ otherwise (if α_2 is odd) $h_{\alpha_1,\alpha_2}(x_1, x_2) = 2^{k-1} \mod 2^k$; likewise for α_1. So we get that

$$h_{\alpha_1,\alpha_2}(x_1, x_2) = h_{\alpha_1,\alpha_2}(y_1, y_2)$$

with probability $1/2 > 1/2^k$, so the family is not universal.

(c) This family is universal. To see that, fix $x, y \in \{0, 1, \ldots, m-1\}$ with $x \neq y$. Now we need to figure out the following: how many (out of the $(m-1)^m$ in total) functions $f : [m] \to [m-1]$ will collide on x and y, i.e. $f(x) = f(y) = k$, for some fixed $k \in [m-1]$. Well, there are $(m-1)^{m-2}$ different functions $f : [m] \to [m-1]$ that have the property $f(x) = f(y) = k$ (because I just fixed the output of 2 inputs to some fixed $k \in [m-1]$ and allow the output of f for all other inputs to range over all $m-1$ possible values). Finally, ranging over all $m-1$ values of k, we get that there are $(m-1)^{m-1}$ functions $f : [m] \to [m-1]$ with the property $f(x) = f(y)$. So the probability of picking one such f is exactly $\frac{(m-1)^{m-1}}{(m-1)^m} = \frac{1}{m-1}$.

There are $(m-1)^m$ functions in this family, so we need $\log(m-1)^m = m \log(m-1)$ bits to distinguish between them.

3 Polynomial Identity Testing

Suppose we are given a polynomial $p(x_1, \ldots, x_k)$ over the reals such that p is not in any normal form. For example, we might be given the polynomial:

$$p(x_1, x_2, x_3) = (3x_1 + 2x_2)(2x_3 - 2x_1)(x_1 + x_2 + 3x_3) + x_2$$

We want to test if p is equal to 0, meaning that $p(a_1, a_2, \ldots, a_k) = 0$ on all real inputs $a_1, a_2, \ldots a_k$. This suggests a simple algorithm to check if p equals 0: test p on some $a_1, \ldots, a_k$ and say p equals 0 if and only if $p(a_1, \ldots, a_k) = 0$. However, this algorithm will fail if $p(a_1, \ldots, a_k) = 0$ and p does not equal 0. We will show through the following questions that if the degree of p is d, then for any finite set of reals S, by randomly choosing $a_1, \ldots, a_k \in S$, we get $Pr[p(a_1, \ldots, a_k) = 0] \leq \frac{d}{|S|}$. So by choosing a large enough subset, S, we are very likely to correctly determine whether p is equal to 0 or not. Note that the degree of, say, $p(x, y) = x^2y^2 + x^3$ is 4, since the monomial x^2y^2 has degree $4 = 2 + 2$. You may assume here that all basic operations on the reals take constant time.

(a) First let's see why a probabilistic algorithm might be necessary. Give a naive deterministic algorithm to test if a given polynomial p is equal to 0. What is the worst-case runtime of your algorithm?

(b) Show that if p is univariate and degree d, then $Pr[p(a_1) = 0] \leq \frac{d}{|S|}$

(c) (Challenge question) Show by induction on k that for all nonzero degree-d polynomials p on k variables:

$$Pr[p(a_1, \ldots, a_k) = 0] \leq \frac{d}{|S|}$$

Hint: Write $p(x_1, \ldots, x_n) = \Sigma_{i=0}^{d} p_i(x_1, \ldots, x_{n-1}) x_n^i$

(d) Use the bound you found in the last part to show a way to check if two polynomials p and q are identical (i.e., yield the same outputs on all inputs).

Solution:

(a) We can multiply out all parentheses and determine if there is any monomial with a non-zero coefficient. This takes exponential time in the worst case (e.g., it takes 2^k time if $p(x_1, x2) = (x_1 + x_2)(2x_1 + 2x_2) \ldots (kx_1 + kx_2)$)

(b) A degree d univariate polynomial can have at most d zeros. In the worst case, all d zeros are in S, yielding the probability $Pr[p(a_1) = 0] = \frac{d}{|S|}$.

(c) This result is known as the Schwartz-Zippel Lemma. We will write the polynomial as in the hint, with the n-variable polynomial $p(x_1, \ldots, x_n)$ written as a univariate polynomial in x_n after fixing $x_1, \ldots, x_{n-1}$: $\Sigma_{i=0}^{d} p_i(x_1, \ldots, x_{n-1}) x_n^i$. Note that some of these p_i polynomials will be identically zero, and they will have varying degrees – since the total degree of p is d, the degree of p_i can only be at most $d - i$ (or else $p_i(x_1, \ldots, x_{n-1}) x_n^i$ would have degree greater than d). Let's consider the 'largest-j nonzero polynomial p_j; this has degree at most $d - j$, and since j is as large as possible, the univariate polynomial in x_n that we get after fixing $x_1, \ldots, x_{n-1}$ has degree at most j.

Note that $p(x_1, \ldots, x_n) = 0$ requires either every $p_i(x_1, \ldots, x_{n-1})$ to be zero (in particular, $p_j(x_1, \ldots, x_{n-1}) = 0$), or x_1 to be a zero of the univariate polynomial. Since p_j has degree at most $d - j$, the first event happens with probability at most $\frac{d-j}{|S|}$ by the inductive hypothesis, and since the univariate polynomial has degree j and thus at most j zeros in the case where the first event does not happen, the second event happens with probability at most $\frac{j}{|S|}$

The probability that one of the above events happens is at most the sum of their probabilities, or at most $\frac{d-j+j}{|S|} = \frac{d}{|S|}$.

This completes the correctness analysis of the randomized algorithm given in the problem description. Interestingly, although this simple randomized algorithm works well, no efficient deterministic algorithm has been found.

(d) Given $p(x_1, \ldots, x_k)$ and $q(x_1, \ldots, x_k)$, we create a new polynomial $r(x_1, \ldots, x_k) = p(x_1, \ldots, x_k) - q(x_1, \ldots, x_k)$. We know r is equal to zero if and only if p and q are equivalent, so we can just apply our algorithm from the last part to r.

4 Monte Carlo Games

Let's suppose we have a Monte Carlo algorithm (a randomized algorithm which has a deterministic bound on its runtime, but which only outputs the correct answer some of the time). Call this algorithm A; then $A(x, r)$ is the output of A on input x and random bits r. In this question, we will think of A as a distribution over many deterministic algorithms. Convince yourself that this makes sense: after all, if we fix a setting to the random bits r, we get $A_r(x)$, which is a deterministic algorithm (which may be wrong on some inputs). Let's fix a set of algorithms S (say, polynomial-time algorithms). Note that A has whatever property defines S if and only if it is a distribution over only algorithms in S (for example, we say a Monte Carlo algorithm is polynomial time if and only if it runs in polynomial time for all settings to the randomness, which is equivalent to all the deterministic algorithms in its distribution running in polynomial time).

We will define a function $c(a, x)$ which indicates whether the deterministic algorithm $a \in S$ is correct on input x; $c(a, x) = 1$ if a is correct on input x, and 0 if it is incorrect.

Let's use this function to define a zero-sum game; the row player will choose a and the column player will choose x; then a payoff of $c(a, x)$ will go to the row player.

(a) Describe the action and goal of the row and column players. Interpret these in the setting of 'correctness of the randomly chosen algorithm' that we constructed the game from. *Hint: Since $c(a, x)$ is an indicator, $\mathbb{E}[c(a, x)] = \Pr[c(a, x) = 1]$.*

Solution: The row player chooses a distribution over algorithms in order to maximize $\min_x \mathbb{E}[c(a, x)] = \min_x \Pr[c(a, x) = 1]$; that is, maximize the probability the algorithm will be correct, for the worst-case input for that distribution. The column player chooses a distribution over inputs in order to minimize $\max_{a \in S} \mathbb{E}[c(a, x)] = \max_{a \in S} \Pr[c(a, x) = 1]$; that is, minimize the probability that any single algorithm will be correct.

(b) Using zero-sum game duality in conjunction with your interpretation above, what can we say about a problem if we know there exists a polynomial-time randomized algorithm which is correct with probability 2/3 on all inputs? What can we say if we know that there is a distribution of inputs on which no deterministic algorithm is correct with probability 2/3?

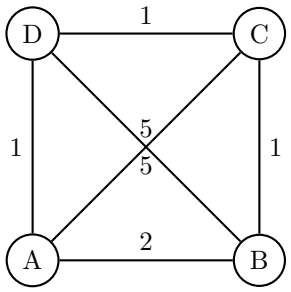
Hint: use the fact that a randomized algorithm induces a distribution over deterministic algorithms.

Solution: If there is a randomized algorithm which is correct with probability at least 2/3 on all inputs, then there is a distribution D_a for which $\min_x \Pr_{a \sim D_a}[c(a, x) = 1] \geq 2/3$; by weak zero-sum game duality, this means that for all distributions D_x, $\max_{a \in S} \Pr_{x \sim D_x}[c(a, x) = 1] \geq 2/3$; in other words, for any distribution over the inputs, there is a deterministic algorithm which is correct with probability at least 2/3 on an input randomly selected from that distribution. The second question is the contrapositive of the first: if we have this average-case hardness result for deterministic algorithms, we know there cannot be a good randomized algorithm.

Here we used correctness, but this argument works just as well for any 'cost' of an algorithm (time, space, ...) and is known as Yao's principle: The performance of the best deterministic algorithm on an average-case input is no better than the performance of the best randomized algorithm on a worst-case input with average-case randomness.

5 Traveling Salesman Problem

In the lecture, we learned an approximation algorithm for the Traveling Salesman Problem based on computing an MST and a depth first traversal. Suppose we run this approximation algorithm on the following graph:



The algorithm will return different tours based on the choices it makes during its depth first traversal.

1. Which DFS traversal leads to the best possible output tour?

2. Which DFS traversal leads to the worst possible output tour?

3. What is the approximation ratio given by the algorithm in the worst case for the above instance? Why is it worse than 2? (*Hint*: Consider the triangle inequality on the graph).

Solution:

1. **A-D-C-B**, **D-C-B-A**, **B-C-D-A**, or **C-D-A-B**
   Explanation:
   The MST is {(A,D),(D,C),(C,B)}. The optimal tour uses all of these edges.
   For example, if we start traversing the tree at A, then the only traversal would be to follow the tree and go through the vertices in the order D, C, B
   Another potential traversal would be to start at D, move to C, then B, and backtrack, before going to A.
   Notice that if we started at D and moved to A first, then we would have to visit C next, and this would not lead to the best possible output tour (as per part (b)).

2. **C-B-D-A** or **D-A-C-B**
   See above explanation.
   Notice that the worst possible tour overall, **D-C-A-B**, is not possible to be output by this algorithm, regardless of the DFS traversal used.

3. $\frac{12}{5}$
   From previous parts, the optimal tour length is 5 while the worst possible is 12. This is worse than 2 because our graph does not satisfy the triangle inequality, specifically $\ell(A, C) > \ell(A, B) + \ell(B, C)$ and $\ell(B, D) > \ell(A, B) + \ell(D, A)$, which is the necessary condition for our algorithm to guarantee an approximation ratio of 2.