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 Multiplicative Weights Intro

Multiplicative Weight Updates is an online algorithm, in which you take into account the advice of n experts. Every day you get more information on how good every expert is until the last day T.

Let's first define some terminology:

- $x_i^{(t)} :=$ probability that you "trust" expert i on day t (note that $x_i^{(t)} \in [0,1], \forall i, t$)
- $l_i^{(t)} :=$ loss you would incur on day t if you invested everything into expert i
- total regret: $R_T := \sum_{t=1}^{T} \sum_{i=1}^{n} x_i^{(t)} l_i^{(t)} - \min_{i=1,\dots,n} \sum_{t=1}^{T} l_i^{(t)}$

$\forall i \in [1,n]$ and $\forall t \in [1,T]$, the multiplicative update is as follows:

$$w_i^{(0)} = 1$$

$$w_i^{(t)} = w_i^{(t-1)} \cdot (1-\epsilon)^{l_i^{(t-1)}}$$

$$x_i^{(t)} = \frac{w_i^{(t)}}{\sum_{j=1}^{n} w_j^{(t)}}$$

If $\epsilon \in (0, 1/2]$, and $l_i^{(t)} \in [0,1]$, we get the following bound on total regret:

$$R_T \le \epsilon T + \frac{\ln(n)}{\epsilon}$$

Let's play around with some of these questions. For this problem, we will be running the randomized multiplicative weights algorithm with two experts. Consider every subpart of this problem distinct from the others.

(a) Let's say we believe the best expert will have cost 20, we run the algorithm for 100 days, and epsilon is $\frac{1}{2}$. What is the maximum value that the total loss incurred by the algorithm can be?

(b) What value of ϵ should we choose to minimize the total regret, given that we run the algorithm for 25 days?

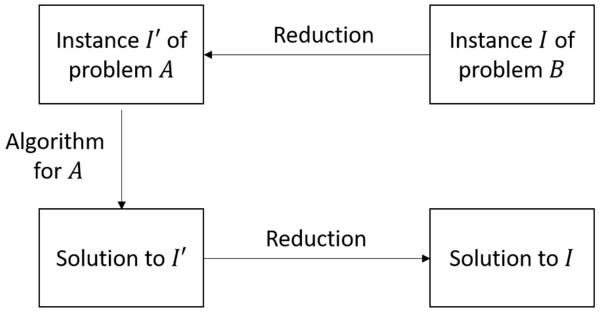
(c) We run the randomized multiplicative weights algorithm with two experts. In all of the first 140 days, Expert 1 has cost 0 and Expert 2 has cost 1. If we chose $\epsilon = 0.01$, on the 141st day with what probability will we play Expert 1? (Hint: You can assume that $0.99^{70} \approx \frac{1}{2}$)

Reduction: Suppose we have an algorithm to solve problem A, how can we use it to solve problem B?

This has been and will continue to be a recurring theme of the class. Examples so far include

- Use LP to solve max flow.
- Use max flow to solve min s-t cut.
- Use minimum spanning tree to solve maximum spanning tree.
- Use Huffman tree to solve twenty questions.

In each case, we would transform the instance I of problem B we want to solve into an instance I' of problem A that we can solve, and also describe how to take a solution for I' and transform it into a solution for I:



Importantly, the transformation should be efficient, i.e. takes polynomial time. If we can do this, we say that we have reduced problem B to problem A.

Conceptually, a efficient reduction means that if we can solve problem A efficiently, we can also solve problem B efficiently. On the other hand, if we think that B cannot be solved efficiently, we also think that A cannot be solved efficiently. Put simply, we think that A is "at least as hard" as B to solve.

To show that the reduction works, you need to prove **both** (1) if there is a solution for instance I' of problem A, there must be a solution to the instance I of problem B and (2) if there is a solution to instance I of B, there must be a solution to instance I' of problem A.

2 Bad Reductions

In each part we make a wrong claim about some reduction. Explain for each one why the claim is wrong.

(a) The shortest simple path problem with non-negative edge weights can be reduced to the longest simple path problem by just negating the weights of all edges. There is an efficient algorithm for the shortest simple path problem with non-negative edge weights, so there is also an efficient algorithm for the longest path problem.

(b) We have a reduction from problem B to problem A that takes an instance of B of size n, and creates a corresponding instance of A of size n^2. There is an algorithm that solves A in quadratic time. So our reduction also gives an algorithm that solves B in quadratic time.

(c) We have a reduction from problem B to problem A that takes an instance of B of size n, and creates a corresponding instance of A of size n in $O(n^2)$ time. There is an algorithm that solves A in linear time. So our reduction also gives an algorithm that solves B in linear time.

(d) Minimum vertex cover can be reduced to shortest path in the following way: Given a graph G, if the minimum vertex cover in G has size k, we can create a new graph G' where the shortest path from s to t in G' has length k. The shortest path length in G' and size of the minimum vertex cover in G are the same, so if we have an efficient algorithm for shortest path, we also have one for vertex cover.

3 (3,3)-SAT

Consider the (3,3)-SAT problem, which is the same as 3-SAT except each literal or its negation appears *at most* 3 times across the entire formula. Notice that (3,3)-SAT is reducible to 3-SAT because every formula that satisfies the (3,3)-SAT constraints satisfies those for 3-SAT. We are interested in the other direction.

Show that (3,3)-SAT is NP-Complete via reduction from 3-SAT. By doing so, we will have eliminated the notion that the "hardness" of 3-SAT was in the repetition of variables across the formula. (Hint: If variable x_i appears in k clauses, your reduction should replace x_i with k variables $x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(k)}$. How can we enforce that the copies of x_i all have the same value?).

Give a precise description of the reduction and prove its correctness.

4 Graph Coloring Problem

An undirected graph $G = (V, E)$ is k-colorable if we can assign every vertex a color from the set $1, \cdots, k$, such that no two adjacent vertices have the same color. In the k-coloring problem, we are given a graph G and want to output "Yes" if it is k-colorable and "No" otherwise.

(a) Show how to reduce the 2-coloring problem to the 3-coloring problem. That is, describe an algorithm that takes a graph G and outputs a graph G', such that G' is 3-colorable if and only if G is 2-colorable. To prove the correctness of your algorithm, describe how to construct a 3-coloring of G' from a 2-coloring of G and vice-versa. (No runtime analysis needed).

(b) The 2-coloring problem has a $O(|V| + |E|)$-time algorithm. Does the above reduction imply an efficient algorithm for the 3-coloring problem? If yes, what is the runtime of the resulting algorithm? If no, justify your answer.