CS 181b
Spring 2012
Homework 5
Due Wednesday, February 22

This is the last assignment for this course! You have all you need to do Problems 1 through 3. Problem 4 is based on material from next Monday's lecture. The final exam will be sent out to the class list on Friday, February 24 and will be due back at Ran's office on Friday, March 2 at 5 PM.

1. **[25 Points] Roving Robots.** Consider a collection of k identical robots in Euclidean 2-space, initially located at k distinct points, $p_1, \ldots, p_k$. Periodically, a "request" arises to dispatch a robot to a given point in the plane. If there is a robot already at that point, nothing needs to happen. If there is no robot at the point, we must dispatch one of our robots to service that request (and all other robots must remain where they are). The objective is to minimize the total Euclidean distance travelled by all robots for a given request sequence $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ where each σ_i is a request point in the plane.[1]

   We'd like to show that no deterministic online algorithm can achieve a competitive ratio better than k times optimal. More precisely, you'll show that for sufficiently long request sequences, the ratio between the total distance travelled by the robots using any online ALG and the total distance traveled by an optimal solution will approach k (or more). You'll do this using the "average of stupids" method.

   (a) Begin by considering any given online algorithm ALG. Consider the k distinct starting points for the robots, $p_1, \ldots, p_k$, and add one new distinct point to get a set of $k+1$ points $S = \{p_1, \ldots, p_{k+1}\}$. Using this very restricted set of possible points will help keep our later analysis relatively clean. Let $d(p_i, p_j)$ denote the distance between points p_i and p_j. Now, describe how to construct a request sequence σ of length n, such that all of the points in that sequence are drawn from the set S, and so that ALG pays "a lot." The first request in your sequence should, of course, be for p_{k+1}. Give an expression for the amount paid by ALG to service this request sequence.

[1] Some of you may recall, from CS 140, a k-competitive online algorithm when the robots are on a line or a tree. Whether you've seen this or not doesn't matter here. This problem does not involve anything from that algorithm or its analysis.

(b) We'd like to know how the optimal offline algorithm would do on this request sequence. Since it's hard to find the cost of the optimal algorithm, we'll define k "stupid" algorithms and use the average of their costs as a bound on OPT. The i^{th} stupid algorithm, $1 \leq i \leq k$, will begin by moving its robot at p_i to point p_{k+1} before servicing any requests. So, stupid algorithm 1 starts with a robot at each point other than p_1 and stupid algorithm k has a robot at each point other than p_k. From now on, the i^{th} stupid algorithm services the request at point $p_j \in S$ as follows: If there is a robot at p_j no robot is moved. Otherwise, the robot that is stationed at the point of the previous request is moved to this new request point. (Notice that there must be a robot there since that previous request just got serviced. Moreover, the first request in your sequence was p_{k+1} and every stupid algorithm already has a robot there!) Now, imagine that we were to watch each of the stupid algorithms service the sequence σ. Argue carefully that after servicing each request, no two stupid algorithms have their robots in exactly the same configuration. (A configuration is the collection of locations of the k robots.)

(c) Now, use the "average of stupids" method to argue that the ratio between ALG's travel cost and OPT's travel cost approaches k (or more) for sufficiently large n.

2. **[25 Points] Taking 3SAT to the Max!** MAX 3SAT takes an instance of 3SAT (a boolean expression in 3-CNF form) and asks for the valuation which maximizes the number of satisfied clauses.

   Here's the approximation algorithm for MAX 3SAT that we saw in class: Choose a literal (a "literal" is a variable or the negation of a variable) which appears the most number of times in the given expression (breaking ties arbitrarily). Make that literal true, thereby satisfying some clauses. Remove all of these satisfied clauses. In the remaining clauses, every occurrence of the negation of the literal is crossed-off, thereby reducing the number of "live" literals in some clauses. Repeat until there are no clauses with "live" literals remaining. Notice that in this way, the number of "live" literals in each clause starts off at 3 but can eventually drop to 0, at which point the clause is "dead" and thus unsatisfied!

   (a) First, generalize this algorithm for an approximation algorithm for MAX k-SAT, the variant in which each clause contains exactly k literals. Your

approximation ratio will depend on k and should approach 1 as k approaches infinity!

(b) You may be saying to yourself: "That's very cool, but we should be able to do even better!" You're right! We can do MUCH better. Consider the following algorithm for the MAX κ-SAT problem: Give each clause of the C clauses a weight 2^{-k}. Now, repeatedly choose some variable which has not been assigned a value yet. Let x be this variable. Let C_x be the set of clauses containing x and let $C_{\overline{x}}$ be the set of clauses containing $\overline{x}$. (These sets are disjoint, since otherwise the clauses in the intersection are trivially satisfiable.) If the total weight of the clauses in C_x is at least as large as the total weight of the clauses in $C_{\overline{x}}$, then make x true. Otherwise make x false. In the former case, remove all clauses containing x and, for each clause containing $\overline{x}$, remove $\overline{x}$ from the clause and double that clause's weight. Do the analogous thing if x is made false. Prove that this algorithm is a polynomial-time approximation algorithm with approximation ratio of $\frac{2^k-1}{2^k}$. Nice! *Hint:* Keep track of the total weight of all clauses. What happens to the total weight after reach round of the algorithm? What is the weight of a clause when it "dies" ?

3. **[25 Points] Precedence Constrained Scheduling.** In class we say a $\frac{3}{2}$-approximation algorithm for the problem of task scheduling with precedence constraints on 2 processors. Describe a generalization to the algorithm that works for any positive integer m of processors greater than or equal to 2 and show that the approximation ratio of your algorithm is $2 - \frac{1}{m}$.

4. **[25 Points] Finishing Up the Bin Packing Approximation Scheme!** In class we examined a polynomial time approximation scheme for Bin Packing. We showed that if all elements have size at least ϵ then in polynomial time we can find solutions which are at most $1 + \epsilon$ times optimal.

   Now we turn to those elements of size less than ϵ. We try to place them in the existing bins using the First Fit heuristic. If they all fit in the existing bins, we're done.

   The remaining case is that there are some items of size less than ϵ and they don't all fit in existing bins when using the First Fit heuristic. In this case, the First Fit heuristic was required to start some new bins. Your job is to show that in this case, the number of bins used by the algorithm is at most $(1 + 2\epsilon)OPT + 1$, assuming $\epsilon \leq 1/2$.

   Our analysis here will be entirely from scratch!

(a) Let M be the number of bins used by this algorithm. Observe that with the exception of the last bin, all other bins are full to the extent $1 - \epsilon$ or more. Explain briefly why this is so.

(b) Observe that the sum of the items sizes is at least $(M-1)(1-\epsilon)$. Briefly explain why.

(c) Show the mathematical fact that if $\epsilon \leq \frac{1}{2}$ then

$$\frac{1}{1-\epsilon} \leq (1+2\epsilon)$$

(d) Explain why the algorithm uses a number of bins at most $(1+2\epsilon)OPT+1$.

(e) Show that the complete algorithm still has running time that qualifies it as a PTAS.