CS264: Homework #1

Due by the beginning of class on Wednesday, October 1, 2014

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group.

(2) Turn in your solutions at `http://rishig.scripts.mit.edu/cs264-bwca/submit-paper.html`. You can contact the TA at `bwca-staff@lists.stanford.edu`. Please type your solutions if possible and feel free to use the LaTeX template provided on the course home page.

(3) Students taking the course for a letter grade should complete all exercises and problems. Students taking the course pass-fail should complete 5 of the exercises.

(4) Write convincingly but not excessively. Exercise solutions rarely need to be more than 1-2 paragraphs. Problem solutions rarely need to be more than a half-page (per part), and can often be shorter.

(5) You may refer to your course notes, and to the textbooks and research papers listed on the course Web page *only*. You cannot refer to textbooks, handouts, or research papers that are not listed on the course home page. Cite any sources that you use, and make sure that all your words are your own.

(6) If you discuss solution approaches with anyone outside of your team, you must list their names on the front page of your write-up.

(7) Exercises are worth 5 points each. Problem parts are labeled with point values.

(8) No late assignments will be accepted.

Lecture 1 Exercises

Exercise 1

This exercise and the next consider instance-optimal algorithms in the comparison model. Here, $\text{cost}(A, z)$ is defined as the number of comparisons that algorithm A uses to determine the (correct) solution to the input z. An algorithm A is *instance-optimal* if

$$\text{cost}(A, z) \leq c \cdot \text{cost}(B, z) \tag{1}$$

for every (correct) algorithm B and input z, where $c \geq 1$ is a constant independent of B and z.

Prove that there is no instance-optimal algorithm for searching a sorted array for a given element.

Exercise 2

Prove that there is no instance-optimal algorithm for sorting a given array (in the comparison model).

Lecture 2 Exercises

Exercise 3

Prove that, for arbitrarily large n, there are n-point instances of the 2-D Maxima problem on which the Kirkpatrick-Seidel (KS) algorithm uses $\Omega(n \log n)$ comparisons.

Exercise 4

Give a direct proof that the KS algorithm uses $O(n \log h)$ comparisons on every input, where n is the number of input points and h is the number of output points (i.e., maxima).

[Hint: recall the intuition from lecture. You should not use Exercise 5 below in your solution.]

Exercise 5

Prove that the comparison bound from lecture

$$\min_{\text{legal } S_1,\ldots,S_k} \left\{ \sum_{i=1}^{k} \left(|S_i| \log \frac{n}{|S_i|} \right) \right\} \tag{2}$$

is always at most $O(n \log h)$, where n is the number of input points and h is the number of output points.

[Hint: consider vertical slabs.]

Exercise 6

Give an infinite family of 2-D Maxima instances in which the number of output points h tends to infinity (so $n \log h$ is super-linear) but the upper bound (2) is linear (at most cn for a constant $c > 0$ that is independent of n and h).

Problems

Problem 1

The point of this problem is to study another example of an instance-optimal algorithm, in a different application domain. Consider a set X of objects, each with m real-valued attributes between 0 and 1 (higher is better). Given is a *scoring function* $\sigma : [0,1]^m \to [0,1]$ which aggregates m attribute values into a single score; we always assume that σ is nondecreasing in each component. Example scoring functions include the average and the maximum.

The objects X can only be accessed in a restricted way. For each attribute, there is a list L_i that contains X, sorted according to ith attribute values (highest to lowest). Think of X as large and m as a small constant.[1] An algorithm can only access the data by requesting the next object in one of the lists (like popping a stack). Thus an algorithm could ask for the first (highest) object of L_4, followed by the first object of L_7, followed by the second object of L_4, and so on. Such a request reveals the name of said object along with all m of its attribute values. We charge an algorithm a cost of 1 for each such data access — $\text{cost}(A, z)$ is the number of data accesses that the algorithm A needs to correctly identify the solution given the input z.

The computational problem we consider is: given a positive integer k, identify k objects of X that have the highest scores according to σ (ties can be broken arbitrarily).[2]

(a) (4 points) Consider the "straightforward algorithm" shown in Figure 1. Prove that, for every scoring function, this algorithm is correct.

[Recall that scoring functions are always assumed to be monotone.]

(b) (4 points) Prove that there is a scoring function σ and choices of m and k such that, for some inputs z, the straightforward algorithm incurs cost strictly larger than m times that of other (correct) algorithms.

[Hint: How does the execution of the straightforward algorithm depend on σ? Think of a (possibly uninteresting) scoring function where much less work is needed to correctly solve the problem.]

[1] For example, X could be Web pages, and attributes the ranking (e.g., PageRank) of a Web page under m different search engines.

[2] To develop intuition for the problem, note that an object with the highest score might only appear far from the front of every one of the sorted lists.

```
Input: a parameter $k$ and $m$ sorted lists.
  1. Repeat
      (a) Fetch the next item from each of the $m$ lists.
      (b) Let $R$ denote the set of objects that have been encountered in at least one list. Let $S \subseteq R$ denote
          the set of objects that have already been encountered in every one of the $m$ lists.
     until $S$ contains at least $k$ objects.
  2. Return the $k$ objects of $R$ that have the highest score under $\sigma$, breaking ties arbitrarily.
```

Figure 1: The straightforward algorithm.

(c) (5 points) Consider the "threshold algorithm" shown in Figure 2. Prove that, for every scoring function, this algorithm is correct.

[Hint: the intuition is that t acts as an upper bound on the best-possible score of an unseen object.]

```
Input: a parameter $k$ and $m$ sorted lists.
Invariant: of the objects seen so far, $S$ is those with the top $k$ scores.
  1. Fetch the next item from each of the $m$ lists.
  2. Compute the score $\sigma(x)$ of each object $x$ returned, and update $S$ as needed.
  3. Let $a_i$ denote the $i$th attribute value of the object just fetched from the list $L_i$, and set a threshold
     $t := \sigma(a_1, \ldots, a_m)$.
  4. If all objects of $S$ have score at least $t$, halt; otherwise return to step 1.
```

Figure 2: The threshold algorithm.

(d) (8 points) Prove that, for every scoring function, the threshold algorithm is instance-optimal, in the sense that

$$\text{cost}(A, z) \leq m \cdot \text{cost}(B, z) \tag{3}$$

for every algorithm B and input z (i.e., lists $L_1, \ldots, L_m$).

[Hint: the hard part is to lower bound the cost of every correct algorithm. Argue that if an algorithm B stops too early on an input z, then there exists an input z' consistent with B's execution-so-far for which B's answer is incorrect.]

(e) (4 points) Prove that, in general, there is no deterministic algorithm that satisfies (3) with an approximation factor smaller than m.

[Hint: take $k = 1$ and suppose there is a unique highest-scoring object.]

Problem 2

The point of this problem is to apply the "order-oblivious" version of instance optimality from the Afshani/Barbay/Chan paper to the online bin packing problem.

In the online bin packing problem, items with sizes between 0 and 1 arrive online. At any point you can open up a new bin, which has capacity 1. Every item that arrives must be placed in a bin, subject to the capacity constraints. Previous items cannot be moved or removed. The objective is to minimize the number of bins opened by the end of the item sequence. A *greedy online algorithm* is one that only opens a new bin as a last resort (when the current item doesn't fit in any existing bins).

(a) (5 points) The *first fit (FF) algorithm* is the greedy online algorithm that packs the current item into the earliest bin in which it fits, or opens a new bin for the item if there is currently no room for it. Show that this algorithm is order-oblivious instance-optimal among greedy online algorithms, in the sense that for every (unordered) set S of items and every greedy online algorithm B:

$$\max_{\pi}\{\text{cost}(FF, \pi(S))\} \le \max_{\pi}\{\text{cost}(B, \pi(S))\},$$

where $\text{cost}(A, z)$ denotes the number of bins used by algorithm A on input z, $\pi(S)$ denotes an ordering of the items in S, and each of the max operations ranges over all such orderings.

(b) (5 points) Prove that not every greedy online algorithm is order-oblivious instance-optimal in the sense of (a).

(c) (10 points) Identify the minimum value of $\alpha \ge 1$ for which the following statement is true: for every two greedy online algorithms A and B and every set S of items,

$$\max_{\pi}\{\text{cost}(A, \pi(S))\} \le \alpha \cdot \max_{\pi}\{\text{cost}(B, \pi(S))\}.$$

Extra Credit

Extra Credit Problem 1

(10 points) In our definition (1) of instance optimality, we insisted that the approximation factor c be independent of the algorithm B (and of course the input z). This problem explains the importance of this choice for settings in which cost denotes the running time of an algorithm: allowing the approximation to depend on B permits highly impractical instance-optimal algorithms.

Consider an arbitrary search problem — given an instance, the responsibility of the algorithm is to exhibit a correct solution or report that none exist — in which every correct algorithm takes at least linear time (e.g., from reading the input). Suppose further that the correctness of a purported solution can be verified in linear time. Show that for every such problem there is an instance-optimal algorithm A in the sense that

$$\text{cost}(A, z) \le f(|A'|) \cdot \text{cost}(A', z)$$

for every input z and (correct) algorithm A', where $|A'|$ denotes the size (e.g., number of lines of code) of the algorithm A' and $f(|A'|)$ is some function that depends only on $|A'|$ and not on the length of z.

[Hint: consider enumerating and deftly simulating all programs of at most a given length.]