Problem Set 9

Due: Wednesday, November 9, 2005.

Problem 1. In class we used a "rescaling" argument to show that if one had an absolute approximation algorithm for maximum independent set (MIS), one could transform it into a better absolute approximation algorithm for the problem (and in fact, solve the problem exactly). In this problem, we will show that a similar result holds true for relative approximation: that any constant-factor relative approximation algorithm can be improved to a better constant factor relative approximation.

Suppose one has an α-(relative) approximation algorithm for MIS. Consider the following "graph product" operation for a graph G. Create a distinct copy G_v of G for each vertex v of G. Then connect up the copies as follows: if (u, v) is an edge of G, then connect every vertex in G_u to every vertex in G_v.

(a) Prove that if there is an independent set of size k in G, then there is an independent set of size k^2 in the product graph.

(b) Prove that given an independent set of size s in the product graph, one can find an independent set of size $\sqrt{s}$ in G.

(c) Prove that if there is an α-approximation for MIS for *some* fixed α, then there is a polynomial approximation scheme for MIS.

Since MIS was shown to be MAX-SNP-hard, meaning there is some constant to within which it *cannot* be approximated (unless $P = NP$) this proves that MIS has *no* constant-factor relative approximation.

NONCOLLABORATIVE Problem 2. The *Steiner Tree Problem* presents an undirected graph G with edge costs c_e, and a subset T of the vertices called *terminals*. The goal is to construct a minimum cost tree spanning all the terminals (it may also include any desired subset of the non-terminals; these included vertices are called *Steiner points*).

(a) Suppose you compute all-pairs shortest paths in G, and create a complete graph G' on the terminals T where each edge cost is equal to the shortest path between its (terminal) endpoints in G. Relate the cost of the minimum spanning tree in this graph to the cost of the optimum steiner tree in G.

(b) Give a 2-approximation algorithm for the Steiner tree problem.

Problem 3. The following is the NP-hard problem of **bin packing**: Given n items with sizes a_1,...,$a_n \in (0,1]$, find a packing of the items into unit-sized bins that minimizes the number of bins used. Let B^* denote the optimum number of bins for the given instance. Bin packing is a lot like $P||C_{\max}$, but somewhat more difficulty because you have no flexibility to increase the bin sizes.

(a) Suppose that there are only k distinct item sizes for some constant k. Argue that you can solve bin-packing in polynomial time. **Hint:** no new algorithm needed here!

(b) Suppose that you have packed all items of size greater than ϵ into B bins. Argue that in linear time you can add the remaining small items to achieve a packing using at most $\max B, 1 + (1+\epsilon)B^*$ bins.

(c) For $P||C_{\max}$, we reduced to the previous case by rounding each job size up to the next power of $(1+\epsilon)$. Why doesn't that work for bin packing?

(d) Consider instead the following *grouping* procedure. Fix some constant k. Order the items by size. Let S_1 denote the largest n/k items, S_2 the next largest n/k, and so on. Suppose that you increase the size of each item to equal the largest size in its group, so that there are only k distinct sizes. Argue that this increases the optimal number of bins by at most n/k. **Hint:** imagine setting aside the jobs in S_1. Argue that the remaining items, with their increased sizes, can still fit into the bins used by the original packing.

(e) By applying the grouping procedure to items of size greater than $\epsilon/2$, solving the result optimally, and then adding the small items, devise a polynomial time scheme that uses at most $(1+\epsilon)B^* + 1$ bins.

Observe that the algorithm above just misses the definition of polynomial approximation scheme, because of the additive error of 1 bin. In practice, of course, this is unlikely to matter. The above scheme is known as an *asymptotic PAS* since its approximation ratio is $(1+\epsilon)$ in the limit as the optimum value grows.

The techniques above have been augmented to give an algorithm that finds a packing using $B^* + O(\log^2 B^*)$ bins—giving asymptotic approximation ratio 1. Indeed, at present it remains concievable that we might achieve $B^* + O(1)$ bins in polynomial time!

Problem 4. You are given a collection of jobs, each with a *processing time* p_j. There are also *precedence constraints*: job j cannot be started until after all jobs in its *precedence set* $A(j)$ have been completed. Each job gets a *weight* w_j. Our goal is to minimize the *weighted average completion time* $\sum w_j C_j$ (where C_j is the time job j completes). In other words, we are looking at the problem $1 \mid prec \mid \sum w_j C_j$. Assuming that the p_j are polynomially-bounded integers, we will give a constant-factor approximation for this problem via a linear programming relaxation. Define variables x_{jt} for each (integer) time step t, denoting the "indicator" that job j completed at time t.

(a) Write down constraints forcing the ILP to solve the problem. In particular, enforce that only every job completes, that a job is not processed before ites predecesors, and (most subtly) that the total processing time of jobs completed before time t is at most t.

(b) The LP relaxation of this ILP is a kind of "timesharing" schedule for jobs. Define the *fractional commpletion time* of job j to be $\overline{C}_j = \sum_t t x_{jt}$. To turn it into an actual order, consider the *halfway point* h_j of each job: this is the time at which half the job is completed. Prove that $\overline{C}_j \geq h_j/2$.

(c) Consider the schedule that processes jobs in order of their halfway points. Prove that no job runs before its predecessors.

(d) Prove that for the given order, the actual completion time for job j is at most $2\overline{C}_j$.

(e) Conclude that you have a constant-factor approximation for $1 \mid prec \mid \sum C_j$.

OPTIONAL (f) Suppose that each job comes with a *release date* r_j before which it cannot be processed. Generalize your algorithm to handle this case (with a slightly worse constant).