1 Approximation Schemes

Last week we studied linear programming as a tool for finding good approximation algorithms. Namely, we formulated our NP-hard problems as integer programs, relaxed them to linear programs, and used the optimal solution to the linear program as a lower bound on the optimal solution to the integer program. For VERTEX COVER we saw how to deterministically round a fractional solution to yield a 2-approximation for the general problem. After this we saw how to use *randomized rounding* to give a $O(\log n)$-approximation for SET COVER. Both of these approximation factors are close to, or essentially tight, modulo $P = NP$. That is, finding a better approximation is NP-hard.

This week we will study problems which afford *arbitrarily close* approximations. What does this mean? Given some minimization problem instance $\mathcal{I}$ and some small value ϵ, we are interested in finding a solution to $\mathcal{I}$ that is within $(1 + \epsilon)$ of $OPT(\mathcal{I})$. We call procedures of this sort *approximation scheme*. Of course, the quality of the solution often trades with the running time of our algorithm. Given a problem of size n and an $\epsilon > 0$, we say that an approximation scheme is *fully polynomial-time* when it returns a solution that is within a factor of $(1 + \epsilon)$ of optimal in time polynomial in n and 1ϵ. We say that an approximation scheme is simply *polynomial-time* when approximation factor remains at $(1 + \epsilon)$ but the running time is only polynomial in n; in other words, it may be exponential (or worse) in 1ϵ.

Reading 1. *Begin by reading Mark de Berg's lecture notes on approximation schemes. These notes are available on the website.*

Question 1. *Why do problems like* VERTEX COVER*,* SET COVER*, and* GRAPH COLORING *not have* fully *polynomial time approximation schemes?*

Question 2. *What is the salient characteristic of problems like* KNAPSACK *and* SUBSET SUM *that allows them to have* fully *polynomial time approximation schemes?*

Problem 1. *Implement the fully polynomial time approximation scheme for the* KNAPSACK *problem in a language of your choice. Your program should accept input from* `stdin` *having the following form:*

```
epsilon
W
n
w1 p1
w2 p2
.
.
.
wn pn
```

where `epsilon` *gives the approximation quality (i.e. a real number),* n *denotes the number of items in the initial set,* W *denotes the total weight of the sack,* w_i *denotes the weight of item* i*, and* p_i *denotes the profit of item* i*. Your output should be a set of items, the total weight of the items, and the total profit. Perform the following experiment to empirically verify the correctness of your implementation:*

1. *Generate a set of 100 random integers in the range* $[0, 2^{16} - 1]$*. Select 50 of these numbers uniformly at random. Let* W *be their total weight and* P *be their total profit. Let the 100 numbers plus* W *be your instance. Note that there may be a completely different set of elements with weight at most* W *and profit larger than* P*. However, in this case,* P *serves as a lower bound on the optimal solution.*

2. *Run your FTPAS on your instance with varying values of* ϵ*. Record the cost of the solution.*

3. *Since you have a lower bound on OPT which in this case is P, you can determine the approximation ratio of your solution with P. Plot this ratio. Also plot $(1-\epsilon)$ which should be a lower bound on your approximation ratio. Your plots should use the x-axis for the varying (say, decreasing) values of ϵ and the y-axis for the approximation ratio.*

4. *If you have time, try running the exact dynamic programming algorithm on your instances to determine the true optimum. Your FPTAS solution should still be within a factor of $(1-\epsilon)$ of this value too.*

Problem 2. *Consider the non-negative sets of integers $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, and a non-negative integer C. For a subset $K \subseteq \{1, \ldots, n\}$, denote $A_K = \sum_{j \in K} a_j$ and $B_K = \sum_{j \in K} b_j$. The goal is to find an index set K with $A_K + B_K \leq C$ such that $A_K^2 + B_K^2$ is maximized. Design an FPTAS for this problem.*

Reading 2. *Read* Huffman Coding with Unequal Letter Costs *by Golin, Kenyon (Mathieu), and Young. The paper appeared at STOC 2002 and is available on the course website. The Huffman coding problem is a classic: Given an alphabet with n characters $A = \{a_1, \ldots, a_n\}$ and associated frequencies $w_1, \ldots, w_n$, develop a binary prefix-free code for A that has minimum expected codeword length. The general version assumes that the code can be k-ary instead of binary and that each encoding symbol α_j (often called a* letter*) has a length l_j. The complexity of this problem remains open!*

Question 3. *Write a short review of the Golin et al. paper. What algorithmic ideas did they employ that were similar to the FTPAS for the* KNAPSACK *problem? What algorithmic techniques were different?*