Approximating SET-COVER

Today we look at two more examples of approximation algorithms for **NP**-hard optimization problems. The first, for the SET-COVER problem, has an approximation factor of $\Theta(\log n)$. The second, for the KNAPSACK problem, is our first example of a **fully polynomial approximation scheme** or **FPTAS**.

In the SET-COVER problem our input is a set U and a collection of subsets $S_1, \ldots, S_m$, where each S_i has a non-negative weight w_i. A solution is a subcollection of the set whose union is the entire set U. The score of a solution is the total weight of the sets used, and our goal is to minimize this score.

SET-COVER is a very general problem. For example, given an undirected graph G we can let U be the set of edges E and let S_v for every vertex v be the set of edges incident on v. If we give each S_v a weight of 1, the minimum weight set cover is just the minimum size vertex cover of G. Thus a polynomial-time solution to the SET-COVER optimization problem could be used to get a polynomial-time solution to the VERTEX-COVER decision problem, so the former problem is **NP**-hard.

Our approximation algorithm for SET-COVER provides an example of **amortized cost**. Suppose we have a collection of sets already chosen and we are considering choosing a new set S_i. The cost of choosing S_i is w_i, and the benefit is that some nonzero number k of new elements will be covered – elements that were not covered by the previous collection. We say that the amortized cost of covering each of these elements is w_i/k. Note that this cost depends on the context – on which sets have already been chosen.

Here is a greedy algorithm to approximate SET-COVER. Begin with the empty collection of sets and add sets one by one until every element has been covered. At each stage, choose the set S_i that minimizes the quantity w_i/k, where S_i covers k new elements. Thus we cover one or more new elements each time, using the set that covers a new element most cheaply in the amortized cost measure.

We can argue that this is a reasonably good approximation. In fact, given any set S_i at all, this greedy algorithm covers the elements of S_i with a cost that is $O(\log n)$ times greater than the cost of S_i itself. Therefore, whatever sets the optimal covering uses, the greedy algorithm covers all the elements using a cost that is only $O(\log n)$ times the cost of those sets, hence $O(\log n)$ times the cost of the optimal covering.

To verify this claim about an arbitrary set, let the set S_i be $\{a_1, \ldots, a_d\}$, where the elements are numbered in the order that they are covered by the greedy algorithm. (If more than one of these elements are covered at the same time we can choose an arbitrary order among them.)

Look first at element a_d. When it was covered, the available sets to cover it must have included S_i itself. Covering a_d with S_i would have incurred an amortized cost of w_i, if a_d was the only element covered, or less than that if there were others covered at the same time. The greedy algorithm would have taken this option unless it had another at least as good, so it incurred an amortized cost of at most w_i.

Now look at element a_{d-1}. When it was covered, it would have been possible to cover it, *and* a_d, with the set S_i. This would have incurred an amortized cost of at most $w_i/2$ (less if other elements were covered at the same time). Thus the amortized cost incurred by the greedy algorithm must have been at most $w_i/2$, as it either used S_i or chose another option with at most the same cost.

Similarly element a_{d-2} was covered by the greedy algorithm with an amortized cost of at most $w_i/3$. Each element a_j incurs an amortized cost of at most $w_i/(d-j+1)$, all the way through a_1 which has a cost of w_i/d. The total cost of covering all the elements of S_i is thus at most:

$$w_i(\frac{1}{d} + \frac{1}{d-1} + \ldots + \frac{1}{2} + 1).$$

Remember that this sum $1+\frac{1}{2}+\ldots+\frac{1}{d}$ is called $H(d)$ and is between $\ln d$ and $1+\ln d$. (This is easy to see by looking at the graph of $y = 1/x$ and its integral from 1 to d which is $\ln d$.) If we let d^* be the maximum size of any set in the collection, it follows that the approximation ratio of this algorithm is at most $1+\ln d^*$. Since d^* can be at most n, the total number of elements in U, the approximation ratio is at most $1 + \ln n$ and thus $O(\log n)$.

Can we put a better upper bound on the approximation ratio of this algorithm? No, and in fact we have already seen the bad example of its behavior. Recall that before we presented the 2-approximation algorithm for VERTEX-COVER, we looked at a greedy approximation algorithm that always chose the vertex of highest degree (among the edges not yet covered).

When we translate this greedy algorithm to the equivalent SET-COVER problem, we find that it chooses the set S_v that maximizes k, the number of new edges covered, and thus minimizes the amortized cost $1/k$ of covering another edge. It is the same greedy algorithm. We found in our example that the approximation ratio was $\Omega(\log n)$ (in fact about $H(n)$ itself).

We have not proved that some other approximation algorithm might not do better than this one, but in fact it is **NP**-hard to approximate SET-COVER to within $c \log n$ for some positive constant c. We won't prove this here – it's hard.

Approximating KNAPSACK

Now we look at an **NP**-hard approximation that has a **fully polynomial approximation scheme**. This means that for any positive number ϵ, even one that depends on n, there is a polynomial-time algorithm that approximates the optimal solution within a factor of $1 + \epsilon$. Furthermore, the running time of the algorithm is polynomial both in n and in $1/\epsilon$. Thus, for example, we could have $\epsilon = 1/n^3$ and still have a polynomial-time algorithm in n.

Our problem in this case is KNAPSACK, an optimization problem that generalizes the **NP**-complete decision problem SUBSET-SUM. We have n **items**, each with a **weight** and a **value** (both non-negative). A solution is a set of items whose total weight meets a given **weight target**, the score of a solution is its total value, and our goal is to maximize the score.

Remember that if the weight and value of each item are the same, this becomes the SUBSET-SUM problem. We proved that it is **NP**-complete to decide whether we can meet a particular weight target, and thus **NP**-hard to find the maximum possible weight achievable within the target.

But also recall that this optimization problem is **pseudopolynomial**, meaning that if the weights of each object are given in **unary**, we can solve the problem exactly in a time that is polynomial in the new, larger input size. The KNAPSACK problem is similarly pseudopolynomial. Even if just the weights are integers given in unary, and the values are real numbers, we can use dynamic programming to find the largest value obtainable for each possible weight. If there are only polynomially many possible weights, this is a poly-time algorithm.

This suggests an approximation algorithm. Just *approximate* the weights by rounding them to the first k significant digits for some k. Then there are only 2^k possible values for each weight, and the total weight (and the weight target) can be represented by an integer whose value is at most $n2^k$. We then use dynamic programming to solve the problem exactly for the revised weights. We might not get exactly the best possible set of items, but we ought to come pretty close, right?

Sadly, no. We have two choices – round our weights up, or round them down. If we round them down, we risk finding a set of items whose *rounded* weights meet the rounded weight target, but whose actual weights don't meet the actual weight target. This isn't a good approximation because it isn't a valid solution to the original problem at all.

No problem, we'll just round the weights up. Now we're bound to have our approximate solution be a valid solution to the original problem. But what if there's a very low-weight item of very high value? We could have a solution to the rounded-weight problem that omits this item when it could actually fit along with this solution in the original problem. The optimal solution could of course include it, and its value could be a significant fraction of the score and prevent us from getting a $1 + \epsilon$ approximation.

The way out of this is interesting – we approximately solve a *different optimization problem* called the **dual** of KNAPSACK. Here the input sets are the same, but we have a **value target** instead of a weight target and our goal is to meet the value target with the smallest possible weight.

Remember that in the dynamic programming solution to the original problem we only needed the weights to be small integers, not the values. In the dual problem it's the other way round – we can have real-valued weights as long as the values are small integers, because our dynamic programming table has an entry for every possible value instead of for every possible weight.

Now we can approximate the values while keeping the weights exact. For each rounded value target, we find a set of items whose *actual weight* is as small as we can find, and which meets the target. By dynamic programming, we find such a set if it exists. The actual value of the set we find may be slightly different from the rounded value, and thus we might miss a different set that has the same or worse rounded value, but a better real value.

But the margin by which the optimal set can outscore the set we find is only the rounding error in approximating the values. **The rest of this slide is revised slightly based on the discussion in lecture.** If we keep the first k bits of the value (rounding up or rounding down, as we like) then each of our estimated item values may be in error by at most $v_{max}/2^k$, where v_{max} is the maximum value. The estimated value of a set may be off by as much as $nv_{max}/2^k$. Since this is the difference between the estimated and actual value of the optimal set, and we achieve at least the same estimated value as does the optimal set, this is the maximum error of the algorithm.

We thus want to set k so that the maximum possible error $nv_{max}/2^k$ is at most an ϵ fraction of the optimum value. We may assume that the optimum is at least v_{max}, because if the most valuable item violates the weight target by itself we might as well delete it from the problem. We thus want k to be $\log(n/\epsilon)$, and our table is then size $2^k = n/\epsilon$. Our running time is then $O(n^2/\epsilon)$, a polynomial in both n and $1/\epsilon$ as desired. We have constructed an FPTAS for the KNAPSACK problem.

On-Line Algorithms

In the time remaining in Lecture 21 I spoke briefly without slides about on-line algorithms. These slides were added to the notes after the lecture.

Many algorithmic problems in the real world are **on-line problems**, in that the algorithm must respond to each input item as it arrives, without knowledge of future items. Usually there is a worst-case input sequence that causes any given on-line algorithm to perform badly. Often an **off-line** algorithm for the same problem, that sees the entire input before having to respond, can do much better.

On HW#5 we'll look at the problem of taking a sequence of items of various sizes and packing them into bins. The decision problem BIN-PACKING has as input the set of items with their sizes, the set of bin sizes, and a target for the maximum number of bins to be used. Often all the bins are the same size. The corresponding optimization problem is to minimize the number of bins. The optimization problem DUAL-BIN-PACKING has as input the items with their sizes and a number of bins, and the goal is to minimize the bin size needed to pack the items in that many bins. This is isomorphic to a scheduling problem where you have tasks of various sizes, each of which require one worker, and a number of workers, and you want to minimize the total time needed to do all the jobs.

In the homework we'll look at the two-bin case of DUAL-BIN-PACKING and some on-line algorithms for it where each item must be placed in a bin as soon as it arrives. Algorithm A_0 places each item in the bin that is less full at the time it is considered. Algorithm A_k gets the k largest items first and can divide them optimally between the bins – then it gets the rest of the items in arbitrary order and places each in the bin that is less full at the time.

Sometimes we can find an on-line algorithm that is **competitive** with the offline algorithm in the following sense. To be c-competitive for some constant $c > 1$, the cost of the online algorithm on any input sequence must be at most c times the cost of the optimal offline algorithm on the same sequence.

We illustrate this with the NEW-SKIER optimization problem, due to Borodin. It costs \$300 to buy skis, and \$30 to rent skis if you don't own them. The sequence of requests consists of zero or more ski outings, followed by a decision to quit skiing. The objective is to minimize the cost of skis for the requests.

The optimal offline strategy is to rent skis if there are fewer than ten outings before you quit, buy skis immediately if there are going to be more than ten, or do either if the number is exactly ten.

Here is a 2-competiive online algorithm: Rent skis the first ten times, then buy if you go out an eleventh time. If the number of outings is ten or fewer, this algorithm performs the same as the optimal one. The worst case is if you quit after the eleventh time, having spent \$600 (ten rentals and the purchase). The optimal algorithm would only spend \$300 to meet these requests, since it would be smart enough to buy immediately.

Such **competitive analysis** is an important branch of the theory of algorithms.