Advanced Algorithms Course. Lecture Notes. Part 3

An Approximation Scheme for Knapsack

So far we have seen some approximation algorithms whose approximation ratio on an instance is fixed, either an absolute constant or depending on the input size. But often we may be willing to spend more computation time to get a better solution, i.e., closer to the optimum. In other words, we may trade time for quality. A **polynomial-time approximation scheme (PTAS)** is an algorithm where the user can freely decide on some accuracy parameter ϵ and gets a solution within a factor $1+\epsilon$ or $1-\epsilon$ of optimum, and within a time bound that is polynomial for every fixed ϵ but grows as ϵ decreases. The actual choice of ϵ may then depend on the demands and resources. A nice example is the following Knapsack algorithm.

In the Knapsack problem, a knapsack of capacity W is given, as well as n items with weights w_i and values v_i (all integer). The problem is to find a subset S of items with $\sum_{i\in S} w_i \le W$ (so that S fits in the knapsack) and maximum value $\sum_{i\in S} v_i$. Define $v* := \max v_i$.

You may already know that Knapsack is NP-complete but can be solved by some dynamic programming algorithm. Its time bound $O(nW)$ is polynomial in the numerical value W, but not in the input size n, therefore we call it pseudopolynomial. (A truly polynomial algorithm for an NP-complete problem cannot exist, unless P=NP.) However, for our approximation scheme we need another dynamic programming algorithm that differs from the most natural one, because we need a time bound in terms of values rather than weights. (This point will become more apparent later on.) Here it comes:

Define $OPT(i, V)$ to be the minimum (necessary) capacity of a knapsack that contains a subset of the first i items, of total value at least V. We can compute $OPT(i, V)$ using the OPT values for smaller arguments, as follows. If $V > \sum_{j=1}^{i-1} v_j$ then, obviously, we *must* add item i to reach V. Thus we have $OPT(i, V) = w_i + OPT(i-1, V-v_i)$ in this case. If $V \leq \sum_{j=1}^{i-1} v_j$ then item i may be added or not, leading to

$$OPT(i, V) = \min(OPT(i-1, V), w_i + OPT(i-1, \max(V - v_i, 0))).$$

Since $i \leq n$ and $V \leq nv^*$, the time is bounded by $O(n^2 v^*)$. As usual in dynamic programming, backtracing can reconstruct an actual solution from the OPT values.

Now the idea of the approximation scheme is: If v^* is small, we can afford an optimal solution, as the time bound is small. If v^* is large, we round the values to multiples of some number and solve the given instance only approximately. The point is that we can divide all the rounded values by the common factor without changing the feasible solutions. In the following we work out this idea precisely. We do not specify what "small" and large" exactly means in the above sketch. Instead, some free parameter $b > 1$ controls the problem size.

First compute new values v_i' as follows: Divide v_i by some fixed b and round up to the next integer: $v_i' = \lceil v_i/b \rceil$. Then run the dynamic programming algorithm for the new values v_i' rather than v_i.

Let us compare the solution S found by this algorithm, and the optimal solution S^*. Since we have not changed the weights of elements, S^* still fits in the knapsack despite the new values. Since S is optimal for the new values, clearly $\sum_{i \in S} v_i' \geq \sum_{i \in S^*} v_i'$. Now one can easily see: $\sum_{i \in S^*} v_i/b \leq \sum_{i \in S^*} v_i' \leq \sum_{i \in S} v_i' \leq \sum_{i \in S}(v_i/b + 1) \leq n + \sum_{i \in S} v_i/b$. This shows $\sum_{i \in S^*} v_i \leq nb + \sum_{i \in S} v_i$, in words, the optimal total value is larger than the achieved value by at most an additional amount nb.

Depending on the maximum value v^* we choose a suitable b. By chosing $b := \epsilon v^*/n$, the above inequality becomes $\sum_{i \in S^*} v_i \leq \epsilon v^* + \sum_{i \in S} v_i$. Since trivially $\sum_{i \in S^*} v_i \geq v^*$, this becomes $\sum_{i \in S^*} v_i \leq \epsilon \sum_{i \in S^*} v_i + \sum_{i \in S} v_i$, hence $(1-\epsilon) \sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i$. In words: We achieve at least a $1-\epsilon$ fraction of the optimal value. The time is $O(n^2 v^*/b) = O(n^3/\epsilon)$. Thus we can compute a solution with at least $1-\epsilon$ times the optimum value in $O(n^3/\epsilon)$ time.

For any fixed accuracy ϵ this time bound is polynomial in n (not only pseudopolynomial as the exact dynamic programming algorithm). However, the smaller ϵ we want, the more time we have to invest.

The presented approximation scheme is even an FPTAS, which is stronger than a PTAS. Here is the definition: A **fully polynomial-time approximation scheme (FPTAS)** is an algorithm that takes an additional input parameter ϵ and computes a solution that has at least $1-\epsilon$ times the optimum value (for a maximization problem), or at most $1+\epsilon$ times the optimum value (for a minimization problem), and runs in a time that is polynomial in n and $1/\epsilon$.

Approximation Algorithms Using Linear Programming

A **linear program (LP)** is the following task: Given a matrix A and vectors b, c, compute a vector $x \geq 0$ with $Ax \geq b$ that minimizes the inner product $c^T x$. This is succinctly written as:

$\min c^T x$ s.t. $x \geq 0$, $Ax \geq b$.

The entries of all matrices and vectors are real numbers. The $\geq$ relation between vectors means the componentwise $\geq$ relation, and 0 denotes the zero vector.

LPs can be solved efficiently (theoretically in polynomial time). However, algorithms for solving LPs are not a subject of this course. LP solvers are implemented in several software packages. Here we use them only as a "black box" to solve hard problems approximately.

A simple example of this technique is again Weighted Vertex Cover in a graph $G = (V, E)$. The problem can be reformulated as $\min \sum_{i \in V} w_i x_i$ s.t. $x_i + x_j \geq 1$ for all edges (i, j). This is almost an LP, but the catch is that the x_i must be 1 or 0 (for node i is in the vertex cover or not), whereas the variables in an LP are real numbers. Hence we cannot use an LP solver directly. (Weighted Vertex Cover is NP-complete after all ...)

Instead we solve a so-called LP relaxation of the given problem and then construct a solution of the actual problem "close to" the LP solution. If this works well, we should get a good approximation. In our case, a possible LP relaxation is to allow real numbers $x_i \in [0, 1]$ for the moment. Let S^* be a minimum weight vertex cover, and w_{LP} the total weight of an optimal solution to the LP relaxation. Clearly $w_{LP} \leq w(S^*)$. Let x_i^* denote the value of variable x_i in the optimal solution to the LP relaxation. These numbers are in general fractional. To get rid of these fractional numbers

we do the most obvious thing: we round them! More precisely: Let S be set of nodes i with $x_i^* \geq 1/2$. Variables corresponding to nodes in S are rounded to 1, others are rounded to 0. S is obviously a vertex cover. Moreover, $w_{LP} \leq w(S^*)$ implies $w(S) \leq 2w(S^*)$, since by rounding we have at most doubled the variable values from the LP relaxation. This gives us yet another algorithm with approximation ratio 2. – We know already simpler 2-approximation algorithms for Weighted Vertex Cover, but this was only an example to demonstrate the general technique of LP relaxation and rounding.