Advanced Algorithms Course. Lecture Notes. Part 3

An Approximation Scheme for Knapsack

So far we have seen approximation algorithms whose approximation ratios on every instance were fixed (either constant or depending on the instance size). But often we may be willing to spend more computation time to get a solution closer to the optimum. In other words, we may trade time for quality. This gives rise to the following definition:

A **polynomial-time approximation scheme (PTAS)** for a maximization problem is an algorithm that, given a problem instance and an accuracy parameter ϵ, returns a solution whose objective value is at least $1-\epsilon$ times the optimum, and runs within a time bound that is polynomial for every fixed ϵ. PTAS for minimization problems are similarly defined, except that the objective value must be at most $1+\epsilon$ times the optimum.

Note that the time may grow as ϵ decreases, and the user can freely decide on some desired value of ϵ. This way, the choice of ϵ may be steered by both the quality demands and the computational resources.

By far not every problem admits a PTAS. A nice positive example is Knapsack.

In this problem, a knapsack of capacity W is given, as well as n items with weights w_i and values v_i ($i=1,\ldots,n$). All these numbers are positve integers. The problem is to find a subset S of items with $\sum_{i\in S} w_i \leq W$ (so that S fits in the knapsack) and maximum value $\sum_{i\in S} v_i$. We define $y := \max v_i$.

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. Just acecpt the statement for the moment.) 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 these values by

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

(We omit some straightforward details regarding initial values and negative arguments.) Since $i \leq n$ and $V \leq n \cdot y$, the time is bounded by $O(n^2 y)$. As usual in dynamic programming, backtracing can reconstruct an actual solution from the OPT values.

Now the rough idea of the approximation scheme is: If y is small, we can afford computing an optimal solution, as the time bound is small. If y 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, thus reduce the time, without changing the feasible solutions. In the following we work out this idea precisely. Instead of specifying "small" and large", another free parameter $b > 1$ will control the time bound smoothly.

First compute new values v'_i as follows: Divide v_i by some fixed b and round upwards 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, and output the solution S. The solution is feasible, because we have not changed the weights and capacity, but only the values v_i.

This was already the algarithm! Now we are going to analyze it.

Let us compare S to an optimal solution S^*. Since S is an optimal solution for the changed values, we have $\sum_{i \in S} v'_i \geq \sum_{i \in S^*} v'_i$. It follows: $\sum_{i \in S^*} v_i/b \leq \sum_{i \in S^*} v'_i \leq \sum_{i \in S} v'_i \leq \sum_{i \in S}(v_i/b + 1) \leq n + \sum_{i \in S} v_i/b$. 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 term nb.

Depending on the maximum value y we choose a suitable $b > 1$. We choose $b := \epsilon y/n$, provided that $\epsilon y > n$. Then the above inequality becomes $\sum_{i \in S^*} v_i \leq \epsilon y + \sum_{i \in S} v_i$. Since the best item alone is a feasible solution, we also have $\sum_{i \in S^*} v_i \geq y$. Together this implies

$$\sum_{i \in S^*} v_i \leq \epsilon \sum_{i \in S^*} v_i + \sum_{i \in S} v_i,$$

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 $O(n^2y/b) = O(n^3/\epsilon)$. Thus we can compute a solution with at least $1-\epsilon$ times the optimum value in $O(n^3/\epsilon)$ time.

We had assumed $\epsilon y > n$. In the opposite case $\epsilon y \leq n$ we do not round and approximate, but solve the problem instance exactly in $O(n^2y)$ time, which is now bounded by $O(n^3/\epsilon)$ as well.

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 a stronger type of approximation scheme: 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

Rounding can also be used together with a generic tool from Optimization, in order to obtain nice approximation algorithms. We present the idea here only briefly and succinctly.

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^Tx. This is written as: $\min c^Tx$ s.t. $x \geq 0$, $Ax \geq b$.

The entries of all these matrices and vectors are real numbers. The $\geq$ symbol between vectors means the componentwise $\geq$ relation, and 0 denotes the zero vector. Read “s.t.” as “such that” or “subject to”. Of course, matrix and vectors must have suitable sizes, such that multiplications and comparisons are well defined.

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”.

As an example we use again the Weighted Vertex Cover problem 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). The variables x_i are only capable of two values:

$x_i = 1$ if node i is in the solution, and $x_i = 0$ if not. Since the x_i are not arbitrary real numbers, this formulation is not an LP, but an **integer linear program (ILP)**. Hence we cannot use an LP solver directly. – Weighted Vertex Cover is NP-complete after all, therefore no magic can solve it in polynomial time, unless $P = NP$.

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 $0 \leq x_i \leq 1$ for the moment, and solve the resulting LP.

Let S^* be a minimum weight vertex cover, and w_{LP} the total weight of an optimal solution to our LP relaxation. Since the relaxation offers more feasible solutions to choose from, we have $w_{LP} \leq w(S^*)$.

Next, let x_i^* denote the value of variable x_i in the optimal solution to our LP relaxation. These numbers are in general fractional, so how do we obtain a vertex cover from them?

To get rid of the fractional values 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, all others are rounded to 0. The set S is clearly a vertex cover. (But note that it is crucial to round 1/2 to 1, not to 0.) Moreover, $w_{LP} \leq w(S^*)$ implies $w(S) \leq 2w(S^*)$, since by rounding we have at most doubled the value of each variable from the LP relaxation. We conclude that the approximation ratio is 2.

One might object that this section was superfluous, since we know already a simpler 2-approximation for Weighted Vertex Cover, without LP solvers. But this was only an *example* used to introduce the general technique of LP relaxation followed by rounding. It is widely usable for other problems, often with more sophisticated rounding rules.