Advanced Algorithms.
Approximation Algorithms Continued

In this part of the course we will see that hard optimization problems behave very differently when it comes to approximation. In some problems, optimal solutions are even hard to approximate, while in other problems we can get arbitrarily close to the optimum in polynomial time.

Disjoint Paths and Routing

Given a directed graph with m edges, and k node pairs (s_i, t_i), we wish to find directed paths from s_i to t_i for a maximum number of indices i. These paths shall not share any edges. We also call such paths edge-disjoint.

This is a fundamental problem for routing in networks. Imagine that we want to send goods, information, etc., from source nodes to destination nodes along available directed paths, without unreasonable congestion. In general we cannot send everything simultaneously, but we may try and maximize the number of served requests.

The problem is NP-complete (which we do not prove here), but we present an algorithm with approximation ratio $O(\sqrt{m})$. The square root function does not grow fast, hence this result is not too bad. Yet the guaranteed quality of the solution deteriorates with growing network size. However $O(\sqrt{m})$ is the best possible guarantee one can achieve in polynomial time, and still better than no guarantee at all.

As many other approximation algorithms, this one is a simple and obvious greedy algorithm. The intuitive idea is that short paths should minimize the chances of conflicts with other paths, and shortest paths can be computed efficiently.

Therefore, the proposed algorithm just chooses a shortest path that connects some yet unconnected pair and adds it to the solution, and it iterates

this procedure as long as possible. After every iteration we delete the edges of the path used, in order to avoid collisions with paths chosen later.

However, the idea is not as powerful as one might hope: In each step there could exist many short paths to choose from, and we may easily miss a good one, since we only consider the length as selection criterion. But at least we can prove the $O(\sqrt{m})$ ratio, as follows.

Let I^* and I denote the set of indices i of the pairs (s_i, t_i) connected by the optimal and the greedy solution, respectively. Let P_i^* and P_i denote the selected paths for index i. The analysis works with case a distinction regarding the length: We call a path with at least $\sqrt{m}$ edges long, and other paths are called short. Let I_s^* and I_s be the set of indices i of the pairs (s_i, t_i) connected by the short paths in I^* and I, respectively.

Since only m edges exist, I^* can have at most $\sqrt{m}$ long paths. Consider any index i where P_i^* is short, but (s_i, t_i) is not even connected in I. (This is the worst that can happen to a pair, hence our worst-case analysis focusses on this case.) The reason why the greedy algorithm has not chosen P_i^* must be that some edge $e \in P_i^*$ is already in some P_j chosen earlier. We say that e "blocks" P_i^*. We have $|P_j| \leq |P_i^*| \leq \sqrt{m}$. Every edge in P_j can block at most one path of I^*. Hence P_j blocks at most $\sqrt{m}$ paths of I^*. The number of such bad indices i is therefore bounded by $|I_s^* \setminus I| \leq |I_s|\sqrt{m}$. Finally some simple steps prove the claimed approximation ratio: $|I^*| \leq |I^* \setminus I_s^*| + |I| + |I_s^* \setminus I| \leq \sqrt{m} + |I| + |I_s|\sqrt{m} \leq (2\sqrt{m} + 1)|I|$.

An Approximation Scheme for Knapsack

So far we have seen examples of approximation algorithms whose approximation ratios on every instance are fixed, either as 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 get a solution within a factor $1+\epsilon$ 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 $\sum_{i \in S} w_i \leq W$ (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 pseudo-polynomial. (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. The reasons will become clear 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
$OPT(i, V) = \min(OPT(i-1, V), w_i + OPT(i-1, \max(V-v_i, 0)))$.
(Think a while, to see the correctness.) 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 b and solve the given instance only approximately. The point is that we can divide all the rounded values by the common factor b without changing the solution sets, which gives us again a small problem instance. In the following we work out this idea precisely. We do not specify what "small" and large" means, instead, some free parameter b (integer) controls the problem size.

First we compute new values v'_i as follows: Divide v_i by the 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. 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. 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 we get the final result

$$(1 - \epsilon) \sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i.$$

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 pseudo-polynomial 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 $1/\epsilon$.

Optional: If you have time and want to elaborate more on this theme, it is highly recommended to study the following paper (the title says what it is about). And if you are scared off by the details, maybe you can at least

grasp the problem, the additional difficulty, and the new ideas towards its solution.

Zhou Xu, Xiaofan Lai: A Fully Polynomial Approximation Scheme for a Knapsack Problem with a Minimum Filling Constraint. Algorithms and Data Structures – 12th International Symposium, WADS 2011, Lecture Notes in Computer Science 6844, pp. 704 ff.

Using Linear Programming for Approximation Algorithms

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. 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 (indicating that 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 $x_i \in [0, 1]$. Let S^* be a minimum weight vertex cover, and w_{LP} be 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. The set S is obviously a vertex cover. Moreover, $w_{LP} \leq w(S^*)$ implies $w(S) \leq 2w(S^*)$, since by the rounding we have at most doubled the values of variables 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.

Reductions and Approximability

The class of optimization problems where a solution within a constant factor of optimum can be obtained in polynomial time is denoted APX ("approximable"). There exist problems in APX that do not have a PTAS (unless P=NP). They are called *APX-hard* problems. Such results are shown by reductions, in analogy to NP-hardness results. But beware: A polynomial-time reduction from one problem to another one does in general not imply anything about their approximability. Reductions that establish APX-hardness must also preserve the solution sizes within constant factors. Here we do not develop the whole theory but we illustrate this type of reductions by an example.

A dominating set in a graph is a subset D of nodes such that every node is in D or has at least one neighbor in D. The Dominating Set problem asks to find a dominating set with a minimum number of nodes, in a given graph with n nodes. A minimum dominating set can be approximated within a factor $O(\log n)$ of the optimum size, by a reduction to Set Cover that preserves the solution sizes. (This is a pretty straightforward exercise.) Now a natural question is whether we can approximate dominating sets better, in some other way.

The answer is negative, due to the following reduction from Set Cover to Dominating Set. Consider any instance of Set Cover problem, on a set U of size n, and with subsets $S_i \subset U$ with unit weights. Let I denote the set of all indices i. We construct a graph $G = (V, E)$ with node set $V = I \cup U$. We insert all possible edges in I. Furthermore we insert all edges between $i \in I$ and $u \in U$ where $u \in S_i$. Now we prove that the size of a minimum set cover equals the size of a minimum dominiating set in G. Note that every set cover of size k corresponds to a subset of I which is also a dominating set of size k. Conversely, let D be any dominating set of size k in G. If D contains some $u \in U$, we can replace it with some adjacent node $i \in I$. This yields a set of size at most k which is still dominating. This way we get rid of all nodes in $D \cap U$ and finally obtain a dominating set no larger than k, which is entirely in I. Such a dominating set corresponds to a set cover of size at most k. Together this implies equality.

This polynomial-time and size-preserving reduction shows the following: If we could approximate Dominating Set with a factor better than $O(\log n)$, then we could also do so for Set Cover, which is believed to be impossible. Hence our Dominating Set approximation is already as good as it can be.

Summarizing Remarks about Approximation Algorithms

Most of the practically relevant optimization problems are NP-complete, nevertheless solutions are needed. We call an algorithm an approximation algorithm if it runs in polynomial time and gives a solution close to optimum. The approximation ratio is the ratio of the values of the output and of an optimal solution, minimized or maximized (depending on what type of problem we have) over all instances. It can be analyzed by relating "simple" upper and lower bounds on the values of solutions. Some approaches to the design of approximation algorithms are: greedy rules, solving dual problems (pricing methods), and LP relaxation followed by rounding, and there are many more techniques.

All NP-complete decision problems are "equally hard" subject to polynomial factors in their time complexities, but they can behave very differently as optimization problems. Even different optimization criteria for the same problem can lead to different complexities. Some problems are approximable within a constant factor, or within a factor that mildly grows with some input parameters, and some can be solved with arbitrary accuracy in polynomial time. In the latter case we speak of polynomial-time approximation schemes. One should also notice that the proved approximation ratios are only worst-case results. The quality of solutions to specific instances is often much better. On the other hand, there exist problems for which we cannot even find any good approximation in polynomial time. One example is finding maximum cliques in graphs. However, such "hardness of approximation" results require much deeper proof methods than in the theory of NP-completeness.

If you encounter a problem and wonder how well it might be solvable approximately: There is material on the Web, e.g., "A compendium of NP optimization problems" edited by Crescenzi and Kann.