Advanced Algorithms Course. Lecture Notes. Part 4

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, LP solution algorithms 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.

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 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 a 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 dominating 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.

Network Flow with Applications

The Basic Facts

You are supposed to be already familiar with these basics. This section is inserted only for the sake of completeness, as a reminder, and for providing the terminology and fundamental facts. Internal details of the Ford-Fulkerson algorithm and residual graphs are not in the lecture, and we will not need them in the following sections.

Let $G = (V, E)$ be a directed graph where every edge e has an integer capacity $c_e > 0$. Two special nodes $s, t \in V$ are called **source** and **sink**, all other nodes are called internal. We may suppose that no edge enters s or leaves t. A **flow** is a function f on the edges such that: $0 \leq f(e) \leq c_e$ holds for all edges e (capacity constraints), and $f^+(v) = f^-(v)$ holds for all internal nodes v (conservation constraints), where we define $f^-(v) := \sum_{e=(u,v)\in E} f(e)$ and $f^+(v) := \sum_{e=(v,u)\in E} f(e)$. (As a menominic aid: $f^-(v)$ is consumed by node v, and $f^+(v)$ is generated by node v.) The value of the flow f is defined as $val(f) := f^+(s)$. The **Maximum Flow** problem is to compute a flow with maximum value.

The problem can be written as an LP, but there is also a special-purpose algorithm for Maximum Flow, that we sketch now.

For any flow f in G (not necessarily maximum), we define the **residual graph** G_f as follows. G_f has the same nodes as G. For every edge e of G with $f(e) < c_e$, G_f has the same edge with capacity $c_e - f(e)$, called a **forward edge**. The difference is obviously the remaining capacity available on e. For every edge e of G with $f(e) > 0$, G_f has the opposite edge with capacity $f(e)$, called a **backward edge**. By virtue of backward edges we can "undo" any amount of flow up to $f(e)$ on e by sending it back in the opposite direction. The residual capacity is defined as $c_e - f(e)$ on forward edges and $f(e)$ on backward edges

Now let P be any simple directed $s - t$ path in G_f, and let b be the smallest residual capacity of all edges in P. For every forward edge e in P, we may increase $f(e)$ in G by b, and for every backward edge e in P, we may decrease $f(e)$ in G by b. It is not hard to check that the resulting function f' on the edges is still a flow in G. We call P an **augmenting path** and f' is the augmented flow, obtained by these changes. Note that $val(f') = val(f) + b > val(f)$.

Now the generic **Ford-Fulkerson algorithm** works as follows: Initially let $f := 0$. As long as a directed $s - t$ path in G_f exists, augment the flow f (as described above).

To prove that Ford-Fulkerson outputs a maximum flow, we must show: If no $s - t$ path in G_f exists, then f is a maximum flow.

The proof is done via another concept of independent interest: An $s - t$ **cut** in $G = (V, E)$ is a partition of V into sets A, B with $s \in A, t \in B$. The **capacity** of a cut is defined as $c(A, B) := \sum_{e=(u,v):u\in A,v\in B} c_e$.

For subsets $S \subset V$ we define $f^+(S) := \sum_{e=(u,v):u\in S,v\notin S} f(e)$ and $f^-(S) := \sum_{e=(u,v):u\notin S,v\in S} f(e)$. Remember that $val(f) = f^+(s) - f^-(s)$ by definition. (Actually we have $f^-(s) = 0$ if no edge goes into s.) We can generalize this equation to any cut: $val(f) = \sum_{u\in A}(f^+(u) - f^-(u))$, which follows easily from the conservation constraints. When we rewrite the last expression for $val(f)$ as a sum of flows on edges, then, for edges e with both nodes in A, terms $+f(e)$ and $-f(e)$ cancel out in the sum. It remains $val(f) = f^+(A) - f^-(A)$. It follows $val(f) \leq f^+(A) = \sum_{e=(u,v):u\in A,v\notin A} f(e) \leq \sum_{e=(u,v):u\in A,v\notin A} c_e = c(A, B)$. In words: The flow value $val(f)$ is bounded by the capacity of any cut (which is also intuitive).

Next we show that, for the flow f returned by Ford-Fulkerson, there exists a cut with $val(f) = c(A, B)$. This implies that the algorithm in fact computes a maximum flow.

Clearly, when the Ford-Fulkerson algorithm stops, no directed $s-t$ path exists in G_f. Now we specify a cut as desired: Let A be the set of nodes v such that some directed $s-v$ path is in G_f, and $B = V \setminus A$. Since $s \in A$ and $t \in B$, this is actually a cut. For every edge (u, v) with $u \in A$, $v \in B$ we have $f(e) = c_e$ (or v should be in A). For every edge (u, v) with $u \in B$, $v \in A$ we have $f(e) = 0$ (or u should be in A because of the backward edge (v, u) in G_f). Altogether we obtain $val(f) = f^+(A) - f^-(A) = f^+(A) = c(A, B)$. In words: The flow value $val(f)$ equals the capacity of a minimum cut (which is still intuitive).

The last statement is the famous **Max-Flow Min-Cut Theorem**.

Another important observation is that the Ford-Fulkerson algorithm returns a flow where all $f(e)$ are integer.

Time Complexity of Computing Flows and Cuts

Let n and m denote the number of nodes and edges, respectively.

The Ford-Fulkerson algorithm may need $O(mC)$ time, where C is any trivial upper bound on the flow value, e.g., the sum of capacities of the edges at the source. The factor m comes from the time needed to find an augmenting path, and the factor C is there since at most C augmentations are needed. This time bound is not polynomial in the input length. By a careful choice of augmenting paths one can make the Ford-Fulkerson algorithm polynomial. For instance, taking the shortest augmenting path each time (i.e., with the smallest number of edges) leads to an $O(n^2m)$ time bound, which we will not prove here. This variant is known as Dinitz' algorithm. There exist even faster Maximum Flow algorithms based on somewhat different principles.

Once we have a maximum flow f, we can also compute a minimum cut (A, B) in $O(m)$ additional time. The proof of the Max-Flow Min-Cut Theorem hints to an algorithm for this: A is the set of all nodes reachable from s via directed edges in the residual graph G_f, and B is the rest.