Advanced Algorithms Course. Lecture Notes. Part 4

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 keep the solution sizes within constant factors. Here we do not present the whole theory on a technical level, but we only illustrate this type of reductions by an example.

A dominating set in a graph is a subset D of nodes such that every node not being in D 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. This is easy to show by a reduction to Set Cover. (Do you see how?) A natural question is: Can we approximate dominating sets better?

The answer is negative, due to the following reduction from Set Cover to Dominating Set. (Note that we need a reduction in the opposite direction now!) Consider any instance of the 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 $V = I \cup U$ as follwos. (It is recommneded to draw a sketch or a small example.) We insert all possible edges in I; in other words, we make I a clique. Furthermore we insert all edges between $i \in I$ and $u \in U$ where $u \in S_i$.

Now this bipartite graph "encodes" the Set Cover instance in a natural way. We prove that, in fact, the size of a minimum set cover equals the size of a minimum dominating set in G:

Every set cover of size k corresponds to a subset of I, which is also a dominating set with k nodes (because I is a clique). This was the easy direction.

Conversely, let D be any dominating set of size k in G. If D contains some $u \in U$, we can replace u with some adjacent node $i \in I$. This yields a set of size at most k which is still dominating (because I is a clique). This way we get rid of all nodes in $D \cap U$. Eventually we obtain a dominating set which is entirely in I and no larger than k. Such a dominating set corresponds to a set cover of size at most k.

This polynomial-time and size-preserving reduction implies the following: If we could approximate Dominating Set within a factor better than $O(\log n)$, then we could do so also for Set Cover. But the latter is believed to be impossible. Conditional on this conjecture, our Dominating Set approximation is already the best possible one.

Summarizing Remarks about Approximation Algorithms

Most of the practically relevant optimization problems are NP-complete, nevertheless solutions are needed. Approximation algorithms give guarantees for both the time and the solution quality. They can be analyzed, e.g., by relating "simple" upper and lower bounds on the values of solutions, or by relating items in the optimal and in the algorithmic solutions in some clever way. Some approaches to the design of approximation algorithms are: greedy rules, solving dual problems (pricing methods), and LP relaxation followed by rounding. There are many more techniques, but we could only glimpse into the field.

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 reasonable approximation in polynomial time. One example is finding maximum cliques in graphs. However, such "hardness of approximation" results require much deeper proof methods than NP-completeness results.

Network Flow with Applications

"Everything flows." (Heraclitus)

Maximum Flow and Minimum Cut

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. An $s - t$ flow, or simply 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 mnemonic 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 $val(f) := f^+(s) - f^-(s)$. The **Maximum Flow** problem asks to compute a flow with maximum value.

Do not mess up all these concepts, always distinguish them carefully and denote them properly when you write about them. (A "flow" is a function on all edges, the flow "value" is a single number associated with the flow, etc.) Confusion of notation is not only a formal error, it easily leads to faulty reasoning as well.

One obvious motivation of the problem is shipping of goods from a source (supplier) to a sink (customer). A flow describes a steady stream of goods along the edges (that can be one-way streets) of limited throughput. The conservation constraints say that nothing is added or removed on the way from the source to the sink.

The problem can be written as an LP, but there is also a more efficient special-purpose algorithm for Maximum Flow that we outline 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 clearly 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**. The reason for creating backward edges may be less obvious: By virtue of backward edges we can "undo" any amount of flow up to $f(e)$ on e, by sending it in the opposite direction. As indicated above, the **residual capacity** is defined as $c_e - f(e)$ on forward edges and $f(e)$ on backward edges

Next, let P be any directed $s - t$ path (shorthand for "directed path from s to t") using only directed edges in G_f. Let b be the smallest residual

capacity of all edges in P (the bottleneck residual capacity). On every forward edge e in P we may increase $f(e)$ in G by b. Formally: $f'(e) = f(e) + b$. On every backward edge e in P we may decrease $f(e)$ in G by b. Formally: $f'(e) = f(e) - b$. It is easy to check that the resulting function f' on the edges is still a flow in G. That is, the capacity and conservation constraints are still sarisfied. Accordingly, we call P an **augmenting path** and f' is the augmented flow obtained by these changes. Note that $val(f') = val(f) + b > val(f)$.

We are ready to state the generic **Ford-Fulkerson algorithm**: Initially let $f(e) := 0$ on all edges e. As long as some directed $s-t$ path in G_f exists, augment the flow f as described above, and update G_f.

In order to prove that Ford-Fulkerson outputs a maximum flow we must show exactly this: If no $s-t$ path in G_f exists, then f is, in fact, a maximum flow. (Hence the algorithm can terminate only with an optimal flow.

The proof is done via another concept of independent interest: An $s-t$ **cut** in $G = (V, E)$ is a partitioning of V into sets A, B with $s \in A$ and $t \in B$. When s and t are clear from context, we simply say "cut". The **capacity** of a cut is defined as $c(A, B) := \sum_{e=(u,v):u\in A,v\in B} c_e$. We stress that *any* such partitioning with source and sink at different sides is called a cut; no special properties are assumed.

For any subset $S \subset V$ 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. We can generalize this equation to arbitrary cuts and obtain: $val(f) = \sum_{u\in A}(f^+(u) - f^-(u))$. This follows easily from the conservation constraints, but we omit the tedious calculation details.

Next, 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, the terms $+f(e)$ and $-f(e)$ cancel out in the sum. (Again we omit some intermediate steps.) It remains $val(f) = f^+(A) - f^-(A)$. This finally implies:

$$val(f) \le f^+(A) = \sum_{e=(u,v):u\in A,v\notin A} f(e) \le \sum_{e=(u,v):u\in A,v\notin A} c_e = c(A, B).$$

In words: *The flow value $val(f)$ is bounded by the capacity of any cut.* This conclusion is also plausible and should not be a surprise.

Next we show that, for the flow f returned by Ford-Fulkerson, there exists a cut with $val(f) = c(A, B)$. That is, the above inequality becomes an equation. 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. Consider the following cut. Let A be the set of nodes v such

that some directed $s - v$ path exists in G_f, and $B = V \setminus A$. Since $s \in A$ and $t \in B$, this is actually a cut. Furthermore, this cut has the desired property:

For every directed edge (u, v) with $u \in A$, $v \in B$, we have $f(e) = c_e$ (or v would be in A). For every directed edge (u, v) with $u \in B$, $v \in A$, we have $f(e) = 0$ (or u would be in A because of the backward edge (v, u) that exists in G_f). Altogether we obtain $val(f) = f^+(A) - f^-(A) = f^+(A) = c(A, B)$. So *the maximum flow value $val(f)$ equals the capacity of a minimum cut.* 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 values $f(e)$ on the edges are integers.* This follows immediately from the augmentation rules. We started from integers, and all changes and residual capacities are always integers.

Time Complexity of Computing Flows and Cuts

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

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, e.g., by DFS or BFS, and the factor C is there since at most C augmentation steps are needed. (Every augmentation increases $val(f)$ by at least 1.) This time bound is *not* polynomial in the input length, because C may be exponential.

Note that the "generic" Ford-Fulkerson algorithm does not specify which augmenting path to take. By a careful choice of augmenting paths one can make the Ford-Fulkerson algorithm polynomial. Dinitz' algorithm is the Ford-Fulkerson algorithm that always takes the shortest augmenting path, i.e., one with the smallest number of edges. It runs in $O(n^2m)$ time. Another strategy considers only edges with the largest residual capacities, such that $val(f)$ increases a lot in every augmentation step It achieves $O(m^2 \log C)$ time. In both cases the time analysis is somewhat long and technical, therefore we onit it and only mention the results. There exist even faster Maximum Flow algorithms based on different principles (called Preflow-Push algorithms).

Once we have a maximum flow f, we can also compute a minimum cut (A, B), by using $O(m)$ additional time. The proof of the Max-Flow Min-Cut Theorem yields 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.

Appendix

It may be helpful to illustrate the issues with flows and cuts on a minimalistic example. Consider the network with the following directed edges and capacities.

(s, u) 2
(s, v) 1
(u, v) 3
(u, t) 1
(v, t) 2

We may send 1 flow unit along each of the paths (s, u, t) and (s, v, t), which yields a flow with value 2. Now the edges (s, v) and (u, t) are already exhausted, thus we cannot further improve this solution by greedy steps. The heavy edge (u, v) is not even used.

Alternatively, we may send 2 flow units along the path (s, u, v, t), which is another solution with flow value 2 that cannot be further improved by greedy steps: The edges (s, v) and (v, t) have free capacities, but they are connected "in the wrong direction", by (v, u).

Here the key idea of Ford-Fulkerson applies. The reversed edge (v, u) exists in the residual graph, even with capacity 2. Hence we can send 1 flow unit on (v, u), which means to reduce the flow on (u, v) to $2 - 1 = 1$. This yields the following flow of value 3.

(s, u) 2
(s, v) 1
(u, v) 1
(u, t) 1
(v, t) 2

This flow is also optimal, since the only edge with free capacity is (u, v), the other edges are replaced with the reversed edges, and they cannot be connected to some directed $s - t$ path. The cut provided by the proof of the max-flow-min-cut theorem is $A = \{s\}$ and $B = \{u, v, t\}$ and has capacity 3.