**6.890: Algorithmic Lower Bounds: Fun With Hardness Proofs** Fall 2014

# Lecture 10 Scribe Notes

*Prof. Erik Demaine*

## Overview

This lecture begins a series on inapproximability proving the impossibility of approximation algorithms. We begin with a brief overview of most of the typical approximation factor upper and lower bounds in the world of graph algorithms. Then we'll introduce several general concepts, including new complexity classes (NPO, PTAS, APX, Log-APX, etc.) and stronger notions of reductions that preserve approximability (and inapproximability). Finally, if there's time, we'll cover some APX-hardness reductions.

## Optimization Problems

The types of problems we've seen so far are called decision problems: every instance has a "yes" or "no" answer. These problems are inappropriate for talking about approximation precisely because approximation requires the possibility of being wrong to a lesser or larger degree; this is only possible with a range of possible answers instead of just a boolean "yes" or "no". In order to deal with approximation, we need to define a different type of problem.

An *optimization problem* consists of the following:

- a set of instances of the problem
- for each instance: a set of possible solutions, each with a nonnegative cost
- an objective: either min or max

The goal of an optimization problem is to (for any given instance) find a solution which acheives the objective (minimization or maximization) for the cost.

It is useful to define the following: For a optimization problem $x$, we will use $OPT(x)$ to refer to

- either the cost of the optimal (minimal or maximal as appropriate) solution
- or an optimal solution itself.

Now we can define an NP-optimization problem. The class of all NP-optimization problems is called NPO; it is the optimization analog of NP.

An *NP-optimization problem* is an optimization problem with the following additional requirements:

- all instances and solutions can be recognized in polynomial time

- all solutions have polynomial length (in the length of the instance which they solve)
- the cost for a solution can be computed in polynomial time.

We can convert any NPO problem into an analagous decision problem in NP: For a minimization problem we ask "Is $OPT(x) \leq q$?" and for a maximization problem we ask "Is $OPT(x) \geq q$?". The optimal solution can serve as a polynomial space certificate of a "yes" answer, and so these analagous decision problems are in NP.

This means that NPO is in some sense a generalization of NP problems.

## Approximation Algorithms

Suppose we have an algorithm for an optimization problem. This algorithm takes as input the problem instance and outputs one of the possible solutions. We will call an algorithm $ALG$ a *c-approximation* in several situations:

- in the minimization case if for all valid instances $x$, $\frac{cost(ALG(x))}{cost(OPT(x))} \leq c$ (here $c \geq 1$)
- in the maximization case if for all valid instances $x$, $\frac{cost(OPT(x))}{cost(ALG(x))} \leq c$ (here $c \geq 1$)
- or also sometimes in the maximization case if for all valid instances $x$, $\frac{cost(ALG(x))}{cost(OPT(x))} \geq c$ (here $c \leq 1$)

Usually when we say a $c$-approximation algorithm we are interested in polynomial time algorithms. This is not always the case as exponential time $c$-approximation algorithms can sometimes be useful. However, we will assume in this couse that unless otherwise mentioned, any $c$-approximation algorithm runs in polynomial time.

The value $c$ is generally constant in the size of the input, though we can also consider other approximation algorithms whose failure at achieving the optimal cost increases with instance size according to some function. For examle, we may consider a $\log(n)$-approximation algorithm.

In some sense an approximation algorithm is doing pretty well if it is a $c$-approximation algorithm with some constant value $c$. But sometimes, we can do even better:

A polynomial time approximation scheme (PTAS) is in some sense a $1+\epsilon$ approximation (for any $\epsilon > 0$). A PTAS can be expressed as an algorithm solving an optimization problem which is given an additional input $\epsilon > 0$. The PTAS then produces a solution to the instance that is a $1+\epsilon$ approximation for the optimal solution. The only runtime constraint is that if we fix $\epsilon$, the PTAS must run in time polynomial in the size of the remaining input. Note that this allows very bad runtimes in terms of $\epsilon$. For example, a PTAS can run in time $n^{2^{2^{\epsilon^{-1}}}}$ because for any given value of $\epsilon$ this is a polynomial runtime.

We can now define several complexity classes:

The class $PTAS$ is the set of all NPO problems for which a PTAS algorithm exists.

For any class of functions $F$, the complexity class $F$-$APX$ is the set of all NPO problem for which an $f(n)$-approximation algorithm exists for some $f \in F$.

The class $O(1)$-$APX$ of problems for which a $c$-approximation algorithm exists (with $c$ constant) is referred to simply as $APX$.

Similarly, $Log$-$APX$ is another name for $O(\log n)$-$APX$.

These classes satisfy $PTAS \subseteq APX \subseteq Log\text{-}APX$, and if $P \neq NP$ then equality does not hold.

## Lower Bounds on Approximability

How can we prove lower bounds on approximability? We can use reductions. The simple concept of polynomial time reductions that we use to prove NP-hardness, however, is not sufficient. There are at least 9 different definitions of reductions, of which we will use at least 4 in this class.

What we want to use are approximation preserving reductions. In an approximation preserving reduction, if we wanted to go from problem $A$ to a problem $B$ we need to be able to do the following things:

- for any instance $x$ of $A$, we should be able to produce an instance $x' = f(x)$ of $B$ in polynomial time
- for any solution $y'$ of $x'$, we should be able to find a solution $y = g(x, y')$ to $x$ in polynomial time

The idea here is that $f$ transforms instances of $A$ into instances of $B$ while $g$ transforms solutions for $B$ instances into (in some sense equally good) solutions for the $A$ instance.

Let's refine the above idea further:

A *PTAS reduction* is an approximability preserving reduction satisfying the following additional constraint:

$\forall \epsilon > 0, \exists \delta = \delta(\epsilon) > 0$, such that if $y'$ is a $(1 + \delta(\epsilon))$-approximation to $B$, then $y$ is a $(1 + \epsilon)$-approximation to $A$. Note that here we allow that $f$ and $g$ functions to depend on $\epsilon$.

This reduction is of interest because if $B \in PTAS$ then $A \in PTAS$, or equivalently, if $A \notin PTAS$ then $B \notin PTAS$.

Additionally, this type of reduction also gives you the same facts for membership in APX: if $B \in APX$ then $A \in APX$, and if $A \notin APX$ then $B \notin APX$.

Next we define an *AP reduction*:

A PTAS reduction is also an AP reduction when the $\delta$ function used is linear in $\epsilon$. This is a convenient reduction to use because if $B \in O(f) - APX$, then $A \in O(f) - APX$. For example, this stronger type of reduction would be useful for discussing log approximations.

Next we define a *strict reduction*:

A strict reduction is an AP reduction where $\delta(\epsilon) = \epsilon$. This is a very strong concept of reduction.

Having defined these reductions, we can now define hardness for approximibility. A problem is *APX-hard* if there exists a $PTAS$-reduction from any problem in $APX$ to the given problem. If $P = NP$, then a problem being $APX$-hard implies that it is not in PTAS.

These reductions are all a bit awkward to work with directly because all of the approximability results use a multiplicative factor. In practice, people use L reductions. L reductions are stronger than AP reductions (as we will show) so the existance of an L reduction implies the existance of an AP reduction which implies the existance of a PTAS reduction. Note that L reductions are not stronger than strict reductions.

An L-reduction is an approximation preserving reduction which satisfies the following two properties:

- $OPT_B(x') = O(OPT_A(x))$
- $|cost_A(y) - OPT_A(x)| = O(|cost_B(y') - OPT_B(x')|)$

Let's prove that this is an AP reduction. We will do this in the minimization case.

$OPT_B(x') = O(OPT_A(x))$ so $OPT_B(x') \leq \alpha OPT_A(x)$ for some positive constant $\alpha$.

$|cost_A(y) - OPT_A(x)| = O(|cost_B(y') - OPT_B(x')|)$ so because this is a minimization problem, $cost_A(y) - OPT_A(x) \leq \beta cost_B(y') - OPT_B(x')$ for some positive constant $\beta$.

Define $\delta(\epsilon) = \frac{\epsilon}{\alpha\beta}$. To prove that we have an AP reduction, we need to show that if $y'$ is a $(1+\delta(\epsilon))$-approximation to $B$, then $y$ is a $(1+\epsilon)$-approximation to $A$.

Starting with a $(1+\delta(\epsilon))$-approximation to $B$ we have

$$\begin{aligned}
&\frac{cost_A(y)}{OPT_A(x)} \\
&\qquad \leq \frac{OPT_A(x) + \beta \times (cost_B(y') - OPT_B(x'))}{OPT_A(x)} \\
&\qquad = 1 + \beta \times \frac{cost_B(y') - OPT_B(x')}{OPT_A(x)} \\
&\qquad \leq 1 + \alpha\beta \times \frac{cost_B(y') - OPT_B(x')}{OPT_B(x')} \\
&\qquad = 1 + \alpha\beta \times (\frac{cost_B(y')}{OPT_B(x')} - 1) \\
&\qquad \leq 1 + \alpha\beta \times (1 + \delta(\epsilon) - 1) \\
&\qquad = 1 + \alpha\beta\delta(\epsilon) \\
&\qquad = 1 + \alpha\beta \times (\frac{\epsilon}{\alpha\beta}) \\
&\qquad = 1 + \epsilon
\end{aligned}$$

Thus we have a $(1+\epsilon)$-approximation to $A$ as desired.

# Examples

## Max E3-SAT-E5

This is a stronger form of max 3-SAT. The E stands for "exactly". Every clause has exactly 3 distinct literals and every variable appears exactly 5 times.

This is APX-hard, but we will not prove it.

## Max 3SAT-3

Instead we will prove a slightly different result: Max 3-SAT-3 (each variable appears at most 3 times) is APX-hard.

The normal reduction from 3-SAT to 3-SAT-3 is the following. Suppose varaible $x$ appears some number of times $n(x)$. Make a cycle of implications of size $n(x)$ containing new versions of the variable $x_1, x_2, ..., x_{n(x)}$. An implication $x_i \to x_j$ can be rewritten as a clause $\neg x_i \vee x_j$. These $n(x)$ clauses use each of the new variables twice and can only be satisfied if all of the variables have equal value. Then if we add these new clauses to the formula and change the original occurrences of $x$ into occurrences of the new variables, we can yield a new formula where each variable occurs exactly 3 times which is satisfied if and only if the original formula is satisfied.

This reduction is not going to work to show APX-hardness because in this reduction, leaving only two clauses unsatisfied can result in half of the occurrences of a variable being false and half being true (simply make two implications in the cycle false).

We now describe an L reduction from 3-SAT (which we assume is APX-hard) which does work.

Suppose variable $x$ appears $k$ times, we will make $k$ variables and force all of them to be the same using an expander graph (the way that we used a cycle in the polynomial time reduction).

What is an expander graph? An expander graph is a graph of bounded degree such that for every cut $A, B$ (a partition of the vertices into two sets), the number of cross edges from vertices in $A$ to vertices in $B$ is $\geq \min(|A|, |B|)$. An expander graph is something like a sparse clique in the sense that there are a lot of connections across every cut, but there are only a bounded number of edges for each vertex.

Expander graps are known to exist, and in particular, degree 14 expander graphs exist.

Back to our variable $x$. Construct an expander graph with $k$ vertices. Each vertex in the graph will be a different variable (all of which should turn out equal). For each edge in the expander graph, add two clauses to the formula corresponding with the implications between the two vertices. Next replace all of the original occurrences of the variable $x$ with the $k$ new variables.

The new formula uses exactly 29 occurrences of each variable. So if we show that this is an L reduction we will have proved that Max 3-SAT-29 is APX-hard.

Suppose we have a solution to the new formula. We want to convert this solution into a solution that is at least as good which has all of the versions of a variable with the same boolean value.

So suppose some of the versions of $x$ are true and some are false. The set of true variables ($A$)

and the set of false ones ($B$) form a cut of the expander graph. Thus the number of cross edges is $\geq \min(|A|, |B|)$. Each of those cross edges corresponds to two clauses, at least one of which is currently not satisfied. We change all of the variables to have the value that the majority of them already have. The clauses from the expander graph are now all satisfied, so we have gained at least $\min(|A|, |B|)$ satisfied clauses (the number that we said were not satisfied). The minority that were changed each occur in one other clause. The clauses of the minority might have stopped being satisfied, so we have decreased the number of satisfied clauses by at most $\min(|A|, |B|)$. In total, this new solution must have at least the same score as the old.

Thus we can convert from a solution to the 3-SAT-29 problem to a solution of the original 3-SAT instance.

Now let's make sure this is an L reduction. We added a bunch of contraints, all of which will be satisfied in the optimal solution of the 3-SAT-29 instance. In particular, we see that $OPT(x') = OPT(x) + c$ where $c$ is the number of new constraints. This reduction simply shifts the cost of every solution up, so this satisfies the second property of L reductions. Fortunately, both $c$ and $OPT(x)$ are linear in the number of clauses in the original formula, so we also have the first property of L reductions.

This shows that Max 3-SAT-29 is APX-hard.

To show that Max 3-SAT-3 is APX-hard we reduce again from Max 3-SAT-29. This time, we simply use the original cycle reduction. It is pretty easy to argue that the reduction is an L reduction.

### Independent Set

The general case is very hard, so we consider a special case: independent set in graphs of bounded degree. $O(1)$-degree independent set (which asks to maximize the size of an independent set) is APX-complete. We can prove this with a strict reduction from MAX 3-SAT-3.

Convert each variable into a complete bipartite graph with as many occurrences as necessary of the true variables and as many as necessary of the false. Then convert each clause of size 3 into a traingle between occurrences of the variables (in appropriately positive/negative forms) and each clause of size 2 into an edge between occurrences of the variables.

Each variable can only have one sign of occurrence included in the independent set because every positive version of a variable is adjacent to every negative occurrence. Every clause can have at most one element in the independent set, and it can have that element only if truth values are chosen for the variables so that the clause is satisfied. The size of the idependent set is exactly the number of clauses satisfied in the Max 3-SAT-3 instance.

### Vertex Cover

$O(1)$-degree Vertex Cover asks to minimize the size of a vertex cover for a given bounded degree graph.

We reduce from Independent Set. Independent sets and vertex covers are complements with $|VC| + |IS| = |V|$. Thus maximizing the size of an independent set also minimizes the size of a vertex cover. Since both $|VC|$ and $|IS|$ are $O(|V|)$, we can conclude that both the absolute difference and

ratio conditions of L reductions are satisfied.

### Dominating Set

We can reduce to bounded degree Dominating Set (which asks to find the smallest set of vertices such that every vertex is either selected or neighbors a selected vertex) from bounded degree vertex cover with a very simple construction:

For every edge, add a new vertex neighboring the two endpoints of the edge.

In this new graph, only vertices from the old graph will be selected because any of the new vertices dominates a subset of the vertices dominated by either of its neighbors. Selecting one of its neighbors is then strictly better.

In this new graph, dominating vertices is equivalent to the old idea of covering edges.

# 6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs

Fall 2014