18.415/6.854 Advanced Algorithms	November 7, 2001

Lecture 16

Lecturer: Michel X. Goemans *Scribes: Nicole Immorlica and Mana Taghdiri*

Today we begin our discussion of approximation algorithms. We will talk about:

- NP-hard Optimization Problems
- Inapproximability
- Analysis of Approximation Algorithms

1 NP-hard Optimization Problems

Many of the algorithms in this world of ours deal with optimization. Some of these problems we can solve exactly in polynomial time. We spent the whole first half of this term dealing with these types of problems (linear programming, circulations). Other optimization problems are impossible to solve in polynomial time unless P=NP. We now focus our attention on handling these NP-hard optimization problems.

Definition 1 *An optimization problem is NP-hard if the corresponding decision problem is NP-hard, i.e. a polynomial-time algorithm finding the exact optimum would imply* $P = NP$.

There are many examples of NP-hard optimization problems. In fact, you can find a rather comprehensive and up-to-date list of these problems along with known results at:

`http://www.nada.kth.se/~viggo/problemlist/compendium.html`

We will see three examples of NP-hard optimization problems today.

- **Vertex Coloring:** Given a graph $G = (V, E)$ assign colors to the vertices so that for all $(v, w) \in E$, v is not the same color as w. Minimize the number of colors used.
- **LIN-k-MOD-q:** All the computation in this problem is considered in $\mathbb{Z}_q$. Given a set of variables X and a set of equations E each in k of the variables in X, find an assignment of X. Maximize the number of equations in E that are satisfied by this assignment of X.
- **Scheduling:** Given a set of jobs, a processing time for each job, and a set of processors, assign the jobs to the processors. Minimize the time it takes to complete all the jobs assuming a processor can only process one job at a time.

In order to prove that these are NP-hard optimization problems, we must prove their corresponding decision problems are NP-complete. For example, to prove vertex coloring is NP-hard, we must show that to decide whether a given graph is colorable with less than k colors is NP-complete. As complexity theory is not a pre-requisite for this course, we will just take these facts on faith. However the interested reader can find a nice treatment of NP-completeness in [2, 1, 3]. A proof that k-colorability is NP-complete can be found in the exercises in each of these books.

As these problems are NP-hard, we must abandon all hope of solving them exactly. Instead, we find ways to cope with their intractability. A common method used in practice is to design heuristic algorithms and analyze their performance empirically. As the below definitions for the words heuristic and empirical indicate, we won't be able to study this method in this theoretical class. To see a treatment of this approach, take artificial intelligence.

```
heu.ris.tic \hyu.-'ris-tik\ aj [G heuristisch, fr. NL heuristicus, fr.
   Gk heuriskein to disc]over; akin to OIr fu-ar I have found : serving to
   guide, discover, or reveal; specif : valuable for empirical research but
   unproved or incapable of proof
em.pir.i.cal or em.pir.ic \-i-k*l\ \-ik\ \-i-k(*-)le-\ aj 1: relying on
   experience or observation alone often without due regard for system and
   theory 2: originating in or based on observation or experience 3: capable
   of being verified or disproved by observation or experiment {~ laws} -
   em.pir.i.cal.ly av
```

Instead, we will design polynomial time algorithms that approximately solve these NP-hard optimization problems. We will analyze the solutions returned by these algorithms in the worst-case (i.e. the input for which the solution is farthest from the optimum solution). These algorithms are called approximation algorithms.

Definition 2 *Let $c_{OPT}(x)$ be the optimal solution of a minimization problem P on input x. An α-approximation algorithm for P is a polynomial time algorithm that is guaranteed to deliver a solution such that for all inputs x the value of the solution $c_H(x)$ is $c_H(x) \leq \alpha c_{OPT}(x)$. If P is a maximization problem, we require $c_H(x) \geq \alpha c_{OPT}(x)$.*

2 Inapproximability

The problems we are trying to approximate are NP-hard. Therefore we can't hope to find an α-approximation algorithm for which $\alpha = 1$. But this does not imply that α must be bounded away from one, and in fact for some problems we can find $\alpha = 1 + \epsilon$ for all $\epsilon > 0$ (or $\epsilon < 0$ if it's a maximization problem). Of course, in these cases the running time often depends on ϵ. However, we are not always so lucky. Sometimes we can prove that it is NP-hard to find an approximation algorithm with $\alpha < p(n)$ for some function $p(n)$ (often constant). Such results are called *inapproximability results*.

Traditionally, inapproximability of an NP-hard optimization problem is derived from the NP-completeness proof of the decision problem. We can use this method to prove vertex coloring is inapproximable within $4/3$. This is because k-colorability (for $k > 2$) is NP-complete. Therefore, given a graph G, it is hard to distinguish whether it can be colored with at most 3 colors or if it needs at least 4 colors. If we were able to find a $4/3 - \epsilon$ approximation algorithm, we could call this algorithm on G. If G needs at most three colors, our algorithm will find a coloring that uses at most $\alpha c_{OPT} \leq 3(4/3 - \epsilon) < 4$ colors. But if G needs more than three colors, our algorithm must return a coloring that uses at least four colors. Therefore, this polynomial time algorithm solves the problem we claimed is NP-complete! This can not be, unless P=NP. There are actually stronger (non-constant) inapproximability results for this problem.

Since 1992, another approach to proving inapproximability has emerged. This approach uses PCPs (i.e. probabilistically checkable proofs, not the Communist Party of Peru or phencyclidine or People for a Clearer Phish). As an example, consider the LIN-3-MOD-2 problem. Recall in this problem we have a set of variables x_i and m equations each in three of these variables. All computation takes place in $\mathbb{Z}_2$. For example, we might have

$$x_1 + x_2 + x_3 = 0$$

$$x_2 + x_4 + x_5 = 1$$

$$x_3 + x_4 + x_5 = 1$$

$$x_1 + x_3 + x_5 = 0$$

We try to maximize the number of satisfied equations. First we present a trivial approximation algorithm for this problem. Take x^0 identically zero and x^1 identically one. Note for any given equation, either x^0 or x^1 satisfies it. Therefore, either x^0 or x^1 must satisfy $m/2$ of the equations. As the optimal solution can satisfy at most m equations, this is a 1/2-approximation algorithm. Johan Håstad used PCPs to prove there is no $(1/2+\epsilon)$-approximation algorithm for any $\epsilon > 0$ [4]. As a side note, Håstad's theorem does not hold for LIN-2-MOD-2. Notice our approximation algorithm works only for LIN-k-MOD-2 where k is odd. Can you find an algorithm that works for LIN-2-MOD-2? What's its approximation factor?

3 Analysis of Approximation Algorithms

Since it's hard to characterize c_{OPT}, it's often quite hard to prove that $c_H \leq \alpha c_{OPT}$, directly (Assume we're solving a minimization problem.) Instead, we'll find a good lower bound on c_{OPT} (say LB) and then prove that:

- $LB \leq C_{OPT}$
- $C_H \leq \alpha LB$

Following problems are examples of using such an analysis method.

3.1 Scheduling Problem

Given m machines and n jobs, each with a certain processing time length p_j, we're supposed to minimize

$$c_{max} = \max_j \ c_j$$

where c_j is the completion time of jth job, such that each job be assigned to exactly one machine - no matter which one - and no 2 jobs can overlap on the same machine.

The problem is NP-Hard even in case of 2 machines. In that case we have n jobs with

$$\sum_{i=1}^{n} p_i = 2b$$

and we need to determine whether there's a $S \subseteq \{1, 2, \ldots, n\}$ s.t.

$$\sum_{i \in S} p_i = b$$

or not, which is the partition problem, known to be NP-Complete.

As an approximation algorithm, we introduce Graham's List Scheduling algorithm:

- Consider jobs in any order
- Put job j on the first available machine.

Claim 1 *Graham's List Scheduling is a $(2-1/m)$-approximation algorithm.*

Proof:

let

$$LB = \max\{\frac{\sum p_i}{m}, \ \max_j p_j\}$$

Trivially $c_{OPT} \geq LB$. If c_H is the solution found by Graham's algorithm, let j be the job s.t. $c_j = c_H$. So, at time $c_H - p_j$ all machines were busy. But then

$$c_H - p_j \leq \frac{\sum_{i \in \{1,\ldots,n\}, i \neq j} p_i}{m} \Rightarrow c_H = (c_H - p_j) + p_j \leq$$

$$\frac{\sum p_i - p_j}{m} + p_j \leq \frac{\sum p_i}{m} + (1 - 1/m) \max_j p_j \leq (2 - 1/m) LB$$

□

3.2 Vertex Covering Problem

Given a graph $G = (V, E)$ and a weight function w which maps each vertex v to a non-negative weight $w(v)$, we have to find a vertex cover $S \subseteq V$ s.t.

$$\sum_{v \in S} w(v)$$

is minimized, where $S \subseteq V$ is a vertex cover iff

$$\forall (i,j) \in E,\ \{i,j\} \cap S \neq \varnothing$$

The VC problem is NP-hard (but it is polynomial-time solvable if G is bipartite). We'll use linear programming to find an approximation algorithm.

In order to use LP to find a lower bound (LB) in general, a "relaxation" is used. Let $Q \subset \mathbb{R}^n$ denote the characteristic vectors of all feasible solutions (in our case, vertex covers of G). Let $f(x)$ be the function we are trying to optimize. Instead of optimizing over Q, we optimize over a polytope P such that $Q \subseteq P$. We know that

$$LB = \min_{x \in P} f(x) \leq \min_{x \in Q} f(x) = c_{OPT}$$

so LB is our lower bound. Finally, we have to find a feasible integer solution which will be within a constant factor of LB.

So first, we have to formulate the problem as an integer program. For the above example we can define:

$$\begin{cases} x(v) \in \{0,1\} & \forall v \in V \\ x(u) + x(v) \geq 1 & \forall (u,v) \in E \end{cases}$$

and then try to find

$$\min \sum w(u) x(u)$$

with the idea of

$$x(v) = \begin{cases} 1 & v \in S \\ 0 & o.w. \end{cases}$$

The LP relaxation is:

$$\min \sum_u w(u) x(u) = LB \leq c_{OPT}$$

s.t.

$$\begin{cases} x(u) + x(v) \geq 1 & \forall (u,v) \in E \\ 1 \geq x(u) \geq 0 & \forall u \in V \end{cases}$$

which can be solved in polynomial time. After solving the corresponding LP, we have LB and $0 \leq x^*(u) \leq 1$. But $x^*(u)$ is not always 0 or 1, e.g. in graph K_5 given $w(v) = 1\ \forall v \in V$, $c_{OPT} = 4$ but $LB = 2.5$ with $x^*(u) = 1/2$. We'll solve this problem by rounding:

$$Output:\ S = \{u | x^*(u) \geq 1/2\}.$$

Claim 2 *This is a 2-approximation algorithm.*

Proof:
$\forall (u,v) \in E,\ x^*(u) + x^*(v) \geq 1 \Rightarrow \max(x^*(u), x^*(v)) \geq 1/2$ so, at least one of them is in S, thus, S is a vertex cover and:

$$\sum_{u \in S} w(u) \leq 2 \sum_{u \in S} x^*(u) w(u) \leq 2 \sum_{u \in V} x^*(u) w(u) = 2LB$$

□

Remark 1 *No* $(2-\epsilon)$*-approximation algorithm is known.*

But in this approach, you always have to solve the corresponding LP, exactly. Although it's not hard in this problem, but in general it may be a bottleneck itself. In the next session we'll show how to use duality to overcome this deficiency.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co., 1979.

[3] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[4] Johan Håstad. Some optimal inapproximability results. *ECCC Report TR97-037*, 1997.