# 1 Heavy Hammers for Approximation Algorithms

Today we will explore a powerful tool in the design and analysis of approximation algorithms, linear programs. An instance of a linear program (LP) is an $m$ by $n$ matrix $A$, a vector $\vec{b}$ of length $m$, and a vector $\vec{c}$ of length $n$. The goal is to find an $n$ dimensional vector $\vec{x}$ maximizing $\vec{c}^T \cdot \vec{x}$ while at the same time satisfying $A\vec{x} \leq \vec{b}$. It is also permissible to ask to minimize the objective function subject to $A\vec{x} \geq \vec{b}$.

The following fact about linear programs is often useful in the design of approximation algorithms. If all the entries in $A, \vec{b}, \vec{c}$ are $l$-bit integers (or $l$-bit rationals), then the problem can be solved optimally in time $poly(n, m, l)$. The two most often cited methods for accomplishing this are the ellipsoid method of Khaciyan '80 or the interior point method of Karmarkar '84.

Integer linear programming is also a powerful paradigm into which to cast approximation algorithms. An integer linear program (IP) has the same constraints and goal as a linear program, but the additional constraint that $\vec{x} \in \mathbf{Z}^n$ is added. It is NP-hard to solve general integer linear programs; you are invited to prove this as an exercise. Additionally, most non-trivial restrictions on the class of IP's are still NP-hard. However, there are still two common uses that IP's have in the design of approximation algorithms.

The first use we will cover is the less common. A result of Lenstra '85 implies that solving an IP in a constant number of variables is possible in polynomial time. This is not a trivial result. To get an idea for some of the difficulties in solving an arbitrary integer program, even with the restriction that the number of variables is constant, consider the following classic IP:

Given $a, b$, find $g, u, v, w, y$ such that

$$wg = a$$

$$xg = b$$

$$g = ua + vb$$

This is just the problem of finding the GCD of $a$ and $b$, and it is easily solved using Euclid's algorithm in time polynomial in the bit size. Although Euclid's algorithm is not too complicated, it is non-trivial. Hopefully this conveys the impression that solving integer linear programs over a constant number of variables in polynomial time can still be tricky. We will take Lenstra's result on faith.

The more common use of IP's is to cast a problem into the IP formulation, *relax* the constraint that the solution $\vec{x}$ must lie in $\mathbf{Z}^n$ in order to obtain an LP, and then solve the LP to obtain a lower bound on the true value of the IP.

# 2 Minimizing Makespan

We now present a PTAS for the Minimum Makespan problem due to Hochbaum and Shmoys. A PTAS is a $(1+\epsilon)$-approximation scheme which is polynomial for fixed $\epsilon$. We define the problem as

**Instance:** $n$ processes (also known as jobs), requiring processing time $t_1 \geq ... \geq t_n$ and $m$ identical processors

**Goal:** find an allocation $S_1, ...S_m$ such that

$$\bigcup_{j=1}^{n} S_j = \{1, ...n\} \qquad \text{(we schedule all the jobs)}$$

and minimizing

$$\max_j \{\sum_{i \in S_j} t_i\} \qquad \text{(the total amount of time spent by the busiest processor)}$$

We start with the following basic observations.

(i) We can assume the optimal solution takes time $T$ and that $T$ is known. This works because we can try out a lot of different guesses for $T$, homing in on the right one through the bisection method. (The bisection method is also commonly called binary search in reference to searching on binary trees.)

(ii) Jobs of length $\leq \epsilon T$ can be scheduled later in a greedy fashion. That is, if $t_1, ...t_{n'}$ can be scheduled to take time $\leq (1+\epsilon)T$, and $t_{n'+1}, ..., t_n$ have length $\leq \epsilon T$, then we can add the small jobs greedily to the earliest finishing processor (i.e., stick the small jobs greedily in the lowest weight bins). If we ever get stuck (meaning we can't add another small job to any one of the processors without increasing the total amount of time that processor is working beyond $(1+\epsilon)T$), then all the processors were already spending time $> T$, and so the total amount of work is so great that there is no way to solve the problem in time $\leq T$. Thus we can safely restrict our attention to jobs of at least moderate size in solving the problem. We formalize this as the following claim which we have just proved.

**Claim:** If $t_1, ...t_{n'}$ have length $\geq \epsilon T$ and can be scheduled in time $(1+\epsilon)T$
and $t_{n'+1}, ..., t_n$ have length $\leq \epsilon T$
and the optimal solution requires time $\leq T$,
then just adding in the small jobs greedily at the end will not cause the value of our solution to exceed $(1+\epsilon)T$.

Now we can assume that $t_1, ...t_{n'} \in [\epsilon T, T]$

At this point, in solving the knapsack problem, we rounded everything to one of $\epsilon T, 2\epsilon T, 3\epsilon T, ...T$

However, this is not enough - proceeding down that path only yields a 2-approximation algorithm. Instead, we will use a method which has been called "geometric rounding". We will round the solution to the values $\epsilon T, \alpha\epsilon T, \alpha^2\epsilon T, ...T$ for some $\alpha > 1$. This approach will allow us to achieve an $\alpha$ approximation on our new instance (without the very small jobs).

Setting $\alpha = 1+\epsilon$ yields a total of $k = \frac{1}{\epsilon}\log\frac{1}{\epsilon}$ different job sizes for us to schedule (since this value of $k$ makes $(1+\epsilon)^k \epsilon T = T$). Since $\epsilon$ is constant, both $\alpha$ and $k$ are constant too.

So far, we have subtracted small jobs and scaled other jobs down. This means that the new optimum time to schedule the jobs can only be smaller. Representing our new instance through the addition of a funny hat over the variables, this could be said

$$OPT(\hat{t_1}, ...\hat{t_n}) \leq T$$

Thus, if we can find a solution which takes time $\leq T$ for our new instance, applying the same schedule to our old instance will yield a solution taking time $\leq (1+\epsilon)T$

Note that we're thinking of $k$ as a constant, and we're now down to $k$ different types of jobs. We dub this the Finite Minimum Makespan problem, consisting of

$n_1$ jobs of size $u_1$, $n_2$ jobs of size $u_2$, ... $n_k$ and $u_k$, $m$ processors, and the same constraints and objective function

We also note that $u_1/u_k \geq \epsilon$, so the job sizes are all within a constant factor of each other.

We define the total number of different possible configurations for a single processor to be $l$. Note that $l \leq k^{1/\epsilon}$. This is true because every processor runs at most $1/\epsilon$ jobs (each job takes time at least $\epsilon T$ and the total time doesn't exceed $T$), and each job can be chosen to be one of at most $k$ different types.

Note that $l$ is still a constant in our setting. Call the possible configurations $C_1, ...C_l$.

The goal of the Finite Minimum Makespan problem is thus to find integers $m_1, ...m_l$ that sum to $m$, indicating that we should schedule $m_1$ machines with configuration $C_1$, $m_2$ with $C_2$, etc. At this point, we are ready to invoke Lenstra's heavy hammer[1].

We restate the problem to make it clear that Lenstra's result applies. The variables are $m_1, ...m_l$ and the first constraint is

$$\sum_{j=1}^{l} m_j = m$$

We now add constraints which say that we allocate every job. Denoting by $a_{ij}$ the number of objects of size $u_i$ in configuration $C_j$, we also have

$$\sum_{j=1}^{l} a_{ij} m_j \geq n_i$$

This can be read as "the total number of objects of size $u_i$ which are scheduled is $\geq n_i$".

This is a $k$ by $l$ size IP. The only non-constant things are the bit sizes, but they are all $O(\log(n+m))$. Thus the IP can be solved in polynomial time.

This shows that a PTAS exists for this problem. We haven't calculated the actual running time, but an off the cuff estimate discussed in class was $O(m^{1/\epsilon^3})$.

After discussing the applicability of Lenstra, it was noted that we didn't even need Lenstra's result in this case. Since we are looking for variables $m_1, ...m_l$ in the range $[0...m]$, we can enumerate all possibilities in time $\leq m^l$, compute the resulting makespans, and take the optimum one. Since $l$ is assumed to be constant, this is still polynomial.

As mentioned at the end of the last lecture, Minimum Makespan is strongly NP-complete. Thus, we do not expect to be able to find an FPAS for Minimum Makespan (because we believe P $\neq$ NP). We have just finished exhibiting a PTAS, but it is unknown whether any algorithm in between the two exists. For example, whether there exists a polynomial time algorithm which produces a $(1 + 1/(\log n))$ approximation is an open problem.

## 3 PTAS Defined

We say that problem $\Pi$ has a Polynomial Time Approximation Scheme (PTAS) if, given $\alpha > 1$, we can construct an algorithm $A_\alpha$ that runs in time $poly(n)$ and produces an $\alpha$ approximate solution on every

[1] This is not common usage - the average theoretician on the street is not familiar with Lenstra's heavy hammer.

instance. We are not using the quantifier placement strongly here, which is normal - most algorithms simply take $\alpha$ as a parameter.

Note that problems having an FPAS (or FPTAS using the notation from the last lecture's notes), possess a PTAS by definition.

Many problems possess PTAS's, and we'll see more examples later. A natural and useful classification of NP-hard optimization problems is into those posessing FPAS's, those posessing PTAS's but not FPAS's, and those for which no PTAS exists (assuming P $\neq$ NP).

# 4 MAX-SAT and MAX-CUT

No PTAS's exist for MAX-SAT or MAX-CUT unless P = NP. We won't be seeing a proof of this for a while, but we mention it now to motivate our exploration of these two problems. MAX-SAT is a beautiful optimization problem, and many things naturally reduce to it. We define it formally to be

**MAX-SAT**

Instance: CNF formula $\Phi$ made up of clauses $C_1, ...C_m$ on the variables $x_1, ...x_n$. Each $C_j$ is a disjunction on literals, i.e., $(x_{i_1} \vee x_{i_2} \vee \bar{x}_{i_3} \vee \bar{x}_{i_4})$.

Goal: Find an assignment of true or false to $x_1, ...x_n$ maximizing the total number of satisfied clauses.

**MAX-CUT**

MAX-CUT is also a very close approximation to many actual optimization propblems which occur in practice. Given a graph $G = (V, E)$, we wish to find an $S \subset V$ with the maximum number of edges passing between $S$ and $\bar{S} = V \setminus S$.

We can straightforwardly add weights to either one of these problems. In MAX-SAT, we add weights to the clauses, and in MAX-CUT, we add weights to the edges. This changes the objective function, but not the constraints.

Now we consider MAX $k$-SAT. This is the version of MAX-SAT where we restrict all the clauses to be of size $k$. MAX 3-SAT is both NP-complete and strongly NP-complete. These two properties follow straightforwardly from the NP-completeness of 3-SAT. MAX 2-SAT is a more interesting example; although 2-SAT is polynomial time solvable, MAX 2-SAT is strongly NP-complete. MAX-CUT is also strongly NP-complete, and as an exercise, you are invited to reduce MAX 2-SAT to MAX-CUT.

We now give a simple 2-approximation algorithm for MAX-SAT. Let $asst_T$ be the assignment where all the variables are set to true, and $asst_F$ be the assignment where all the variables are set to false. Since every clause is satisfied by one of the two assignments (or both), one of them satisfies $\geq 1/2$ of the clauses. Since the total number of clauses is an upper bound on the number of clauses which can be satisfied by any assignment, we have achieved a factor of 2 approximation.

There is a similar 2-approximation algorithm for MAX-CUT. This one is paricularly interesting because it stood unimproved for 15 years. The Goemans Williamson SDP for MAX-CUT was the first improvement, and it appeared in the middle of this decade. The 2-approximation algorithm for MAX-CUT is as follows: start with two empty bags, a left bag and a right bag. Arbitrarily select some vertex to put into the left bag. For every subsequent vertex, put it into whichever bag maximizes the number of edges passing between the left and right bags (ignoring edges incident to a vertex which has not yet been added to either bag). Since the total number of edges is an upper bound on the value of the optimal solution, this greedy building of a partition is a 2-approximation algorithm.

We now start the presentation of a 4/3 approximation algorithm for MAX-SAT. This is originally a theorem due to Yanakakis, but we will present a proof by Goemans and Williamson. The basic idea is to set up the problem as an IP, relax it to obtain an LP, and then use a technique called randomized rounding to obtain an integer solution. Randomized rounding is a common and important technique for converting a fractional solution into an integral one.

Associate to every clause $C_j$ a corresponding variable $z_j$ with the property that $z_j = 1$ if $C_j$ is satisfied and 0 otherwise. We also associate a series of LP variables $y_i$ to the problem variables $x_i$ with the property that $y_i = 1$ if $x_i$ is true, and 0 if false. The objective function is to maximize

$$\sum_j z_j$$

subject to a certain set of constraints, one for each clause. Rather than encumber ourselves with unnecessary formalism, we will give an example of the clause constraints that makes clear what we mean. Say that the clause $C_j$ was

$$C_j = x_{i_1} \vee x_{i_2} \vee \bar{x}_{i_3}$$

then the corresponding constraint we add is

$$z_j \leq y_{i_1} + y_{i_2} + (1 - y_{i_3})$$

Tying up loose ends, we also add $0 \leq z_j \leq 1$ and $0 \leq y_i \leq 1$.

Now that we have set up the LP, we solve it using our black box polynomial time LP solver. This produces a fractional solution, an assignment of the $y_i$ such that each one is a real number between 0 and 1. We contruct our integral solution by independently setting each $x_i$ to 1 with probability $y_i$ and 0 otherwise.

So how well do we do on each individual clause relative to $Z_j$? Consider a simple monotone clause $C_1 = x_1 \vee x_2 \vee x_3$ and the corresponding contraint $z_1 \leq y_1 + y_2 + y_3$. It will be clear that our analysis extends to non-monotone clauses, and summing over all the clauses will give us the approximation ratio in expectation. We find immediately that

$$\Pr\left[C_1 \text{ not satisfied}\right] = \prod_{i=1}^{k}(1 - y_i) \leq 1 - (y_1 + y_2 + y_3) = 1 - z_1$$

So we want to investigate the behavior of

$$1 - \prod_{i=1}^{k}(1 - y_i) \text{ vs } \min\{\sum_{i=1}^{k} y_i, 1\}$$

The worst case for us (when the ratio between the two is maximized) is when the variables are all equal and they sum to 1 (this can be proved through a bit of high-school calculus). This yields
$1 - (1 - 1/k)^k$ vs 1
which evaluates to $1 - 1/e$ vs. 1. Thus we have developed a $\frac{1}{1-1/e} \doteq 1.58$ approximation algorithm. In the next lecture, building upon this work, we will achieve a 4/3 approximation algorithm.