Limits of Approximation Algorithms 28 Jan, 2010 (TIFR)

Lec. 2: Approximation Algorithms for NP-hard Problems (Part II)

Lecturer: Prahladh Harsha *Scribe: S. Ajesh Babu*

We will continue the survey of approximation algorithms in this lecture. First, we will discuss a $(1+\varepsilon)$-approximation algorithm for KNAPSACK in time $\text{poly}(n, 1/\varepsilon)$. We will then see applications of some heavy hammers such as linear programming (LP) and semi-definite programming (SDP) towards approximation algorithms. More specifically, we will see LP-based approximation for MAXSAT and MAXCUT. In the following lecture, we will see a SDP-based algorithm for MAXCUT.

The references for this lecture include Lecture 1 of the DIMACS tutorial on Limits of approximation [HC09], Lectures 2, 3 and 4 of Sudan's course on inapproximability at MIT [Sud99], and Section 8 from Vazirani's book on Approximation Algorithms [Vaz04].

2.1 Knapsack

The KNAPSACK problem is defined as follows.

Input:
- n items, each item $i \in [n]$ has weight $w_i \in \mathbb{Z}^{\geq 0}$ and value $v_i \in \mathbb{Z}^{\geq 0}$.
- A KNAPSACK with capacity $c \in \mathbb{Z}^{\geq 0}$.

Output: Find a subcollection of items $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq c$.

Objective: Maximize the total value of the subcollection: $\sum_{i \in S} v_i$

2.1.1 Greedy approach

The following is a natural greedy approach for this problem.

1 Sort the items by v_i/w_i (in non-increasing order). That is, arrange the items such that $v_j/w_j \geq v_{j+1}/w_{j+1}$.

2 Find the smallest k such that $\sum_{i \leq k} w_i > c$

3 Output $S = \{1, \cdots, k-1\}$

It can be shown that this algorithm performs arbitrarily bad wrt approximation. More precisely, for every M, there is a KNAPSACK instance such that the the greedy output is less than OPT/M. We can do slightly better if we replace the final step as follows.

3' If $v_k > \sum_{i=1}^{k-1} v_i$ then output $\{k\}$ else output S

It can be shown that the above modification to the greedy approach yields a 2-approximation.

2.1.2 FPTAS for Knapsack

We will now show that we can get arbitrarily good approximations for Knapsack. More precisely, we will show that for every $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximation algorithm for KNAPSACK running in time $\text{poly}(n, 1/\varepsilon)$.

For this, we will first require the following claim which shows that KNAPSACK is optimally solvable if we allow the algorithm to run in time $\text{poly}(V)$ where V is the maximum value, ie., $\max_{i\in[n]} v_i$ instead of $\text{poly}\log V$. Such algorithms which run in time polynomial in the value of the input integers rather than polynomial in the bit-complexity of the input integers are called pseudo-polynomial algorithms.

Claim 2.1.1. *There is a pseudo-polynomial time algorithm OPT-*KNAPSACK *that solves* KNAPSACK *optimally, i.e. in time poly$(n, V) = O(n^2V)$.*

This claim tells us that if the value of the items are small enough (more precisely, polynomially small), then the problem is optimally solvable. We will assume this claim for now and show how using it, one can construct a $(1+\varepsilon)$-approximation algorithm. The main idea is that as we only want to obtain a $(1+\varepsilon)$-approximation, we can round the values v_i to a smaller-sized set so that the new set of values is only polynomially large and then apply OPT-KNAPSACK on it.

Let $V = \max_{i\in[n]} v_i$ and $K = \varepsilon V/n$.

$(1+\varepsilon)$ approximate algorithm for KNAPSACK

1 Construct $v'_i = \left\lfloor \frac{v_i}{\varepsilon V/n} \right\rfloor = \lfloor v_i/K \rfloor$, $\forall i \in [n]$

2 Run OPT-KNAPSACK over the input $\{(v'_i, w_i), i = 1, \ldots, n\}$ to obtain set S

3 Output S

Analysis: Let O be the optimal sub-collection $O \subseteq [n]$ for the original input $\{(v_i, w_i)\}$. For any sub-collection $A \subseteq [n]$, let $\mathbf{val}(A) = \sum_{i\in A} v_i$ and $\mathbf{val}'(A) = \sum_{i\in A} v'_i$. We want to compare $OPT = \mathbf{val}(O)$ with $\mathbf{val}(S)$. We know that $Kv'_i \le v_i < Kv'_i + K$ and $V \le OPT$. Hence,

$$\begin{aligned} \mathbf{val}(O) - K\mathbf{val}'(O) &\le |O|K \le nK = \varepsilon V \le \varepsilon OPT \\ \Rightarrow OPT - K\mathbf{val}'(O) &\le \varepsilon OPT \\ \Rightarrow K\mathbf{val}'(O) &\ge (1-\varepsilon)OPT \end{aligned}$$

S is an optimal set for the input $\{(v'_i, w_i)\}$. Therefore,

$$\mathbf{val}(S) \ge K\mathbf{val}'(S) \ge K\mathbf{val}'(O) \ge (1-\varepsilon)OPT.$$

Hence assuming the Claim 2.1.1, there is a $(1+\varepsilon)$-approx algorithm that runs in time $poly(n, n/\varepsilon) = poly(n, 1/\varepsilon)$. Such an algorithm is called a fully polynomial time approximation scheme (FPTAS)

Definition 2.1.2 (FPTAS)**.** *A optimization problem* Π *is said to have fully polynomial time approximation scheme (FPTAS) if there exists an algorithm* A *such that on input instance* I *of* Π *and* $\varepsilon > 0$*, the algorithm* A *outputs a* $(1+\varepsilon)$*- approximate solution in time* $poly(|I|, 1/\varepsilon)$

We conclude by proving Claim 2.1.1.

Pseudo polynomial algorithm for OPT-KNAPSACK**:** The claim is proved via dynamic programming. Typically, all pseudo-polynomial algorithms are obtained via dynamic programming. Define

$$f_{k,v} = \min_{A \subseteq [k], \mathbf{val}(A) \geq v} \left(\sum_{i \in A} w_i \right).$$

The table of $f_{k,v}$'s can be computed using the following recurrence by dynamic programming.

$$\begin{aligned} f_{1,v} &= \begin{cases} w_1 & v_1 \geq v \\ \infty & \text{otherwise} \end{cases}, \forall v \in \{1, \cdots, nV\} \\ f_{k+1,v} &= \min \{ f_{k,v}, f_{k,v-v_k} + w_k \} \end{aligned}$$

The optimal value is then obtained from the $f'_{k,v}s$ using the formula

$$OPT = \max\{v | f_{n,v} \leq c\}.$$

We have thus shown that

Theorem 2.1.3. KNAPSACK *has an FPTAS.*

2.2 Minimum Makespan

The MINIMUM-MAKESPAN problem is defined as follows.

Input:
- n jobs, each job $j \in [n]$ takes time t_j.
- m processessors.

Output: Find a partition of $[n]$ into m sets $(S_1, \cdots, S_m)$ so as to minimize the max time on any processor that is $min_{S_1,\cdots,S_m} max_j \sum_{i \in S_j} t_i$.

Like KNAPSACK, MINIMUM-MAKESPAN has a $(1+\varepsilon)$-approximate algorithm for every $\varepsilon > 0$, however, unlike KNAPSACK this algorithm runs in time $O(n^{1/\varepsilon})$. We will assume this algorithm without proof (for details see [Vaz04, Section 10]).

Thus, even though we have very good approximation for MINIMUM-MAKESPAN, the dependence of the running time on ε is not polynomial. Such, approximation schemes are called polynomial time approximation scheme (PTAS).

Definition 2.2.1 (PTAS)**.** *An optimization problem* Π *is said to have a polynomial time approximations scheme (PTAS), if for every* ε*, there is an algorithm* A_ε *that on input* I *outputs an* $(1+\varepsilon)$ *approximate solution in time* $poly(|I|)$*.*

Note the order of quantifier, first ε and then A_ε. Thus, the running time of algorithm A_ε can have arbitrary dependence on ε (eg., $poly(n, 1/\varepsilon), n^{\text{poly}(1/\varepsilon)}, n^{2^{1/\varepsilon}}$ etc). Clearly, if Π has an FPTAS, then Π has a PTAS.

2.3 MAXSAT and MAXCUT

The first of the problems is MAXSAT.

Input: A set of m clauses on n boolean variables: $c_1, \cdots, c_m$
eg: $c_i = (x_{i_1} \vee \bar{x}_{i_2} \vee x_{i_3})$

Output: An assignment

Goal: Maximize number of clauses being satisfied

There exists a trivial 2-approximation: Try the all-true and all-false assignments. Every clause is satisfied by at least one of them, thus, the best of the two satisifies at least half the total number of clauses and thus gives a 2-approximation.

The second of the problems is MAXCUT (which is actually a special case of MAXSAT).

Input: Graph $G = (V, E)$ undirected

Output: A cut$(S, \bar{S})$, $S \subseteq V$

Goal: Maximize $E(S, \bar{S})$ (the number of edges crossing the cut)

MAXCUT has an easy 2-approximation based on a greedy approach.

1. Put the first vertex in S
2. Iterate through the rest of the vertices and incrementally add them to one of S or $\bar{S}$, based on which of these options are currently better.

It is easy to see that this greedy approach ensures that at least half the edges are cut and thus gives a 2-approximation.

2.3.1 Randomized Rounding

We will now demonstrate the randomized rounding technique taking the example of MAXSAT. This will give us an alternate 2-approximation for MAXSAT.

The rounding procedure is as follows:

$$\text{For each variable } x_i \text{ set it to true with probability } \frac{1}{2} \text{ and false otherwise.}$$

For any clause C_j, the probability that it is not satisifed is exactly $1/2^{k_j}$, where k_j is the number of literals in C_j. Let I_j be the indicator variable such that

$$I_j = \begin{cases} 1 & \text{if } C_j \text{ is satisfied} \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, $\Pr[I_j = 1] = 1 - 1/2^{k_j}$. If C denoted the number of satisfied clauses, then $C = \sum_{j=1}^{m} I_j$. We thus have $\mathbb{E}[C] = \sum_{j=1}^{m}(1 - 1/2^{k_j}) \geq m/2$. So in expection, we satisfy atleast half the number of clauses. Observe that this is a randomized procedure and the above analysis gives only a guarantee on the expected number of clauses. Both of these can be

overcome. We can repeat this randomized algorithm several times to get an assignment that performs nearly as well as a random assignment. Furthermore, the randomzied algorithm can be derandomized using the method of conditional expectations. But for the purposes of this course, we will be satisifed with such randomized algorithms with guarantees only on the expected value of the output.

The following observation of the above trivial randomized algorithm will come useful. Note that the algorithm performs better for wider clauses. More precisely, if all the clauses have atleast k literals, then $E[C] \geq m(1 - 1/2^k)$ and we thus have a $1/(1 - 2^{-k})$-approximation algorithm.

2.4 Heavy hammers for approximation – Linear Programming

We will now see how to use heavy hammers such as LP-solvers towards approximation algorithms.

2.4.1 Linear Program (LP) and LP-solvers

Recall a linear program from Lecture 1.

$$\begin{aligned} \textbf{Maximize} \quad & C^T x \\ \textbf{subject to} \quad & Ax \leq b \end{aligned}$$

where $c \in \mathbb{R}^n, b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$. By now, we now know that such linear programs can be solved in polynomial time using either the interior points methods [Karmarkar '84] or the ellipsoid method [Khachiyan '79]. More precisely, it is known that LPs can be solved in time polynomial in n, m and l, where $l = \#$ bits required to represent the elements of A, b, c. We will use these LP-solvers as black-boxes while designing approximation algorithms. In other words, we will try to frame the given problem, rather a relaxation of the problem as a LP, solve the LP and then perform some rounding to obtain an integral solution.

2.4.2 LP-based approximation algorithm for MAXSAT

We first restate MAXSAT as an Integer Program (IP) and relax it to an (LP),
IP for MAXSAT:

- For each variable x_i we have a 0-1 variable z_i.
  z_i is supposed to indicate if x_i is true. i.e., $z_i = 1$ iff $x_i = True$
- For each clause C_j we have a 0-1 variable y_j. y_j is supposed to indicate if C_j is satisfied. i.e., $y_j = 1$ iff C_j is satisfied
- Constrants on y_j and x_i's that check if y_j is 1 iff one of the literals in C_j are true.
  Instead of stating in generally, let us take a typical clause, say $C_2 = (x_1 \vee x_2 \vee \bar{x}_3)$. It will be easy to see that this can be extended to all clauses. For C_2, we will impose the constraint $E_2 : y_2 \leq z_1 + z_2 + (1 - z_3)$. This ensures that y_2 is 1 iff one of z_1, z_2 or $(1 - z_3)$ are 1.

Thus, the IP is as follows:

$$\begin{aligned}
&\max \sum y_j \\
&\text{subject to constraint } E_j, \forall j \in [m] \\
&\qquad z_i \in \{0,1\}, \forall i \in [n] \\
&\qquad y_j \in \{0,1\}, \forall j \in [m]
\end{aligned}$$

Notice that but for the last two 0-1 constraints all the other constraints are linear. We now relax the last two constraints to be linear as follows.

$$\begin{aligned}
&\max \sum y_j \\
&\text{subject to constraint } E_j, \forall j \in [m] \\
&\qquad z_i \in [0,1], \forall i \in [n] \\
&\qquad y_j \in [0,1], \forall j \in [m]
\end{aligned}$$

Now that we have relaxed the original problem to a LP, we can solve it by running the LP-solver to obtain the LP optimal solution OPT_{LP}.

Observe that the solution space has only increased by this relaxation. Any feasible integral solution to the original IP is a feasible solution to the LP. Hence, we call this program, a *LP relaxation* of the original IP. Furthermore, since it is a relaxation the LP optimal is at least the OPT (i.e., $OPT_{LP} \geq OPT$). However, the LP optimal need not be an integral solution. In fact, this ratio between the LP optimal fractional solution and the integral solution is called the integrality gap of the relaxation.

$$\text{Integrality Gap} = \max_{I: I \text{ is an instance}} \frac{\text{LP fractional solution}}{\text{Integral solution}}.$$

Recall that while designing an approximation algorithm, we compared the quality of the algorithm against a easily computable upper bound on the actual optimum (lower bound in case of minimization problems). The LP optimal is a natural candidate for such an upper bound. In fact, in almost all LP-relaxation based approximation algorithms, we will compare the quality of the algorithm against the true optimum by comparing it against the LP optimum.

Rounding: Though we have solved the LP, we have not yet solved the original problem or an approximation of it since the LP optimal may give a solution in which some x_i, y_j are fractions and not 0-1 solutions. We need to design a rounding mechansim to obtain integral solutions from this fractional solutions. There are two natural rounding algorithms.

Deterministic Rounding: For each variable x_i, set x_i to true if $z_i > 0.5$ and false otherwise.

Randomized Rounding: For each variable x_i independently, set x_i to true with probability z_i and false with probability $(1 - z_i)$.

We will use the latter rounding scheme for this problem. Let us now analyse the quality of this rounding algorithm by comparing its expected output to the LP optimum.

Analysis: For ease of explanation, we will do the analysis only for monotone formulae, that is the literals in each clause appear in their uncomplemented form. It will be easy to see that this analysis extends to all formulae. Consider any such clause, say $C_j = z_{i_1} \vee z_{i_1} \vee \cdots \vee z_{i_{k_j}}$ with k_j literals. The probability that C_j is not satisified is exactly $\prod_{l=1}^{k_j}(1 - z_{i_l})$. Thus, the expected number of clauses satisified is as follows.

$$\begin{aligned}\mathbb{E}\left[\text{number of satisfied clauses}\right] &= \sum_{j=1}^{m} \Pr\left[C_j \text{ is satisified }\right] \\ &= \sum_{j=1}^{m} \left[1 - \prod_{l=1}^{k_j}(1 - z_{i_l})\right]\end{aligned}$$

On the other hand, the LP optimum is $\sum_{j=1}^{m} y_j = \sum_{j=1}^{m} \min\left\{\sum_{l=1}^{k_j} z_{i_l}, 1\right\}$. We will compare the rounded solution to the LP optimum by comparing each term in one summation against the corresponding term in the other summation, i.e., $1 - \prod_{l=1}^{k_j}(1 - z_{i_l})$ vs. $y_j = \min\left\{\sum_{l=1}^{k_j} z_{i_l}, 1\right\}$. We would like to determine the worst possible ratio between

$$1 - \prod_{l=1}^{k_j}(1 - z_{i_l}) \quad \text{vs} \quad y_j = \min\{1, \sum z_{i_l}\}$$

The worst ratio is (i.e., the ratio is minimized) when all the z_{i_l} are equal, i.e., $z_{i_l} = 1/k$, in which case the ratio is $1 - (1 - 1/k)^k \geq 1 - e^{-1}$(See Appendix A for the detailed analysis). Thus, the worst ratio between the expected number of clauses satisfied and the LP optimum is at least $1 - e^{-1}$. Thus, this gives us a $1/(1 - e^{-1}) \approx 1.582$-approximation algorithm.

Observe that the ratio of $1 - e^{-1}$ is attained only for very large k. For smaller k, the ratio $1 - (1/k)^k$ is much better. Thus, in contrast to the vanilla rounding discussed in Section 2.3.1, the LP-based aproximation performs well when the clauses are small. Thus, if all the clauses are narrow it is better to perform the LP-based approximation while if all the clauses are wide, it is better to perform the vanilla rounding algorithm. What about, when we have clauses of both types? In the next section, we will see that the simple algorithm of choosing the best of these two algorithms yields a $4/3 \approx 1.333$-approximation.

2.5 MAXSAT: Best of Vanilla Rounding and LP-based rounding

Conisder the algorithm which on a given MAXSAT, performs both the vanilla rounding algorithm and the LP based rounding algorithm and chooses the better of the two. We can

analyse this algorithm as follows.

$$\begin{aligned}
\mathbb{E}[\max(\text{vanilla}, \text{LP-based})] &\geq \frac{1}{2}\left(\mathbb{E}[\text{vanilla}] + \mathbb{E}[\text{LP-based}]\right) \\
&\geq \sum_{j\in[m]} \left[\frac{1}{2}\left(1 - \frac{1}{2^{k_j}}\right) + \frac{1}{2}\left(1 - \left(1 - \frac{1}{k_j}\right)^{k_j}\right) y_j\right] \\
&\geq \sum_{j\in[m]} \left[\frac{1}{2}\left(1 - \frac{1}{2^{k_j}}\right) y_j + \frac{1}{2}\left(1 - \left(1 - \frac{1}{k_j}\right)^{k_j}\right) y_j\right] \\
&\geq \sum_{j\in[m]} \left[1 - \frac{1}{2^{k_j+1}} - \frac{1}{2}\left(1 - \frac{1}{k_j}\right)^{k_j}\right] y_j \\
&\geq \frac{3}{4}\sum y_j = \frac{3}{4} OPT_{LP}
\end{aligned}$$

The last inequality follows since $1 - \frac{1}{2^{k_j+1}} - \frac{1}{2}\left(1 - \frac{1}{k_j}\right)^{k_j}$ is equal to $3/4$ for $k_j = 1, 2$ and strictly greater than $3/4$ for all other k_j. We thus have a $4/3$-approximation for MAXSAT. Such an approximation was first obtained by Yannakakis, but the above algorithm is due to Goemans and Williamson.

References

[HC09] Prahladh Harsha and Moses Charikar. *Limits of approximation algorithms: PCPs and unique games*, 2009. (DIMACS Tutorial, July 20-21, 2009). arXiv:1002.3864.

[Sud99] Madhu Sudan. *6.893: Approximability of optimization problems*, 1999. (A course on Approximability of Optimization Problems at MIT, Fall 1999).

[Vaz04] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2004.

A Appendix

We show that $\frac{1-\prod(1-z_{i_l})^k}{y_j} \geq 1 - \left(1 - \frac{1}{k}\right)^k$ where $y_j = \min\{1, \sum z_{i_l}\}$. The two cases to consider are as follows.

- Suppose $y_j = 1$: Therefore,

$$\begin{aligned}
&\sum z_{i_l} \geq 1 \\
&\frac{\sum z_{i_l}}{k} \geq \frac{1}{k} \\
\Rightarrow\ &\left(1 - \frac{\sum z_{i_l}}{k}\right)^k \leq \left(1 - \frac{1}{k}\right)^k \\
\Rightarrow\ &1 - \left(1 - \frac{\sum z_{i_l}}{k}\right)^k \geq 1 - \left(1 - \frac{1}{k}\right)^k \\
\Rightarrow\ &1 - \left(\frac{\sum (1 - z_{i_l})}{k}\right)^k \geq 1 - \left(1 - \frac{1}{k}\right)^k \\
\Rightarrow\ &1 - \prod (1 - z_{i_l})^k \geq 1 - \left(1 - \frac{1}{k}\right)^k \left(\text{using AM-GM inequality i.e. } \prod a_i \leq \left(\frac{\sum a_i}{k}\right)^k\right).
\end{aligned}$$

- Suppose $y_j = \sum z_{i_l}$: Therefore, $\sum z_{i_l} < 1$. The function $f(x) = 1 - \left(1 - \frac{x}{k}\right)^k$ is concave in the range $0 \leq x \leq 1$ (reason: $f''(x) = -\frac{k-1}{k}\left(1 - \frac{x}{k}\right)^{k-2}$ is negative in this range). The line $x\left(1 - \left(1 - \frac{1}{k}\right)^k\right)$ has the same value as $f(x)$ at $x = 0$ and $x = 1$ (i.e it is a secant line). Hence, by definition of concave functions, $1 - \left(1 - \frac{x}{k}\right)^k \geq x\left(1 - \left(1 - \frac{1}{k}\right)^k\right)$. Setting $x = \sum z_{i_l}$ and we get $1 - \left(1 - \frac{\sum z_{i_l}}{k}\right)^k \geq \left(\sum z_{i_l}\right)\left(1 - \left(1 - \frac{1}{k}\right)^k\right)$. Applying the AM-GM inequality, we get $\frac{1 - \prod\left(1 - z_{i_l}\right)^k}{\sum z_{i_l}} \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right)$.