CS 224: Advanced Algorithms Spring 2017

Lecture 13 — March 7, 2017

Prof. Jelani Nelson *Scribe: Hongyao Ma*

Today

- PTAS/FPTAS/FPRAS examples
  - PTAS: knapsack
  - FPTAS: knapsack
  - FPRAS: DNF counting
- Approximation algorithms via Semidefinite Programming (SDP)

1 PTAS Knapsack

The Knapsack Problem

- Given weight W of knapsack and weights/values of n items: $w_1, \ldots, w_m$, $v_1, \ldots, v_n$.
- Assume that the capacity W, w_i's and v_i's are integers.
- Assume $w_i \leq W$, w.l.o.g., since we can ignore all items with weights larger than the capacity.

$$\begin{aligned} \max \ & \sum_{i=1}^{n} x_i v_i \\ \text{s.t.} \ & \sum_{i=1}^{n} x_i w_i \leq W \\ & x_i \in \{0,\ 1\},\ \forall i \end{aligned}$$

For PTAS, we want to achieve $\geq (1-\epsilon)\text{OPT}$ in time $O(f(\epsilon)n^{g(\epsilon)})$.

1.1 Greedy Algorithms

Greedy Algorithm Sort items in the order: $v_1/w_1 \geq v_2/w_2 \cdots \geq v_n/w_n$.

- Can prove that this is optimal for fractional knapsack problem, but:
- Let $v_1 = 1.001$, $w_1 = 1$, $v_2 = W$, $w_2 = W$, we can see that for this instance, this is no better than a W-approximation.

How can we improve the performance of the greedy algorithm?

Modified Greedy (ModGreedy)

1. Run greedy to get solution S_1
2. Let $S_2 = \{\text{item with the largest value}\}$
3. Return whichever of S_1, S_2 that has more value

Claim 1. *ModGreedy achieves value* $\geq \text{OPT}/2$*, i.e. it's a "2-approximation"*

Lemma 2. *If greedy takes items* $1, 2, \ldots, k-1$ *in the fractional solution, we know*

$$\sum_{i=1}^{k} v_i \geq \text{OPT},$$

Proof. Since the kth item might not be taken in full in the optimal fractional solution, and that the optimal solution of the integral solution is no better tha the optimal solution of the LP relaxation,

$$\sum_{i=1}^{k} v_i \geq \text{OPT}_{\text{frac}}(LP) \geq \text{OPT}$$

□

Proof of Claim 1. We know from the lemma,

$$(v_1 + \cdots + v_{k-1}) + v_k \geq \text{OPT}$$

We know that one of $\sum_{i=1}^{k-1} v_i$ and v_k is at least OPT/2, therefore ModGreedy achieves at least maximum of the two ($v(S_1) \geq \sum_{i=1}^{k-1} v_i$, and $v(S_2) \geq v_{\max} \geq v_k$), which is at least OPT/2. □

1.2 PTAS for Knapsack

ModGreedy gives us a 2-approximation. Now we try to improve the approximation ratio. First, observe that Greedy achieves at least OPT - $v_{\max}$, since

$$\sum_{i=1}^{k-1} v_i \geq \text{OPT} - v_k \geq \text{OPT} - v_{\max}.$$

If no item has value larger than $\epsilon \cdot \text{OPT}$, we know that the Greedy algorithm is a $(1-\epsilon)$ approximation.

Also observe that the optimal set contains at most $1/\epsilon$ items with value $\geq \epsilon\text{OPT}$ — otherwise the value of the optimal solution would exceed OPT. Denote $S \subseteq [n]$ as the set taken by the optimal solution, and let $H \subseteq S$ be the set of items with value $\geq \epsilon \cdot \text{OPT}$. We now from the above argument that $|H| \leq 1/\epsilon$.

Known OPT Assume that we know OPT, we can run the following algorithm:

1. Try all possibilities for sets H s.t. $|H| \leq 1/\epsilon$
2. While trying some particular H:
   - throw out any item with value $\geq \epsilon$OPT
   - run greed to get set G
3. return the $H \cup G$ with best value ever found

When the guessed H correctly, we know that the loss from running greedy to get set G is at most $\epsilon \cdot \text{OPT}$, since the size of the elements that are not thrown away is bounded by $\epsilon \cdot \text{OPT}$. Therefore we get $v(H \cup G) \geq \text{OPT} - \epsilon\text{OPT} = (1-\epsilon)\text{OPT}$.

Without OPT How to get around not knowing OPT?

1. Guess OPT up to a factor of 2. There are at most $\log(\sum_1^n v_i)$ possibilities, or
2. Use ModGreedy to get a 2-approximation of OPT first, or
3. Try all $\leq n$ thresholds

Runtime The run time of the algorithms proposed above is $(n+1)^{1/\epsilon} \cdot \text{poly}(n)$, since the number of sets with size at most k is at most $(n+1)^k$ (k times I can either pick one of the n elements or pick an $(n+1)$st "null" element, which does enumerate all size at most k sets, with overcounting), and the cost of trying different thresholds and to run the greedy on the rest is $\text{poly}(n)$.

2 FPTAS Knapsack

FPTAS achieves $\geq (1-\epsilon)\text{OPT}$ in time $\text{poly}(n, 1/\epsilon)$. The FPTAS dominates the PTAS for the knapsack problem, but there exists problems for which PTAS exists but no FPTAS exists.

Existence of FPTAS Why can't we always hope for a FPTAS? Consider vertex cover as an example, where the goal is to cover all edges using fewest number of vertices. For vertex cover, even the decision problem is hard. If we have an FPTAS, take $\epsilon < 1/2n$, we would get an objective at most $(1+\epsilon)\text{OPT} \leq \text{OPT} + 1/2$ — since OPT is an integer, this would give us what OPT is. This means that if we have an FPTAS, we will be able to solve the vertex cover problem optimally, which would imply P = NP.

Dynamic Programming for Kanpsack We know that we can already solve knapsack exactly in time $O(nV)$ using dynamic programming. Let

$$f(i, v) = \min\{ \text{ total weight of items, using only items } 1, \ldots, i \text{ to achieve value } b\},$$

and if value v is not possible using items from $\{1, \ldots, i\}$, set $f(i, v)$ to be infinity. We then take the set that achieves maximum b s.t. $f(n, b) \leq W$. This is pseudo polynomial time since describing V needs only $\log V$ bits.

FPTAS for Knapsack We will achieve $O(n^3/\epsilon)$ time today, and the best known is $O(n \lg 1/\epsilon + 1/\epsilon^4)$ by Lawler in Math. Oper. Res. '79[2]. The idea is that the DP solution is not too bad if V is small, so let us transform the problem into one with a smaller V, hoping not to lose too much during the process.

Approach:

1. Set $\alpha = n/(\epsilon \cdot v_{\max})$, and for all i, set $v_i' = \lfloor \alpha v_i \rfloor$.
2. Solve the problem with v_i' using the $O(nV')$ algorithm.

Runtime and Approximation Ratio Since the new total value $V' = \sum_{i=1}^{n} v_i' \leq n \cdot \max\{v_i'\} \leq n^2$, we know that the total run time is $O(nV') = O\left(n^3/\epsilon\right)$. What is left to prove is that the solution to the transformed problem is a $(1-\epsilon)$ approximation to the original problem.

First, we know that the $\mathrm{OPT}' \geq v'_{\max}$ since the optimal solution of the transformed problem can take at least the most valuable item. Moreover, since the v_i's are upper bounded by n/ϵ and the total number of items is n, we get:

$$\frac{n}{\epsilon} \leq \mathrm{OPT}' \leq \frac{n^2}{\epsilon}.$$

Note we can assume $1/\epsilon$ is an integer so that indeed $\lceil \frac{n}{\epsilon} \rceil = \frac{n}{\epsilon}$ (if not, round ε down so that $1/\varepsilon$ is an integer, which changes ε by at most a factor of 2 and thus changes our final runtime by a constant factor)

Now observe that the error comes from the floors, o.w. the solution would have been accurate. The amount of $val(S)$ lost due to rounding is at most the size of S: $|S| \leq n \leq \epsilon \mathrm{OPT}'$. We have the following lemma:

Lemma 3. *Let $v(S) = \sum_{i \in S} v_i$ and $v'(S) = \sum_{i \in S} v_i'$, we have*

$$\alpha v(S) - |S| \leq v_i'(S) \leq \alpha v(S).$$

Now we will show that if A is the optimal set for the ransformed problem, and A^* is the actual optimal solution for the original problem,

$$v(A) \geq (1-\epsilon) v(A^*).$$

How do we prove this? Since A is the optimal solution of the transformed problem, its value is at least as much as A^* in the transformed problem. Also consider $|A^*| \leq n$, we get

$$v(A) \geq \frac{1}{\alpha} v'(A) \geq \frac{1}{\alpha} v'(A^*) \geq \frac{1}{\alpha} (\alpha v(A^*) - |A^*|) \geq v(A^*) - \epsilon \cdot v_{\max} \geq \mathrm{OPT} - \epsilon \mathrm{OPT}.$$

3 FPRAS DNF Counting

FPRAS Recall that a FPRAS achieves a $(1 \pm \epsilon)$ approximation in time $\text{poly}(n, 1/\epsilon)$ with probability at least 2/3. As we have seen in the problem sets, we can repeatedly do this for multiple times, and then take the median. The median only fails only if more than half of the trails fails.

Here, we are going to do a counting problem instead of an optimization problem: counting solutions to DNF [1].

Counting Problems Approximate DNF Counting is known to be #P-Complete.[1]. In a counting problem, we would like to know "how many sets that satisfy this or not". This class of problems is harder than NP-hard problems, since if we can count, we are able to be able to solve the original problem like SAT. Sometimes the counting problem is hard, but the original problem is still easy.

DNF Formula (Disjunctive Normal Form)

- $C_1 \vee C_2 \vee \cdots \vee C_m$
- $C = (x_{i_1} \wedge x_{i_2} \wedge \dots)$
- $x \in \{0,1\}^n$

Approximate DNF counting would like to count the number of x out of the 2^n possible assignments s.t. a DNF formula is satisfied. The decision problem is easy— as long as there doesn't exits a little x and its negation, the clause is satisfiable. The counting problem, however, is hard. Recall that a CNF formula is of the form $(\vee \ \vee \dots) \wedge (\vee \ \vee \dots) \wedge \dots$. Let a CNF formula be

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

we know that the negation of ϕ is a DNF:

$$\bar{\phi} = \bar{C}_1 \vee \bar{C}_2 \vee \cdots \vee \bar{C}_m$$

Knowing CNF is hard implies DNF is hard, since if we can count the number of solutions to $\bar{\phi}$ (say N), we are able to count the number of solutions to ϕ would be $2^n - N$, thus this implies a solution to counting of CNF.[2]

3.1 Karp, Luby, Madras, J. Alg. '89 [3]

Let n be the number of variables and m be the number of clauses. First, observe that randomly sample from the 2^n possible assignments to estimate the fraction p of satisfying assignments is not a good idea. This is because p can be very small, for which case we would need too many samples (thus won't be fast!) to get an good estimate of p. One such example is $\phi = x_1 \wedge x_2 \wedge \cdots \wedge x_n$, for which $p = 1/2^n$.

[1] The name came from counting, thus "#"

[2] For those who believe BPP (the randomized version of P)=P, they may also believe that there exists a deterministic algorithm for DNF counting. However, algorithms that are known so far are randomized.

Approach The idea is to reduce the size of the set from which we are sampling, in which case the fraction would not be too small.

- Let S_i be the set of satisfying assignments to C_i. The size of each such set is

$$|S_i| = 2^{n-\#\text{ of literals in } C_i}$$

- Let $B = \cup_{i=1}^{m} S_i$. We would like to estimate the size of $|B|$.
- Let B' be the set of pairs $\{(i, x) : x \in S_i\}$, we know $|B'| = \sum_i |S_i|$.
- We want to estimate $p' = |B|/|B'|$ up to $1 \pm \epsilon$, then output $p' \cdot |B'|$.

Bijection between subset of B' to B Note that the elements in B and the elements in B' are different stuff (assignments vs. pairs.) What to do?

- Treat $b \in B'$ as being in B if $b = (i, x)$ and C_i is the first clause x satisfies. This gives a natural bijection, and each satisfying assignment is only counted once.
- Now, p' cannot be so small:

$$p' \geq 1/m,$$

  since each satisfying assignment appears at most m times. We just need to argue that the empirical estimation of p' is close to the true probability.

Analysis

- Sample T times from B', and estimate p' as the fraction of samples that were in B (the subset that we put in correspondence with B.)
- Let X_i be the indicator random variable

$$X_i = \begin{cases} 1, & \text{if the ith smaple belongs to B} \\ 0, & \text{otherwise} \end{cases}$$

  We bound the probability by Chernoff bound:

$$\mathbb{P}\left[\left|\sum_{i=1}^{T} X_i - \mu\right| > \epsilon\mu\right] \leq e^{-c\epsilon^2 p' T}$$

  which is smaller than δ if we pick $T \geq C \lg(1/\delta) m/\epsilon^2$.

Sampling from B' How do we randomly draw an element from B'? Note that $|B'| = \sum |S_i|$.

Step 1. Pick i with probability $|S_i|/(\sum_j |S_j|)$

Step 2. Pick random $x \in S_i$ — the variables that appear in the clause has to be fixed, and those that do not appear in the clause can be picked uniformly at random.

4 Semidefinite Programming

We are going to design approximation algorithm for MaxCut via SDP in the next lecture.

MaxCut given an undirected graph $G = (V, E)$, find a cut with the maximum of edges crossing it: find $S \subset V$, $S \neq \emptyset$ s.t. $E(S, V \backslash S)$ is maximized. MaxCut is known to be an NP-hard problem, but have a trivial 2-approximation: pick a random set $S \subset V$. Since each edge is cut with probability half, and OPT cannot be larger than $|E|$,

$$\mathbb{E}(\#\text{ edges cut}) = \sum_{e \in E} \mathbb{P}[e \text{ is cut}] = \frac{|E|}{2} \geq \frac{\text{OPT}}{2}.$$

0.878-approx for MaxCut Goemans, Williamson '95 [4] propose algorithm achieving $\geq 0.878\text{OPT}$ in polynomial time, where $0.878 = \inf_{\theta \in [0,\pi]} \left\{ \frac{2}{\pi} \frac{\theta}{1-\cos\theta} \right\}$. Somehow this number is the *truth*.

Semidefinite Programming (SDP) In linear programming, x is a vector. In SDP, x is a matrix:

$$\begin{aligned} \min\ & \operatorname{tr}(C^\intercal X) \\ \text{s.t. } & \operatorname{tr}(A_i^\intercal X = b_i),\ \forall i = 1, \ldots, m, \\ & X \succeq 0. \end{aligned}$$

The trace of a matrix is defined as: $\operatorname{tr}(A^\intercal B) = \sum_{i,j} A_{i,j} B_{i,j}$— the objective doesn't look too much different from that in LP, so how is this more "matrixy"?

What's more important is the constraint $X \succeq 0$, i.e. X is *positive semi-definite* (PSD). A matrix X is PSD means

$$\forall z \in \mathbb{R}^n,\ z^\intercal X z \geq 0$$

We will see next time that LP is a special case of this (when everything is diagonal). We can also express the PSD constraints as

$$\forall z,\ \operatorname{tr}(X^\intercal (z z^\intercal)) \geq 0$$

i.e. infinitely many linear constraints. Just as we can relax ILP to LP, we can relax quadratic programs to SDPs, solve them, then round to get solutions to the quadratic program with provably good approximation ratios.

References

[1] Richard Karp, Michael Luby, Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *J. Algorithms*, 10(3):429–448, 1989.

[2] Eugene Lawler. Fast Approximation Algorithms for Knapsack Problems. *Math. Oper. Res.*, 4(4):339–356, 1979.

[3] Richard Karp, Michael Luby, and Neal Madras. Monte-Carlo Approximation Algorithms for Enumeration Problems. *J. Algorithms*, 10(3):429–448, 1989.

[4] Michel X. Goemans and David P. Williamson Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming *J. ACM*, 42(6):1115–1145, 1995.