CS264: Beyond Worst-Case Analysis Lecture #18: Smoothed Complexity and Pseudopolynomial-Time Algorithms[*]

Tim Roughgarden[†]

March 9, 2017

1 Preamble

Our first lecture on smoothed analysis sought a better theoretical understanding of the empirical performance of specific algorithms (the simplex method and local search). In our second (and last) lecture on the topic, we use the lens of smoothed analysis to understand the complexity of *problems*. We ask the question: which problems admit algorithms that run in smoothed polynomial time? There is a surprisingly crisp answer to this question — essentially the problems that are solvable in worst-case pseudopolynomial time.[1]

2 Binary Optimization Problems

To make the connection between smoothed polynomial-time algorithms and pseudopolynomial-time algorithms, we focus on the rich class of *binary optimization problems*. The Knapsack problem is a special case, as are many others. An instance of such a problem involves n 0-1 variables $x_1, x_2, \ldots, x_n$ (e.g., $x_i = 1$ if and only if item i is placed in the knapsack). There is abstract set of *feasible* solutions, which is just some subset F of vectors in $\{0,1\}^n$. (E.g., for some weight vector $\mathbf{w}$ and capacity W, we could define F as the vectors $\mathbf{x}$ that satisfy

[†] Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1] Recall that the running time of an algorithm is pseudopolynomial if it runs in time polynomial in the size of the input and the magnitudes of the numbers involved. Equivalently, the running time is allowed to be exponential in the binary representation of the numbers in the input. For example, the canonical dynamic programming algorithm for the Knapsack problem with integer item weights runs in time $O(nW)$, where W is the knapsack capacity. The natural encoding length — the number of keystrokes necessary to describe W — is $\approx \log W$, and the running time is exponential in this quantity.

$\sum_{i=1}^{n} w_i x_i \leq W$.) Finally, there are nonnegative values $v_1, v_2, \ldots, v_n$. The goal is to compute the feasible solution $\mathbf{x} \in F$ that maximizes the total value $\mathbf{v}^T\mathbf{x} = \sum_{i=1}^{n} v_i x_i$.[2]

The Knapsack problem, which is NP-hard but can be solved in pseudopolynomial time (by dynamic programming), is clearly a binary optimization problem. In general, the worst-case complexity of such a problem depends on the feasible set F. For example, if F is simply the vectors $\mathbf{x}$ with at most k 1's, or the characteristic vectors of the spanning trees of a fixed graph, then the problem is easy to solve in polynomial time. But problems much harder than the Knapsack problem, such as the maximum-weight independent set problem, can also be easily encoded as binary optimization problems.

If you only keep one concrete problem in mind as we go through this lecture, Knapsack would be a good one. If you want a second such problem, consider the following scheduling problem, which is likely all too familiar from real life. Given are n jobs, each with a known processing time p_j, deadline d_j, and integral cost c_j. The goal is to sequence the jobs on a single machine to minimize the total cost of the jobs not completed by their respective deadlines. This problem is NP-hard but can be solved in pseudopolynomial time via dynamic programming (see Homework #9). The positive results of this lecture apply directly to this scheduling problem (when the c_j's are perturbed).

3 Main Result

The goal of this lecture is to characterize the binary optimization problems that have polynomial smoothed complexity. We use exactly the same perturbation model as in last lecture, with each value v_i drawn independently according to a "not too spiky" density function $f_i : [0,1] \to \frac{1}{\sigma}$, where σ is a parameter that intuitively measures the "size of the perturbation," or the "amount of randomness" contained in each v_i. Note that when working with a smoothed instance, we assume that the values lie in $[0,1]$; this is more or less without loss of generality, by a scaling argument. Recall that a problem has polynomial smoothed complexity if there is an algorithm that can solve the problem with expected running time polynomial in the input size and in $\frac{1}{\sigma}$. Recall that a pseudopolynomial-time algorithm runs in time polynomial in the input size and in 2^b, where b is the maximum number of bits used to represent any number in the input.

Theorem 3.1 ([2]) *A binary optimization problem has smoothed polynomial complexity if and only if it can be solved by a (randomized) algorithm with pseudopolynomial expected running time.*

When we introduced smoothed analysis, for example to analyze the running time of the simplex method, we saw that moving from a worst-case input model to a smoothed input model can drop the running time of an algorithm from slow (exponential time) to fast (polynomial) time. Theorem 3.1 shows that moving from worst-case to smoothed analysis can

[2] Everything we say today applies also to minimization problems.

drop the computational complexity of a problem from hard (meaning there is no polynomial-time solution, assuming $P \neq NP$) to easy (meaning there is a (smooth) polynomial-time solution). The theorem also nicely corroborates real-world experience with NP-complete problems — those with pseudopolynomial solutions are generally the empirically easiest of the bunch. It is very cool that smoothed analysis yields a novel reformulation of an age-old concept like pseudopolynomial-time solvability.[3]

4 Proof of "Only If" Direction

To prove the "only if" direction of Theorem 3.1, consider a binary optimization problem that admits an algorithm A with smoothed polynomial complexity. We need to show that this problem also admits a (worst-case over inputs) expected pseudopolynomial time algorithm, presumably by invoking A as a black box.

Consider a worst-case instance I, specified by arbitrary rational values $v_1, v_2, \ldots, v_n$. Clearing denominators if necessary, we can assume that all of the v_i's are integral. Then, if two different feasible solutions to I have different objective function values, this difference is at least 1. Now let $v_{max} = \max_{i=1}^{n} v_i$ and scale all the values by v_{max} so that they lie in $[0, 1]$. After scaling, distinct objective function values differ by at least $\frac{1}{v_{max}}$.

Our new algorithm has two simple steps.

1. Perturb (i.e., add to) each v_i by a random amount δ_i, chosen uniformly at random from the interval $(0, 1/nv_{max})$.

2. Run the algorithm A on the perturbed instance, and return the resulting feasible solution.

Previously, we always regarded perturbations as an assumption about inputs. In the above algorithm, the input is arbitrary, and we're making up fictitious perturbations to ensure that the given algorithm A runs quickly![4]

Next we discuss the correctness of the algorithm. The perturbations add a number between 0 and $1/nv_{max}$ to each v_i, and hence a number between 0 and $1/v_{max}$ to each feasible solution (since there are only n v_i's). In terms of Figure 1, every feasible solution moves upward in objective function value, but by less than $1/v_{max}$. Prior to the perturbations, distinct objective function values differed by at least $1/v_{max}$. This means that, with probability 1, the optimal solution after the perturbations was also an optimal solution prior to the perturbation, and is safe to return.[5]

For the running time, observe that every perturbed value v_i fed into algorithm A is distributed uniformly on an interval of length $1/nv_{max}$, which corresponds to a value of

[3] These two concepts are also closely connected to the existence of a "fully polynomial-time approximation scheme (FPTAS)," meaning an algorithm that achieves an approximation guarantee of $1 \pm \epsilon$ in time polynomial in the input size and in $\frac{1}{\epsilon}$. See Homework #9 for more details.

[4] See [4, 6] for two other algorithmic applications of similar fictitious perturbations.

[5] Note that if the original instance had multiple optimal solutions, the algorithm winds up returning the one that gets perturbed the most.

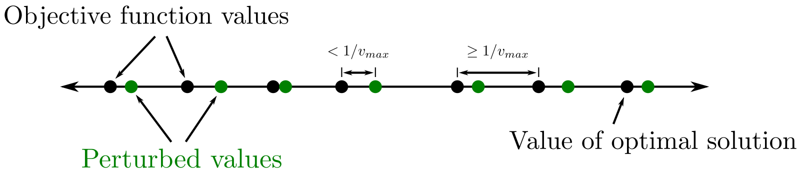

Figure 1: Distinct objective function values are separated by at least $1/v_{max}$, and the perturbations increase each objective function value by less than $1/v_{max}$. The optimal solution remains unchanged after the perturbation.

$\frac{1}{\sigma} = nv_{max}$. By assumption, A has expected (over the perturbation) running time polynomial in n and $\frac{1}{\sigma} = nv_{max}$. Thus, for every input, our two-step algorithm has expected running time polynomial in n and v_{max}, as desired.

Remark 4.1 Why should we care about this direction of Theorem 3.1? The obvious reason is the satisfaction of having an "if and only if" characterization of the problems solvable in smoothed polynomial time. In principle, one could use the "only if" direction to design novel pseudopolynomial-time algorithms, by instead designing an algorithm with smoothed polynomial complexity. It is not clear this is useful, since it seems easier to design pseudopolynomial-time algorithms than smoothed polynomial-time algorithms (think of Knapsack, for example). In its contrapositive form, however, the "only if" direction is definitely interesting: if a problem does *not* admit a (randomized) pseudopolynomial-time algorithm, then it does *not* have smoothed polynomial complexity. In particular, strongly NP-hard problems (see e.g. [5]) do not have smoothed polynomial complexity, under appropriate complexity assumptions. While weakly NP-hard problems like Knapsack go from "hard" to "easy" when passing from worst-case to smoothed instances, most NP-hard problems do not. See also Homework #9.

5 Proof of "If" Direction

The rest of this lecture proves the converse. Here, we consider a problem that admits a worst-case pseudopolynomial-time algorithm A, and our job is to construct an algorithm with expected polynomial running time on smoothed instances. Again, since we know nothing about the problem other than the fact that A exists, we'll presumably proceed by invoking A as a black box (perhaps multiple times). Unlike the previous lecture, we're not using smoothed analysis to explain the performance of an existing algorithm — we're using the goal of polynomial smoothed complexity as a guide to designing a new algorithm.

5.1 The Isolation Lemma

The key tool in the proof is an "Isolation Lemma." This part of the proof applies to every binary optimization problem, not just those that admit pseudopolynomial-time algorithms.

Fix a smoothed instance of a binary optimization problem, with every value v_i drawn from a distribution with density function bounded above by $\frac{1}{\sigma}$. For a parameter $\epsilon > 0$, call the variable x_i *ϵ-bad* if the optimal solution subject to $x_i = 0$ and the optimal solution subject to $x_i = 1$ have objective function values within ϵ of each other.

We claim that, for every $\epsilon > 0$ and $i \in \{1, 2, \ldots, n\}$, the probability that x_i is ϵ-bad is at most $2\epsilon/\sigma$. We show that this is true even after conditioning on the value of v_j for every $j \neq i$ — v_i alone has sufficient randomness to imply the upper bound. Let S_{-i} and S_{+i} denote the optimal solutions with $x_i = 0$ and $x_i = 1$, respectively.[6] Since S_{-i} involves only feasible solutions with $x_i = 0$, both it and its value V_{-i} are fixed after conditioning on v_j for every $j \neq i$. Since v_i contributes the same amount to every feasible solution with $x_i = 1$, the identity of S_{+i} is also fixed — it is simply the feasible solution with $x_i = 1$ that maximizes $\sum_{j \neq i} v_j x_j$. The value of S_{+i} can be written as $Z + v_i$, where Z is the (fixed) sum of the v_j's with $x_j = 1$ in S_{+i}. The variable x_i is ϵ-bad if and only if the objective function values of S_{+i} and S_{-i} are within ϵ of each other, which occurs if and only if $v_i = V_{-i} - Z \pm \epsilon$. By assumption, v_i's distribution has density at most $\frac{1}{\sigma}$ everywhere, so the probability that v_i lands in this bad interval is at most $2\epsilon/\sigma$. This completes the proof of the claim.

We now come to the formal statement of the isolation lemma. Let $V^{(1)}$ and $V^{(2)}$ denote the highest and second-highest objective function values of a feasible solution. Then

$$\mathbf{Pr}\left[\underbrace{V^{(1)} - V^{(2)}}_{\text{“winner gap”}} \leq \epsilon\right] \leq \frac{2\epsilon n}{\sigma}. \tag{1}$$

In proof, let $S^{(1)}$ and $S^{(2)}$ denote the best and second-best feasible solutions (with objective function values $V^{(1)}$ and $V^{(2)}$, respectively). Since $S^{(1)} \neq S^{(2)}$, there is some variable x_i that is set to 0 in exactly one of these two feasible solutions. Say $x_i = 0$ in $S^{(1)}$ and $x_i = 1$ in $S^{(2)}$; the other case is symmetric. Since $S^{(1)}$ is the maximum-value feasible solution, it is certainly the maximum-value feasible solution that also satisfies $x_i = 0$. Since $S^{(2)}$ is the maximum-value feasible solution different from $S^{(1)}$, it is certainly the maximum-value feasible solution that also satisfies $x_i = 1$. Thus, if $V^{(1)} - V^{(2)} < \epsilon$, then x_i is ϵ-bad. It follows that the probability that the winner gap is less than ϵ is at most the probability that there is an ϵ-bad variable. By the claim above and a Union Bound over the n variables, this probability is at most $2\epsilon n/\sigma$, as claimed.

5.2 Completing the Proof of Theorem 3.1

We now return our attention to a binary optimization problem that admits a pseudopolynomial-time algorithm A.[7] Given a smoothed instance, with random v_i's in $[0, 1]$, we execute the following algorithm (left underdetermined for now):

[6] If $x_i = 1$ in every feasible solution, or $x_i = 0$ in every feasible solution, then x_i cannot be ϵ-bad. So, we can assume that S_{-i} and S_{+i} are well defined.

[7] For simplicity, we assume that A is deterministic. Essentially the same argument works if A is randomized, with expected pseudopolynomial running time.

1. For $b = 1, 2, 3, \ldots$:
   (a) Let v_i^b denote the most significant b bits of v_i.[8]
   (b) Invoke algorithm A to compute the optimal solution $\mathbf{x}^b$ for the values $\mathbf{v}^b$.
   (c) (Detailed further below) If $\mathbf{x}^b$ is optimal for $\mathbf{v}$ for every setting of the unseen bits of the v_i's, halt and output $\mathbf{x}^b$.

Because A is a pseudopolynomial algorithm (recall Section 1), the running time of step (b) is polynomial in n and 2^b. If we implement step (c) correctly and the algorithm halts, then it is clear that it halts with an optimal solution.

Intuitively, in the third step, we want to implement the following thought experiment: would $\mathbf{x}^b$ remain optimal even under a "worst-case" unveiling of the remaining bits of the v_i's? From $\mathbf{x}^b$'s perspective, the worst case occurs when all remaining bits are 0 for the values v_i with $x_i = 1$, and all remaining bits are 1 for the values v_i with $x_i = 0$. These remaining bits minimize the reward to $\mathbf{x}^b$ for each variable set to 1, and maximize the penalty for each variable not set to 1. If $\mathbf{x}^b$ remains optimal in this case, then it is optimal no matter what the true values $\mathbf{v}$ are.

More formally, we implement step (c) as follows. We define $\bar{v}_i^b = v_i^b$ whenever $x_i = 1$ and $\bar{v}_i^b = v_i^b + 2^{-b}$ whenever $x_i = 0$. These $\bar{v}^b$ values are even worse for $\mathbf{x}^b$ than the worst case of the previous paragraph, and have the added advantage of being at most $b + 1$ bits long. Thus, we can check if $\mathbf{x}^b$ is optimal for $\bar{\mathbf{v}}^b$ with one further invocation of A, in time polynomial in n and 2^b. If it is, then we can safely halt and return $\mathbf{x}^b$, since it must be optimal for the true values $\mathbf{v}$ as well.

We've argued that the algorithm is correct; what is its running time? The answer depends on how big b needs to get before the stopping condition gets triggered. Since A runs in time exponential in b, we're hoping that a logarithmic number of bits (in all parameters of interest) is enough.

To begin the analysis, note that, by definition,

$$v_i^b \in [v_i - 2^{-b}, v_i]$$

and

$$\bar{v}_i^b \in [v_i - 2^{-b}, v_i + 2^{-b}]$$

for each $i = 1, 2, \ldots, n$. Since there are only n coordinates, for every feasible solution $\mathbf{x}$,

$$\sum_{i=1}^n v_i^b x_i \text{ and } \sum_{i=1}^n \bar{v}_i^b x_i \in \left[\sum_{i=1}^n v_i x_i - n2^{-b}, \sum_{i=1}^n v_i x_i + n2^{-b} \right].$$

In terms of Figure 2, switching the objective function from $\mathbf{v}$ to either $\mathbf{v}^b$ or $\bar{\mathbf{v}}^b$ moves the objective function value of every feasible solution up or down by at most $n2^{-b}$. Thus, if $\mathbf{x}^*$

[8] Since $v_i \in [0, 1]$, the first bit is "1" if and only if $v_i \geq \frac{1}{2}$, the second bit is "1" if and only if $v_i \in [\frac{1}{4}, \frac{1}{2})$ or $v_i \geq \frac{3}{4}$, and so on.

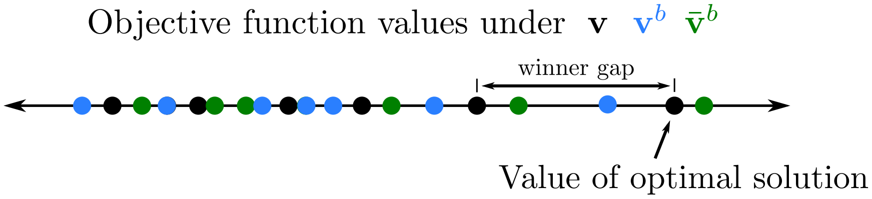
Objective function values under $\mathbf{v}$ $\mathbf{v}^b$ $\bar{\mathbf{v}}^b$
winner gap
Value of optimal solution

Figure 2: As b increases the blue and green points get closer to their respective black points. Eventually the optimal solution under $\mathbf{v}^b$ and $\bar{\mathbf{v}}^b$ is the same, and the algorithm terminates.

is the optimal solution for the objective function $\mathbf{v}$ and the winner gap is more than $2n2^{-b}$, then $\mathbf{x}^*$ is also the optimal solution for the objective functions $\mathbf{v}^b$ and $\bar{\mathbf{v}}^b$. We conclude that the algorithm terminates by the time b is so large that the winner gap exceeds $2n2^{-b}$, if not earlier. How long does this take?

Using inequality (1) in the isolation lemma, with $\epsilon = 2n2^{-b}$, we get that the probability that the winner gap is at most $2n2^{-b}$ is at most $2n^2 2^{-b}/\sigma$. (All probabilities are over the randomness in the v_i's.) For every $\delta > 0$, this is at most δ once $b = \Theta(\log \frac{n}{\sigma\delta})$. Since the running time of the algorithm above is polynomial in n and 2^b, we obtain:

> For every $\delta > 0$, the probability that the algorithm terminates in time polynomial in n, $\frac{1}{\sigma}$, and $\frac{1}{\delta}$, is at least $1 - \delta$.

For example, taking $\delta = \frac{1}{n}$, we find that the algorithm runs in smoothed polynomial time with high probability. This is a bit weaker than our usual definition of smoothed polynomial complexity, which insists that the expected running time of the algorithm is polynomial in the input size and in $\frac{1}{\sigma}$, but it is still a pretty satisfying positive result.

6 Final Comments

The analysis in Section 5 considered a smoothed objective function. The analysis of this lecture can be extended, using more involved versions of the same arguments, to smoothed constraints [2]. The setup is that, in addition to an arbitrary feasible set F, there is one (or even a constant number) of constraints of the form $\sum_{i=1}^{n} w_i x_i \leq W$, with the w_i's chosen independently from distributions with density functions bounded above by $\frac{1}{\sigma}$. It is again true that the existence of a worst-case randomized pseudopolynomial-time algorithm implies the existence of a smoothed polynomial-time algorithm for the problem.

A key step in the proof of Theorem 3.1 was to show that, with high probability, the winner gap is not too small (1). This turns out to be useful also in the analysis of "message-passing" algorithms such as "Belief Propagation." For many problems, such as maximum bipartite matching, these algorithms perform poorly in the worst case but well in practice. One explanation for this is: the running time of such algorithms can often be parameterized by the winner gap, and run quickly whenever the winner gap is large (e.g. [1]). As shown in this lecture, most (smoothed) instances have a reasonably large winner gap. See [3] for further discussion.

References

[1] M. Bayati, D. Shah, and M. Sharma. Max-product for maximum weight matching: Convergence, correctness, and lp duality. *IEEE Transactions on Information Theory*, 54(3):1241–1251, 2008.

[2] René Beier and Berthold Vöcking. Typical properties of winners and losers in discrete optimization. *SIAM Journal on Computing*, 35(4):855–881, 2006.

[3] T. Brunsch, K. Cornelissen, B. Manthey, and H. Röglin. Smoothed analysis of belief propagation for minimum-cost flow and matching. *Journal of Graph Algorithms and Applications*, 17(6):647–670, 2013.

[4] S. Dughmi and T. Roughgarden. Black-box randomized reductions in algorithmic mechanism design. *SIAM Journal on Computing*, 43(1):312–336, 2014.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[6] J. A. Kelner and D. A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 51–60, 2006.