Is sos an "optimal algorithm"?

We have alluded several times in this course to the intuition that sum of squares might be an "optimal algorithm" in some settings, but why would we think that? and what does this even mean? In this lecture we explore these questions:

- What does being an "optimal algorithm" mean?
- Why should we expect an optimal algorithm to exist?
- Even if there is *some* optimal algorithm, why should it be sos?

Optimal algorithms for optimization problems.

Let's start with the first question. There are several ways to define optimality, and we will choose one variant for concreteness. In particular, let us assume that we are dealing with NP maximization problems. That is, problems of the following type:

- **Input:** Function $f\colon \{0,1\}^n \to [0,1]$ in some explicit form that allows us to evaluate it.
- **Goal:** Find $x \in \{0,1\}^n$ that maximizes $f(x)$.

  We define $opt(f) = \max_{x\in\{0,1\}^n} f(x)$.

This is a pretty general formalism. For example, in the *Max Cut* problem, the function f is the one that maps a cut (encoded as a string) to the fraction of edges that it cuts. In the small set expansion problem, given a graph $G = (V, E)$ we could define f so that $f(x) = 0$ if x does not encode a sparse set, and otherwise $f(x)$ equals $1 - |E(S, \overline{S})|/(d|S|)$ where S is the set encoded by x.

A *computational problem* can now be thought of as a set $\mathcal{F} = \cup_n \mathcal{F}_n$ where $\mathcal{F}_n$ is some subset of all functions from $\{0,1\}^n$ to $[0,1]$, while a *class of problems* $\mathcal{C}$ is a family of such problems. We will consider algorithms that take the function f (in some representation) and output an assignment $x \in \{0,1\}^n$. We are restricting ourselves to NP problems, and so assume that the representation of f has $\mathrm{poly}(n)$ size and that one can use it to evaluate f on an input x in $\mathrm{poly}(n)$ time.

1. Definition (Optimal algorithm). Let $T\colon \mathbb{N} \to \mathbb{N}$ and $\epsilon > 0$. We say that an algorithm A is (T, ϵ)-*optimal* for a problem $\mathcal{F}$ if for every every time $T(n)$ algorithm B

$$\overline{\lim_{n\to\infty}} \left[\max_{f\in\mathcal{F}_n} [opt(f) - A(f)] - \max_{f\in\mathcal{F}_n} [opt(f) - B(f)] \right] \leq \epsilon \qquad (1)$$

This is a notion of *worst case* approximate additive optimality. One can also consider multiplicative approximations, as well as notions such as *average case optimality* (where we would replace the maximum over $f \in \mathcal{F}$ in Eq. (1) with expectation over a random $f \in \mathcal{F} \cap [0,1]^{\{0,1\}^n}$ for arbitrarily large n). We can even consider *instance optimality* (where we would require the inquality to hold for every f *pointwise*) though one then has to be careful to rule out algorithms B that have the solution to a particular instance "hardwired" inside them.

Thus, while this is not the be all and end all notion of optimality, Definition 1 does seem like a reasonable starting point to explore it. Let's start by the following observation:

2. Theorem. *For every ("nice", efficiently computable) time complexity function $T\colon \mathbb{N} \to \mathbb{N}$, there exists an algorithm A that runs in time $T(n)\,\mathrm{poly}(n)$ and is (T,ϵ) optimal with respect to any NP problem $\mathcal{F}$.*

Proof. On input f, the algorithm A will enumerate the first n Turing machines $M_1,\ldots,M_n$ and run each one of them for $T(n)$ steps to obtain $x_1 = M_1(f),\ldots,x_n = M_n(f)$. It then outputs the x_i that maximizes $f(x_i)$. We leave the analysis of this as an **exercise**. □

So, there an optimal algorithm *exists* but this kind of "diagonalization based" algorithm is not very satisfying. So the real question is whether there is a **"simple"** or **"nice"** optimal algorithm, where one can think of several ways to define what "nice" means:

- We know it when we see it.
- Doesn't use diagonalization.
- A concrete enough algorithm that we can prove unconditional lower bounds for it.
- Ties in to other mathematical notions such as convexity etc..

Why should a nice optimal algorithm exist?

Should nice optimal algorithms exist? We don't really now. But it is a very important question. At the heart of it is whether we can *understand* the computational difficulty of problems, in the sense of having clean criteria (i.e., the performance of a nice optimal algorithm) that separates the easy problems from the hard ones, and allows us to make concrete predictions on which problems would be easy or hard. It can also in principle, reduce the task of designing algorithm, which

now requires significant creativity and "ad hoc tailoring" to each problem, into a more systematic enterprise. This is similar to the way that equation solving in mathematics evolved from a challenge required the creative genius of people like Leibnitz, Euler, Gauss, etc.. to solve individual equations to a calculation that is now routinely and automatically done by computer programs.

There are some reasons to be hopeful for such a nice optimal algorithm, at least in some restricted domains such as combinatorial optimization:

- The same algorithmic ideas, including notions such as *greedy algorithms*, *divide and conquer*, *convex relaxations* , keep recurring in algorithm design. This is particularly true for combinatorial optimization as opposed to say, algebraic algorithms (e.g., integer factorization) where ad-hoc tricks using algebraic identities are more prevalent.
- In practice *general purpose* software packages for optimization are widely used for a host of different problems, rather than ad hoc programs for a particular application.

This seems consistent with the assumption that at least for optimization there are a few underlying basic ideas that suffice to distinguish between those problems that are inherently unsolvable and those that can be solved efficiently, while clever ad hoc tricks are useful for either gaining (often very important!) second order improvements, or facilitating the *analysis*.

Why sos?

So, one might hope that there is a nice optimal algorithm, and perhaps even one based on convex optimization, but why sum of squares? Once again, we don't really know, and there are certain algorithmic frameworks (such as hyperbolic programming) that could perhaps offer stronger power. But we can try to get some evidence for optimality of sos. There are generally two kinds of potential evidence:

- There are some results *proving* that sos is optimal algorithm in terms of *worst case approximation ratio*. Under the (somewhat controversial) Unique Games Conjecture, Raghavendra [2008] showed that sos (and in fact a very restricted special case of it) is an optimal worst-case approximation algorithm for every constraint satisfaction problems. Without assuming the UGC, the strongest

result along those lines is of Chan [2016] who showed optimality of sos for a restricted subset of CSP's.

- There are results showing that sos captures other convex programming techniques. It has been shown that sos is stronger than certain linear and semidefinite programming hierarchies that have been considered in the literature such as those of Sherali and Adams [1990] and Lovász and Schrijver [1991] .

A recent result of Lee et al. [2015] shows that sos is optimal for CSP's among *all* semidefinite programs of comparable size. This give powerful evidence that if sos is not the strongest convex optimization based efficient algorithm, one would have to go beyond semidefinite programming to beat it.[1]

[1] Formalizing such results is tricky, because both linear programming and semidefinite programming are P-complete, which means that if we do not restrict the way that we formalize a constraint satisfaction problem as an LP or SDP then we can encode an arbitrary polynomial-time computation using it and hence proving optimality of sos would in particular imply (via (Grigoriev [2001], Schoenebeck [2008])) that no polynomial time algorithm can solve 3SAT and hence that $P \neq NP$.

References

Siu On Chan. Approximation resistance from pairwise-independent subgroups. *J. ACM*, 63(3):27, 2016.

Dima Grigoriev. Linear lower bound on degrees of Positivstellensatz calculus proofs for the parity. *Theoret. Comput. Sci.*, 259(1-2):613–622, 2001. ISSN 0304-3975. doi: 10.1016/S0304-3975(00)00157-2. URL http://dx.doi.org/10.1016/S0304-3975(00)00157-2.

James R. Lee, Prasad Raghavendra, and David Steurer. Lower bounds on the size of semidefinite programming relaxations. In *STOC*, pages 567–576. ACM, 2015.

László Lovász and Alexander Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal on Optimization*, 1(2): 166–190, 1991.

Prasad Raghavendra. Optimal algorithms and inapproximability results for every csp? In *STOC*, pages 245–254. ACM, 2008.

Grant Schoenebeck. Linear level lasserre lower bounds for certain k-csps. In *FOCS*, pages 593–602. IEEE Computer Society, 2008.

Hanif D. Sherali and Warren P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM J. Discrete Math.*, 3(3):411–430, 1990. ISSN 0895-4801. doi: 10.1137/0403036. URL http://dx.doi.org/10.1137/0403036.