CS261: A Second Course in Algorithms
The Top 10 List*

Tim Roughgarden†

March 10, 2016

If you've kept up with this class, then you've learned a tremendous amount of material. You know now more about algorithms than most people who don't have a PhD in the field, and are well prepared to tackle more advanced courses in theoretical computer science. To recall how far you've traveled, let's wrap up with a course top 10 list.

1. *The max-flow/min-cut theorem*, and the corresponding polynomial-time algorithms for computing them (augmenting paths, push-relabel, etc.). This is the theorem that seduced your instructor into a career in algorithms. Who knew that objects as seemingly complex and practically useful as flows and cuts could be so beautifully characterized?

   This theorem also introduced the running question of "how do we know when we're done?" We proved that a maximum flow algorithm is done (i.e., can correctly terminate with the current flow) when the residual graph contains no s-t path or, equivalently, when the current flow saturates some s-t cut.

2. *Bipartite matching,* including the Hungarian algorithm for the minimum-cost perfect bipartite matching problem. In this algorithm, we convinced ourselves we were done by exhibiting a suitable dual solution (which at the time we called "vertex prices") certifying optimality.

3. *Linear programming is in P.* We didn't have time to go into the details of any linear programming algorithms, but just knowing this fact as a "black box" is already extremely powerful. On the theoretical side, there are polynomial-time algorithms for solving linear programs — even those whose constraints are specified implicitly through a polynomial-time separation oracle — and countless theorems rely on this fact. In practice, commercial linear program solvers routinely solve problem instances with millions of variables and constraints and are a crucial tool in many real-world applications.

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

4. *Linear programming duality.* For linear programming problems, there's a generic way to know when you're done. Whatever the optimal solution of the linear program is, strong LP duality guarantees that there's a dual solution that proves its optimality. While powerful and perhaps surprising, the proof of strong duality boils down to the highly intuitive statement that, given a closed convex set and a point not in the set, there's a hyperplane with the set on one side and the point on the other.

5. *Online algorithms.* It's easy to think of real-world situations where decisions need to be made before all of the relevant information is available. In online algorithms, the input arrives "online" in pieces, and an irrevocable decision must be made at each time step. For some problems, there are online algorithms with good (close to 1) competitive ratios — algorithms that compute a solution with objective function value close to that of the optimal solution. Such algorithms perform almost as well as if the entire input was known in advance. For example, in online bipartite matching, we achieved a competitive ratio of $1 - \frac{1}{e} \approx 63\%$ (which is the best possible).

6. *The multiplicative weights algorithm.* This simple online algorithm, in the spirit of "reinforcement learning," achieves per-time-step regret approaching 0 as the time horizon T approaches infinity. That is, the algorithm does almost as well as the best fixed action in hindsight. This result is interesting in its own right as a strategy for making decisions over time. It also has some surprising applications, such as a proof of the minimax theorem for zero-sum games (if both players randomize optimally, then it doesn't matter who goes first) and fast approximation algorithms for several problems (maximum flow, multicommodity flow, etc.).

7. *The Traveling Salesman Problem (TSP).* The TSP is a famous NP-hard problem with a long history, and several of the most notorious open problems in approximation algorithms concern different variants of the TSP. For the metric TSP, you now know the state-of-the-art — Christofides's $\frac{3}{2}$-approximation algorithm, which is nearly 40 years old. Most researchers believe that better approximation algorithms exist. (You also know close to the state-of-the-art for asymmetric TSP, where again it seems that better approximation algorithms should exist.)

8. *Linear programming and approximation algorithms.* Linear programs are useful not only for solving problems exactly in polynomial time, but also in the design and analysis of polynomial-time approximation algorithms for NP-hard optimization problems. In some cases, linear programming is used only in the analysis of an algorithm, and not explicitly in the algorithm itself. A good example is our analysis of the greedy set cover algorithm, where we used a feasible dual solution as a lower bound on the cost of an optimal set cover. In other applications, such as vertex cover and low-congestion routing, the approximation algorithm first explicitly solves an LP relaxation of the problem, and then "rounds" the resulting fractional solution into a near-optimal integral solution. Finally, some algorithms, like our primal-dual algorithm for vertex

cover, use linear programs to guide their decisions, without ever explicitly or exactly solving the linear programs.

9. *Five essential tools for the analysis of randomized algorithms.* And in particular, the Chernoff bounds, which prove sharp concentration around the expected value for random variables that are sums of bounded independent random variables. Chernoff bounds are used *all the time*. We saw an application in randomized rounding, leading to a $O(\log n / \log \log n)$-approximation algorithm for low-congestion routing.

   We also reviewed four easy-to-prove tools that you've probably seen before: linearity of expectation (which is trivial but super-useful), Markov's inequality (which is good for constant-probability bounds), Chebyshev's inequality (good for random variables with small variance), and the union bound (which is good for avoiding lots of low-probability events simultaneously).

10. *Beating brute-force search.* NP-hardness is not a death sentence — it just means that you need to make some compromises. In approximation algorithms, one insists on a polynomial running time and compromises on correctness (i.e., on exact optimality). But one can also insist on correctness, resigning oneself to an exponential running time (but still as fast as possible). We saw three examples of NP-hard problems that admit exact algorithms that are significantly faster than brute-force search: the unweighted vertex cover problem (an example of a "fixed-parameter tractable" algorithm, with running time of the form $\text{poly}(n) + f(k)$ rather than $O(n^k)$); TSP (where dynamic programming reduces the running time from roughly $O(n!)$ to roughly $O(2^n)$); and 3SAT (where random search reduces the running time from roughly $O(2^n)$ to roughly $O((4/3)^n)$).