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

Tim Roughgarden†

June 2, 2015

If you've kept up with this class, then you've learned a tremendous amount of material. 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. Before knowing this result, the spaces of flows and of cuts seem so complex — afterward, they are things of beauty, and with lots of concrete applications.

   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").

3. *Edmonds's algorithm* for maximum matching in unweighted general (non-bipartite) graphs. This year marks the 50th anniversary of the publication of this algorithm. Despite its status as one of the classics, few computer scientists have taken the time to learn the algorithm as thoroughly as you now have! It is far from obvious that the maximum matching problem is polynomial-time solvable, and when studying the problem there's a palpable sense of venturing into the outer territories of P, with NP-complete problems lurking in all directions. How did we know we were done? By the Tutte-Berge formula, which sensibly lower bounds the number of vertices exposed in every matching as $\max_{S \subseteq V}(\text{oc}(S) - |S|)$.

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

4. *Linear programming is in P.* We had little time to discuss the specifics of any algorithms for solving linear programs, 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.

5. *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.

6. *The Traveling Salesman Problem (TSP).* The TSP is a famous 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.

7. *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.

8. *Hardness of approximation.* In addition to the rich theory about what polynomial-time approximation algorithms can do, there's also a mature understanding of what such algorithms *can't* do. Almost all of this understanding has been developed in the last 25 years, much of it in this century. For example, $P \neq NP$ not only implies that the Set Cover problem can't be solved exactly, it turns out to also imply that there is no approximation algorithm with worst-case approximation ratio superior to that of the greedy algorithm. Such hardness of approximation results are proved using a minor variant of the NP-completeness reductions that you already know and love — one just adapts the idea from decision to optimization problems, and keeps track of the "gap" between the objective function values that correspond to "yes" and "no" instance of (say) SAT. (That said, the state-of-the-art reductions tend to be highly technical.)

9. *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).

10. *Streaming algorithms.* In the data stream model, the input again arrives online, one element at a time (like data packets at a network switch). The goal is to do useful computations despite having far too little space to remember the input. As a case study, we generalized the one-pass majority element algorithm to solve the highly practical ϵ-heavy hitters problem.