# CSE 5401 Course Review

Our starting point was a recap of undergraduate algorithms techniques for designing optimal, polynomial-time algorithms namely, ① Greedy - e.g., Prim & Kruskal's algorithms for Minimum Spanning Tree. Optimality proved by "exchange argument" — at the first point where greedy chooses a different [edge] from an alternative solution, argue that we can swap in the greedy choice and obtain a solution no worse

② Dynamic Programming - e.g., pseudopolynomial-time algorithm for Knapsack. We store (or pre-compute) the solutions to subproblems for a recursive algorithm, bounding the number of subproblems by a poly. in the input size. Optimality proved by induction on some problem size parameter.

We recalled that we can show that most of the problems we want to solve don't have poly-time optimal algorithms by proving that the problems are NP-complete: every NP complete problem can be solved in polynomial time if we have a function that returns optimal solutions to our problem. (It's actually enough to just show that one NP-complete problem can be solved in this way.)

Thus, we focused on the design of algorithms that find approximately optimal solutions. We saw three families of algorithms

1. Primal-Dual Algorithms. Starting from a discrete/integer

optimization problem with linear inequality constraints, we "relaxed" the integrality constraint to obtain a linear program. We then iteratively improve a feasible solution to the dual until we find some tight constraints, that give a feasible integer solution to the original problem. By relating the costs to the feasible dual solution we construct, we got an approximation bound using weak duality.
The primal-dual schema allows us to design algorithms for on-line optimization problems, where new constraints are iteratively introduced, and we can only add to our previously-chosen solution. We still compare to the optimal ("hindsight") solution.
We developed our understanding of dual linear programs by relating them to zero sum games. The min-max theorem (either player can commit to a randomized choice of play first, and the optimal payoffs are the same) corresponds to "strong duality" (if both primal and dual LPs are feasible, the optimal values coincide). We can view the primal + dual LPs as a game, feasible solutions vs. constraints. Only tight constraints are used in optimal play ("complementary slackness").
We proved the min-max theorem originally using the Multiplicative Weights method. We can now, with hindsight, view this as a fast method for identifying a single dual strategy that we respond to. Note that it was enough to sample a strategy even if there is an exponentially large set of

strategies (e.g., the path-selection problem) and the number of rounds needed was logarithmic in the dimension. Multiplicative Weights solved an on-line learning problem, where we want our cost/score to converge to that attained by the optimal single strategy in hindsight ("regret → 0"). The guarantee holds even if the "opponent's" plays are chosen adversarially.

2. Randomized Rounding of Semidefinite Programs.

Again, starting from a discrete optimization problem, we relax to a vector program — products of variables become inner products — which we round to an integer solution by choosing a random direction and checking if the vector has a positive or negative projection. For problems of the form, "maximize the # of satisfied constraints," where each constraint has only a small # of variables (such as Max-Cut, 2-SAT, ...) these algorithms are conjectured to be optimal.

The vector optimization problem is captured by positive-semidefiniteness constraints because we can compute Cholesky factorizations of the PSD matrix variables. We were also able to compute a separation oracle for the PSD constraints while trying to compute a Cholesky factorization, and thus we could use the framework of convex optimization with separation oracles (e.g., Ellipsoid) to solve the problems. More generally, separation oracles allow us to solve convex optimization problems to high precision when there are many constraints.

(We can also solve linear programs exactly by rounding these solutions to nearby rational values.)

An alternative method for convex optimization that obtains less precise solutions, but in time independent of the dimension is gradient descent: iteratively move in the direction of a (sub) gradient of the objective. For "smooth" and "strongly convex" objectives (quadratic term is $\Theta(1)$) gradient descent also obtains accurate solutions.

3. Markov Chain Monte Carlo. For problems where we wish to count the number of solutions (e.g. colorings, independent sets, etc.) we can obtain arbitrarily accurate approximations as long as we can sample from solutions with arbitrarily close to uniform probability. We can obtain samples efficiently by constructing a Markov chain on solutions with a uniform stationary distribution and bounding the number of steps before the distribution is close enough.

- To get a uniform stationary distribution, we saw that it's enough for the transition probabilities between pairs of states to be symmetric, for all states to be reachable, and for there to always be a chance of remaining in the same state.
- To bound the number of steps, it was enough to bound the number of steps before the states in a coupling meet. Using path coupling, it was enough to just analyze adjacent pairs of states.

Alternatively, the time to reach the stationary distribution is determined by the conductance of the transition graph. We also saw how the eigenvector can be used to find sparse (low-

conductance) cuts by choosing an appropriate threshold, and how other key characteristics (periodicity, connectedness, etc.) of the Markov chain/graph appear in the eigenvalues.

In addition, we studied techniques for "big data" —

- Locality-sensitive hashing for finding near-matches and faster nearest-neighbor search
- Random linear maps of $\log n$ dimension approximately preserve inner products/distances among $n$ points of any dim.
- Algorithms for finding frequent elements, estimating # distinct elements, etc. in <u>data streams</u>: we read the input once, and maintain a data structure of sublinear size/in sublinear time per element. Primarily, we used hashing and linear maps ("sketches") to solve these problems

Back to hashing, we saw how pairwise independence enables a Chebyshev-style analysis of collisions, and $O(\log n)$-wise independence suffices to get polynomially small failure probability. We saw that simply using two hash functions reduces the maximum load in a linear size table from $\log n$ to $\log\log n$.

In an orthogonal direction, we saw that noise in the data (even for an otherwise adversarial input) can sometimes lead simple algorithms with poor worst-case running times to run fast in practice (e.g., k-means). In other cases, the algorithms may also obtain good solutions w.h.p. over the noise.

Finally, we saw that our pseudopolynomial-time algorithms

could be used to obtain arbitrarily accurate approxi-
mations in polynomial time by rounding the values.