DS 320 Algorithms for Data Science | **Lecture #23**
Spring 2024 | Prof. Kira Goldner

Approximation Algorithms: Randomized and Online

So far, we've been sure to always analyze runtime and often space as well. Another thing that we can add to our tradeoff is *optimality.*

Definition 1 (Approximation guarantee)**.** We say that an algorithm obtains an α*-approximation* to a maximization problem if in the worst case, the algorithm obtains at least $\textsc{Alg} \geq \alpha\, \textsc{opt}$ for $\alpha \in (0, 1)$.

Similarly, for a minimization problem, an algorithm obtains an α-approximation for $\alpha > 1$ if the algorithm's cost is at most $\textsc{Alg} \leq \alpha\, \textsc{opt}$.

Many times, it's useful to use randomness in our algorithm—to beat an adversary, or just because random choices do a good job of handling all of the different instances out there with some probability each. We give our guarantees in *expectation* over the randomness in the algorithm, i.e., for a maximization problem, we might say

$$\mathbb{E}_p[\textsc{Alg}] \geq \alpha\, \textsc{opt}$$

where p represents the random choices in the algorithm.

Sometimes, it is our *input* instance I that is random, in which case, we compare how both our algorithm do and how $\textsc{opt}$does in expectation over the random input that arrives, so we might prove a guarantee like

$$\mathbb{E}_I[\textsc{Alg}] \geq \alpha\, \mathbb{E}_I[\textsc{opt}].$$

Two probability essential facts to recall:

- Linearity of expectation: For any random variables X, x_i and constants c_i such that $X = \sum_i c_i x_i$, *regardless of whether or not the x_i's are independent*, $\mathbb{E}[X] = \sum_i \mathbb{E}[x_i]$. (This is not true for things like variance.)
- For a boolean (0/1) random variable x that is 1 with probability p, the expectation of x is $\mathbb{E}[x] = p$.

Randomized Algorithms

MAX SAT

Recall the definition of the 3-SAT problem: given a logical formula of n *boolean variables* $x_1, \ldots, x_n$ in *conjunctive normal form* (CNF), that is,

$$\phi = C_1 \wedge \cdots \wedge C_m$$

where C_i is one of m *clauses* each consisting of 3 disjoint *literals*, which are variables in either their positive or negative form, for example, perhaps

$$C_1 = (x_1 \vee x_2 \vee \overline{x}_3).$$

The goal of 3-SAT is to determine whether the formula ϕ can be satisfied.

Today, we look at the MAX SAT problem: very similar to 3-SAT, except that (1) each clause C_i may consist of any number of literals, and (2) rather than satisfy ϕ, our objective is to satisfy as many clauses C_i as possible. (Of course, if we satisfy all m clauses, then we satisfy ϕ.)

For our algorithm, we'll try the *simplest possible idea*: set each boolean variable to true or false with equal probability. This is the most basic idea in randomized algorithms, and often times, it's actually enough, as we'll see today! In linear programs, we often had a $\{0, 1\}$ decision variable—true or false, take or don't take into a set, something like this. For these variables, the idea of "set to 1 with probability $\frac{1}{2}$" (and thus to 0 with equal probability) often works!

Algorithm: Independently for all i, set

$$x_i = \begin{cases} \text{True} & \text{w.p. } 1/2 \\ \text{False} & \text{w.p. } 1/2 \end{cases}$$

Theorem 1. *This algorithm gives a $\frac{1}{2}$-approximation to* OPT *for MAX SAT.*

Proof. Let Y_j be a random variable that is 1 if clause j is satisfied and 0 otherwise. Let W be a random variable that is equal to the total weight of the satisfied clauses, so that $W = \sum_{j=1}^{m} w_j Y_j$. Let OPTdenote the optimum value of the MAX SAT instance. Then, by linearity of expectation, and the definition of the expectation of a 0-1 random variable, we know that

$$\mathbb{E}[W] = \sum_{j=1}^{m} w_j \mathbb{E}[Y_j] = \sum_{j=1}^{m} w_j \Pr[\text{clause } C_j \text{ satisfied}].$$

For each clause C_j, $j + 1, \ldots, n$, the probability that it is not satisfied is the probability that each positive literal in C_j is set to false and each negative literal in C_j is set to true, each of which happens with probability 1/2 independently; hence

$$\Pr[\text{clause } C_j \text{ satisfied}] - \left(1 - \left(\frac{1}{2}\right)^{l_j}\right) \geq \frac{1}{2},$$

where the last inequality is a consequence of the fact that $l_j \geq 1$. Hence,

$$\mathbb{E}[W] \geq \frac{1}{2} \sum_{j=1}^{m} w_j \geq \frac{1}{2} \text{OPT},$$

where the last inequality follows from the fact that to the total weight is an easy upper bound on the optimal value, since each weight is assumed to be nonnegative. ☐

Observe that if $l_j \geq k$ for each clause j, then the analysis above shows that the algorithm is a $\left(1 - \left(\frac{1}{2}\right)^k\right)$-approximation algorithm for such instances. Thus the performance of the algorithm is better on MAX SAT instances consisting of long clauses. This observation will be useful to us later on.

Although this seems like a pretty naive algorithm, a hardness theorem shows that this is the best that can be done in some cases. Consider the case in which $l_j = 3$ for all clauses j—that is, the MAX 3SAT problem. The analysis above shows that the randomized algorithm gives an approximation algorithm with performance guarantee $\left(1 - \left(\frac{1}{2}\right)^3\right) = \frac{7}{8}$. A truly remarkable result shows that nothing better is possible for these instances unless P = NP.

Theorem 2. *If there is an $(\frac{7}{8}+\varepsilon)$-approximation algorithm for MAX 3SAT for any constant $\varepsilon > 0$, then P = NP.*

MAX CUT

In the maximum cut problem (MAX CUT), the input is an undirected graph $G = (V, E)$, along with a nonnegative weight $w_{ij} \geq 0$ for each edge $(i, j) \in E$. The goal is to partition the vertex set into two parts, U and $W = V \setminus U$, so as to maximize the weight of the edges whose two endpoints are in different parts, one in U and one in W. We say that an edge with endpoints in both U and W is *in the cut*. In the case $w_{ij} = 1$ for each edge $(i, j) \in E$, we have an *unweighted* MAX CUT problem.

It is easy to give a $\frac{1}{2}$-approximation algorithm for the MAX CUT problem along the same lines as the previous randomized algorithm for MAX SAT. Here we place each vertex $v \in V$ into U independently with probability 1/2. As with the MAX SAT algorithm, this can be viewed as sampling a solution uniformly from the space of all possible solutions.

Theorem 3. *If we place each vertex $v \in V$ into U independently with probability 1/2, then we obtain a randomized $\frac{1}{2}$-approximation algorithm for the maximum cut problem.*

Proof. Consider a random variable X_{ij} that is 1 if the edge (i, j) is in the cut, and 0 otherwise. Let Z be the random variable equal to the total weight of edges in the cut, so that $Z = \sum_{(i,j)\in E} w_{ij} X_{ij}$. Let OPT denote the optimal value of the maximum cut instance. Then, as before, by linearity of expectation and the definition of expectation of a 0-1 random variable, we get that

$$\mathbb{E}[Z] = \sum_{(i,j)\in E} w_{ij}\mathbb{E}[X_{ij}] = \sum_{(i,j)\in E} w_{ij}\Pr[\text{Edge } (i,j) \text{ in cut}].$$

In this case, the probability that a specific edge (i, j) is in the cut is easy to calculate: since the two endpoints are placed in the sets independently, they are in different sets with probability equal to $\frac{1}{2}$. Hence,

$$\mathbb{E}[Z] = \frac{1}{2}\sum_{(i,j)\in E} w_{ij} \geq \frac{1}{2}\text{OPT},$$

where the inequality follows directly from the fact that the sum of the (nonnegative) weights of all edges is obviously an upper bound on the weight of the edges in an optimal cut. $\square$

Online Algorithms

The Ski Rental Problem

You're picking up a new hobby of skiing—you think. Every time that you rent skis, It costs \$100 and B-hundred dollars to buy skis (that is, we can think in units of hundreds of dollars). Maybe it'll be a lifelong hobby and you should invest that B up front, or maybe it's better to pay 1 each time in case it doesn't stick. Given that you don't know how many times you'll go skiing (say, some adversary decides whether you like it or not based on whether you purchase or not—isn't that how it always feels?), how should you decide whether to buy or not so that you can give *some guarantee?*

Definition 2 (Competitive ratio)**.** In an online setting, for a maximization problem, we say the *competitive ratio* of an algorithm is α if the algorithm obtains at least α-fraction of the *offline* full-information optimum $\textsc{opt}$, that is,

$$\textsc{Competitive Ratio} = \frac{\textsc{Alg}}{\textsc{opt}} \geq \alpha.$$

And for a minimization problem, the algorithm obtains at most α-times the offline $\textsc{opt}$: $\textsc{Alg} \leq \alpha\, \textsc{opt} \implies$:

$$\textsc{Competitive Ratio} = \frac{\textsc{Alg}}{\textsc{opt}} \leq \alpha.$$

Essentially, the *competitive ratio* is just the *approximation guarantee* in an online setting.

Online algorithm for Ski Rental: Rent at first; buy the skis on day $B + 1$.

Proof that comp ratio is 2: Let T be the length of the season.

- If $T \leq B$, then both $\textsc{opt}$ and $\textsc{Alg}$ pay T, so comp ratio is 1.
- If $T > B$, then $\textsc{opt}$ pays B on day 1, and CR is $(B + B)/B = 2$.

Lower Bound of 2: As soon as the algorithm buys the skis, stop the season. Alg buys on day $x+1$, then alg pays $x + B$, and $\textsc{opt}$ pays at most B. The competitive ration is $(x + B)/B = 2$ when $x = B$.