CSE 347 Analysis of Algorithms April 8, 2019

Lecture 20 Summary

Instructor: Brendan Juba

1. We can use competitive analysis to study policies for caching. The cost of a caching policy is how many times we incur a "miss," i.e., must service a request for data that is not stored in the cache. The problem is, when we get a request for a new address, how do we decide which data to remove ("evict") to make room for it? Recall that we wish to bound the ratio of the number of misses we incur to the optimal number of misses in hindsight.

   (a) The following randomized policy will achieve a $O(\log k)$ competitive ratio: values in the cache can be "marked" or "unmarked" (initially the cache is empty and all values are unmarked), and whenever we access an address in the cache we mark it. If the value was not in the cache, if all values were marked then we unmark all values; either way, we then evict an unmarked address at random to make room for the new request.

   (b) We unmark all values after $k+1$ distinct addresses are requested since the last time everything was unmarked. (Call this a "phase" of the request sequence.)

   (c) Let c_j denote the number of "fresh" requests in the jth phase, i.e., that were not marked during the previous phase. We saw last time that the optimal policy must have at least $\frac{1}{2}\sum_j c_j$ misses.

   (d) As for the number of misses incurred by the randomized policy, there are $k - c_j$ stale requests that the optimal policy may avoid incurring misses on. Since we evict addresses uniformly at random, the probability that we incur a miss on the ith request is at most $\frac{c_j}{k-(i-1)}$; summing over all of these misses gives an upper bound of $H_k c_j$ misses in expectation on this phase, where H_k is the kth harmonic number (which is $O(\log k)$). Thus, summing over all phases, we find that we are indeed $O(\log k)$-competitive.

2. A simple greedy algorithm is 2-competitive for scheduling to minimize "makespan": we have m identical machines and a sequence of requests that take arbitrary lengths of time $t_1, t_2, \ldots$. The *makespan* is the total time of all of the jobs assigned to a single machine.

   (a) A simple algorithm assigns an incoming job to an arbitrary machine with the smallest makespan.

   (b) Prior to assigning this job, the machine had at most a $1/m$ fraction of the total time assigned to it. Since some machine must have a makespan at least a $1/m$ fraction of the total time of all jobs in the sequence, this machine had at most the optimal makespan assigned to it.

   (c) The job we assign to this machine must be completed by some machine, hence the makespan is at least the time of this job.

   (d) Therefore, after we assign the job to a machine, that machine still has at most twice the optimal makespan. Since this holds for all assignments we make, the final allocation is indeed 2-competitive.

(e) Note: this problem is NP-complete! So it is useful to run this algorithm even if we are given all of the jobs up-front, i.e., as an "offline" algorithm. It does not solve the problem optimally, but it achieves a makespan that is within a factor of 2 of the optimum. It is our first *approximation algorithm.*

3. If an algorithm always achieves a solution that is at most $\alpha(n)$ times greater or less than the optimal solution (for minimization or maximization problems, respectively) for some function α, we say that it is an $\alpha(n)$-approximation algorithm.

   (a) Most NP-complete optimization problems have polynomial time approximation algorithms. The approximation factor $\alpha(n)$ that we achieve may or may not be adequate for practical needs. Nevertheless, approximation algorithms are, unlike average-case algorithms, often a practical way to approach NP-complete problems.

4. Greedy algorithms often make good approximation algorithms. There is a greedy $O(\log n)$-approximation algorithm for the Set Cover problem: given a set of subsets $S_1, S_2, \ldots, S_m \subseteq U$ of a universe U of size n, such that each S_i has a nonnegative weight w_i, find a collection of sets $\mathcal{C}$ such that $\bigcup_{i \in \mathcal{C}} S_i = U$ of minimum total weight.

   (a) Note, Set Cover generalizes Vertex Cover, so this problem is also NP-complete.

   (b) The greedy algorithm chooses the S_i that minimizes the ratio of the weight to the number of elements covered, $\frac{w_i}{|S_i \cap R|}$ where R is the set of remaining elements.

   (c) We'll say that the cost c_u the greedy algorithm pays to cover $u \in U$ is the ratio of the set that first covers it. Note that if we sum these costs over U, we get the total cost of our cover.

   (d) We will bound the approximation ratio by comparing the cost we pay to cover each element to the cost paid by an alternative set S_k that might not be picked in our cover. Supposing $|S_k| = d$, let's order the d elements; when we cover the jth element, at least $d - (j-1)$ elements were not yet covered, so S_k had a ratio of at least $\frac{w_k}{d-(j-1)}$. But we used a set S_i with $\frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{d-(j-1)}$ instead. Thus, $\sum_{s_j \in S_k} c_{s_j} \leq H_d w_k$.

   (e) Each set has size at most n, so each $w_i \geq \frac{1}{H_n} \sum_{u \in S_i} c_u$. Now, any optimal cover $\mathcal{C}^*$ has weight $\sum_{i \in \mathcal{C}^*} w_i \geq \frac{1}{H_n} \sum_{i \in \mathcal{C}^*} \sum_{u \in S_i} c_u \geq \frac{1}{H_n} \sum_{u \in U} c_u$ since $\mathcal{C}^*$ covers U. But $\sum_{u \in U} c_u$ is the cost of our cover; hence our cost is at most $O(\log n)$ times optimal.