CSE 347 Analysis of Algorithms April 1, 2019

Lecture 19 Summary

Instructor: Brendan Juba

1. Recall: For the Contraction algorithm (and Kruskal's Minimum Spanning Tree algorithm) we needed a "Union Find" data structure that supports the following operations
   (a) MakeSet(v): returns the "name" of a set containing simply the element v
   (b) Find($\{v\}$): returns the "name" of the set *currently* containing the set that originally contained v. Note that we can use this to check if two elements are contained in the same set by testing if Find($\{u\}$)=Find($\{v\}$).
   (c) Union(S_1, S_2): replaces the sets S_1 and S_2 with a new set $S_1 \cup S_2$ and returns the name of this new set (i.e., contracting two supernodes or joining two trees from a spanning forest into a single tree in Kruskal's algorithm).
2. There is a relatively simple implementation of Union Find that uses $O(1)$ time for both MakeSet and Find, and takes $O(\log n)$ amortized time over n Union operations.
   (a) We represent sets by linked lists, with an extra field that points to the head of the list and an extra field that, in the head of the list, contains the size of the list. We saw last time that this gives $O(1)$ time MakeSet and Find.
   (b) The Union operation is performed as follows. We take the head of the longer list as the head of the union set; we update its size to the sum of the sizes of the two lists, and walk the *shorter* list, updating each element to point to this new head. At the end, we attach the old second node of the larger set to the end of the list. The head's next field is updated to point to the old head of the smaller set.
   (c) If there are n union operations, then observe that since at most $2n$ elements have been involved in the union operations, the size of the largest set created can't be larger than $2n$ (actually, it's much smaller, but this is good enough).
   (d) Since we always walk the *smaller* of the two lists, whenever we access an element, the set containing it must at least double in size. So, for an element to be accessed ℓ times, we must have a set of size at least 2^ℓ. It follows that $\ell \leq \log_2 n + 1$; thus, our time per union operation is at most $O(\log n)$.
3. Another important type of on-line analysis considers the effect of irrevocable decisions that affect our performance in the future. We introduce *competitive ratios* to provide guarantees in this case.
   (a) The competitive ratio of an on-line algorithm is the maximum, over all input sequences, of the ratio of the cost incurred by the algorithm to the cost of the optimal sequence of responses in hindsight.
   (b) For example, we consider the following *elevator problem*: at any second, we can either wait for the elevator or we can take the stairs. The cost of a strategy is the total time we spend waiting.

(c) If taking the stairs takes S seconds and taking the elevator takes E seconds, then for example the strategy that just takes the stairs immediately has a competitive ratio of $\frac{S}{E}$ (because if the elevator comes right away, the optimal strategy only takes $E < S$ seconds). On the other hand, insisting on waiting for the elevator may obtain a competitive ratio of ∞, since there is no bound on how long the elevator takes to arrive.

(d) A good strategy is to wait $S - E - 1$ seconds, and then take the stairs if the elevator has not arrived yet. It is optimal unless it takes the stairs, and then the optimal policy just takes the stairs immediately. Its competitive ratio is then $\frac{2S-E-1}{S} = 2 - \frac{E+1}{S}$.

(e) *No deterministic strategy can achieve a better competitive ratio than* $2 - \frac{E+1}{S}$. Clearly if we wait Δ seconds more, then when we end up taking the stairs, we just increase our cost (and the competitive ratio) by $\frac{\Delta}{S}$. On the other hand, if we wait Δ seconds fewer, we may take $S + S - E - 1 - \Delta$ seconds whereas waiting for the elevator may only take $E + S - E - \Delta$ seconds. Since $S \geq E + 1$, this ratio is also strictly worse.

4. 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?

(a) The optimal policy with knowledge of the future always evicts the data that will be accessed furthest in the future.

(b) Any deterministic policy can be asked to provide the data it just evicted, causing it to miss on every request, whereas the optimal policy on such a sequence only misses at most once out of every $k + 1$ requests if the cache has size k.

(c) 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.

(d) 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.)

(e) Let c_j denote the number of "fresh" requests in the jth phase, i.e., that were not marked during the previous phase. Since in phases j and $j+1$ there are at least $k+c_{j+1}$ distinct requests, the optimal policy must have at least c_{j+1} misses. Therefore, summing over all phases, it incurs at least $\frac{1}{2}\sum_j c_j$ misses. Next time we'll bound the cost of our randomized policy in terms of this quantity.