CSE 347 Analysis of Algorithms April 9, 2020

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.
   (b) The head of the list will be the "name" of the set. So, MakeSet and Find are easily seen to be $O(1)$ time with this setup.
   (c) 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.
   (d) 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).
   (e) 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. As a second example of "amortized analysis," consider the following array-based implementation of a stack: we initially allocate an array of size ℓ, and maintain a "top" index. On pop, if top is 0, we return NULL, and otherwise we decrement top and return the element in index top. On push, if top is less than $\ell + 1$, we put the new element in the top index and increment it. If top is equal to $\ell + 1$, we first allocate a new array of size 2ℓ, copy the old array over to the new one, and then execute push as before. We can show that this takes constant *amortized time* per operation with the following general strategy:

(a) We introduce a *potential function* $\Phi_t \geq 0$ for the data structure after each tth operation. We take as convention that $\Phi_0 = 0$. We can think of Φ_t as the balance of a "bank account" – on cheap operations, we deposit a little extra into the account, which we withdraw to cover expensive operations.

(b) On each push that does not allocate a new array, we put an extra \$2 into the bank ($\Phi_t = \Phi_{t-1} + 2$). Thus, in total, we suppose both pop and these "cheap" push operations can be covered by a cost of \$3.

(c) Let's suppose now that the "expensive" push costs $\$\ell$ to copy the ℓ elements.

(d) Notice, when we last allocated the array, since we double the size each time, the array had at most $\ell/2$ elements in it. Therefore, we had at least $\ell/2$ cheap push operations since then, and so our bank account must contain at least $\$\ell$ to cover the cost of this expensive push, so we take $O(1)$ amortized time in total.

(e) In general, if we spend T_t time on the tth operation and $T_t + (\Phi_t - \Phi_{t-1}) \leq C$, then our amortized time is at most C per operation.

4. 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.