Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 8
September 26, 2012

Problem Set 3 Solutions

Problem 1. **this needs to be fleshed out** We will use 3 arrays A, B, and C. A will represent that actual array and hold the stored items in their stored positions. Arrays B and C will act as "checks" for which entries of A are truly initialized. B will contain a list, built up sequentially from position 0, of all entries of A that have been filled. We use a "length" variable i to keep track of the number of entries in B. Each time we fill a new position k in A, we store the value k at $B[i]$, then increment i. Array C will "index" Array 2. When we store k at $B[i]$, we store i at $C[k]$. Now, when we want to check whether position k is full, we look at $C[k]$ to find out where in B the value k should be stored. If $C[k] \geq i$ then it must be garbage meaning k is empty. If $C[k] < i$ then we check if $B[C[k]] = k$, in which case we have proof that position k is full; if $B[C[k]]$ has any other value then $C[k]$ is garbage meaning position k is empty.

Problem 2. We augment the vEB queue to also hold a maximum element. We implement the desired operations as follows:

- **find(x,Q)**: We check if x is either the minimum or maximum of the current queue. If so, we return it. Otherwise, make a recursive call and find $\mathsf{low(x)}$ in the subqueue $Q[\mathsf{high(x)}]$.
- **predecessor(x,Q)**: If x is less than the minimum of Q, return null. If x is greater than the maximum of Q, return the maximum of Q. Otherwise, we make a recursive call to find the predecessor of $\mathsf{low(x)}$ in the subqueue $Q[\mathsf{high(x)}]$. If the result of this recursive call is non-null, then we return the result. Otherwise, we make a call to find the predecessor of $\mathsf{high(x)}$ in $Q.\mathsf{summary}$. The result of this call tells us the subqueue that is non-empty among the subqueues. In particular, if it is non-null, then we return the maximum element from that subqueue. However, if the result of the call was null, then we can return the minimum of Q.
- **successor(x,Q)**: The algorithm is very similar to predecessor.

Problem 3. The key property to note in Dijsktra's algorithm acting on a graph $G = (V, E)$ starting from a node s is that the maximum difference between the estimated distances from s to any node in the priority queue holding the nodes in $V \setminus S$ is C. This can be proven by induction.

The solution then is to use two vEB queues, where the range of values in each vEB queue is $[1, C]$. The "left" queue will store the smaller values, and the "right" queue will store the larger values. When we want to insert an estimated distance into the queue, we will only insert the distance mod C. Along with each queue, we will store the "base" value of that queue, which will be a multiple of C. By the property noted above, we will never have to worry about filling a third queue. It is also possible to use only one vEB queue, but it is somewhat more annoying to make it work because you need to keep track of the minimum element represented in the queue so you know exactly where the data structure ends. The regular vEB operations will have to be modified appropriately, or it just won't work.

Let us first recall Dijkstra's algorithm. Given a weighted graph $G = (V, E)$, and a node $s \in V$, it finds the length of the shortest path from s to any node $u \in V$. The algorithm works by maintaining a set of nodes S for which the shortest path from s to $v \in S$ is already known. During each iteration, the algorithm chooses the node in $u \in V \setminus S$ whose estimated distance from s is minimum. Then, for each node w that is adjacent to u, the algorithm sets the estimated distance from s to w to be the the minimum of the current estimated distance and the distance from s to u, and then using the edge (u, w).

The key property to note in the above algorithm is that the maximum difference between the estimated distances from s to any node in the priority queue holding the nodes in $V \setminus S$ is C. This can be proven by induction on the iteration on the number of nodes in S.

Problem 4. **(a)** Consider the $(k+1)^{st}$ item inserted. Since only k buckets (at worst) are occupied, the probability that *both* candidate locations are occupied is only $(k/n^{1/5})^2$. Thus, the expected number of times an item is actually inserted into an already-occupied bucket is at most

$$\begin{aligned}\sum_{k=0}^{n-1}(k/n^{1.5})^2 &= \frac{(n-1)(n)(2n-1)}{6n^3} \\ &\leq 1/3\end{aligned}$$

Now let's consider pairwise collisions. Item k collides with item $j < k$ only if (i) one of the candidate locations of item k is the location of item j (this has probability at most $2/n^{1.5}$) and (ii) the other candidate location for item k contains at least one element (probability $k/n^{1.5}$). Thus, the probability k collides with j is at most k/n^3. Summing over the k possible values of $j < k$, we find the expected number of collisions for item k is at most k^2/n^3. Summing over all k, we get the same result as above: $O(1)$ expected collisions.

(b) Start with a 2-universal family of hash functions mapping n items to $2n^{1.5}$ locations. Consider any particular set of n items. Consider choosing a random function from the hash family. The probability that item k collides with item j is $1/2n^{1.5}$ by pairwise independence, implying by the union bound that the probability k collides with *any* item is at most $1/2\sqrt{n}$.

Now suppose that we allocate *two* arrays of size $2n^{1.5}$ and choose a random 2-universal hash function from the family independently for each array. If an item has no collision in *either* array, then it will be placed in an empty bucket by the bash function. We need merely analyze the probability that this happens for every item (this would make the bash function perfect).

The probability that item k has a collision in *both* arrays is at most $(1/2\sqrt{n})^2 = 1/4n$. It follows that the expected number of items colliding with some other item is at most $1/4$. This implies in turn that with probability $3/4$, every item is placed in an empty bucket by the (perfect) bash function. This in turn implies that *some* pair of 2-universal hash functions defines a perfect bash for our set of n items.

Since every set of items gets a perfect bash from this scheme, it follows that the family of pairs of 2-universal functions above is a perfect bash family. Since the 2-universal

family has size polynomial in the universe, so does the family of pairs of 2-universal functions.

(c) When we sample a hash function from the above 2-universal family, we get a 3/4 probability of having no collisions. It follows that if we make 2 or more attempts, we can expect to find a collision-free hash function.

(d) If we map our n items to k candidate locations in an array of size $n^{1+1/k}$, our collision odds work out as above and we get a constant number of collisions. Similarly, k random 2-universal hash families, each mapping to a set of size $n^{1+1/k}$, has a constant probability of being perfect for any particular set of items, so the set of all such functions provides a perfect family (of polynomial size for any constant k). This gives a tradeoff of k probes for perfect hashing in space $O(n^{1+1/k})$.

Note that while we can achieve perfect hashing to $O(n)$ space, the resulting family does *not* have polynomial size (since a different, subsidiary hash function must be chosen for each sub-hash-table).

Problem 5. There is no available solution for this problem at this time.