Diego Escobedo March 4, 2020

6.046 Problem Set 4

Collaborators: *Temi Omitugoon, Ritaank Tiwari, Mikael Nida*

Problem 1

(A) Our algorithm is going to consist of two steps:

1. First, we are going to run DFS and use the algorithms we learned in 6.006 to find possible back edges (including self loops). However, one condition is that we are going to run this only on vertices that have an edge connected to them, because otherwise we are going to be pointlessly checking a lot unnecessary nodes that obviously will not have a cycle. If we find a cycle, we output cycle. Otherwise, we go to step 2.
2. The second step is going to be to actually assign the edges (keys) to the slots (vertices). What we are going to do is for each connected component (excluding the nodes we didn't check above), traverse the edges, and for each edge, assign the element stored in the edge to the slot corresponding to the 'child node' (next in the traversal order) of the edge we are currently on.

- Correctness: This algorithm first uses DFS back edge checking, a technique we learned and proved that works in 6.006, to detect edges. Once we do that, we sequentially assign edges in traversal order. to vertices. This, we ensure that each item is mapped to a different slot and there is no repetition
- Runtime: DFS is typically O(V+E) = O(m+n), but since we restricted the DFS to vertices that actually have edges, we are not checking the worst-case O(m) vertices that have no edge. Therefore, step 1 runs in O(n). For the second part, we are going to be traversing every single edge to assign it to a slot, and no more. Therefore, since there are n nodes and each assignment happens in O(1) time, we get O(n) for step 2. Two O(n) operations mean the whole runtime is O(n).

(B) First we begin by observing that the maximum amount of edges in a graph is $\frac{m^2}{2}$, because there are m possible edges for each of the m possible nodes (to the m-1 other nodes and a self-loop). However, this method double counts the edges, so we divide by 2. If we are assuming that the hash functions that generate the edges behaves randomly, then all of these edges could be equally expected. Further, if there are n items to be hashed, then we will have n edges. The probability of a single edge being in a given place is therefore

$\frac{n}{\frac{m^2}{2}} = \frac{2n}{m^2}$. Therefore, the chance of k consecutive edges being in a given configuration (in this case a k-cycle) is $(\frac{2n}{m^2})^k$.

(C) We first want to prove the hint. We begin by realizing that we can choose a permutation of k nodes to be connected into a cycle (by k edges) by getting a permutation (no combination because the order matters in a cycle) of k of the m nodes (which is $\frac{m!}{(m-k)!}$). However, in this case each cycle is counted k times, once for each starting node. So, we get that there's actually $\frac{m!}{k*(m-k)!} \leq m^k$ possible cycles. So, by the union bound, we get that the probability of any k cycle appearing is the probability of a given k cycle appearing times the total number of possible cycles. By the results of this part and the last part, that would be $m^k * (\frac{2n}{m^2})^k = (\frac{2n}{m})^k$.

(D) When we plug in m=10n into our equation from last part, we get $(\frac{2n}{10n})^k = (\frac{1}{5})^k$. So, the probability of ANY cycle appearing is (by the union bound) is $\sum_{i=1}^{m}(\frac{1}{5})^i > \sum_{i=1}^{\infty}(\frac{1}{5})^i$. This is an infinite geometric sum, which has a solution of $\frac{r}{1-r} = \frac{1/5}{4/5} = \frac{1}{4}$. So, the probability of any cycle appearing in G is less than or equal to $\frac{1}{4}$.

(E) Our algorithm is going to be very simple. Lets call our procedure from part a, ASSIGN (because it assigns items to slots). The algorithm takes O(n) time. From d, we know that performing ASSIGN, given two arbitrary hash functions from a UHF, succeeds with probability greater than or equal to 3/4. Therefore, we can imagine a geometric random variable X tracking the number of Bernoulli trials needed to get one success (defined as a successful run of ASSIGN), with probability of success p = $\frac{3}{4}$. Therefore, the expected number of trials until success is $\frac{1}{p} = \frac{4}{3} \leq 2$. Therefore, we would expect two runs of ASSIGN to successfully put n items into a hash table with m=10n slots, given that if ASSIGN returns 'cycle' we generate two different random hash functions to use on our next try.

Correctness: Our algorithm is literally just using the procedure from part a (whose correctness we proved previously) repeatedly. We know that every outcome from ASSIGN ir correct in and of itself, but we are simply waiting for a 'desired' result to occur. Even though it is not guaranteed to happen, we *expect* it to happen within 2 tries, but obviously we could end up running this algorithm forever and it would never work. It is important to note that after a failure we have to generate two new hash functions to maintain the independence that allows us to model the trials as Bernoulli random variables.

Runtime: We just explained that we expect ASSIGN to succeed after 4/3 (round up to 2) tries. If each ASSIGN takes O(n), a constant number of those operations will run in expected O(n) time.

Problem 2

(A) In this case, the data structure can have false positives, but it cannot have false negatives.

The reason it cannot have false negatives is that there is no way to set bits back to 0. Therefore, if we hash a word w, all the bits $A[h_1(w)], A[h_2(w)], ..., A[h_k(w)]$ are going to turn to 1, and there will *never* be a way to turn it back to 0 with the given operations. Any time we look up w, those bits will be 1 which will indicate to us that the word has already been inserted.

However, the data structure can have false positives. Imagine we have a word w that we have not inserted yet. However, imagine that the bits $A[h_1(w)], A[h_2(w)], ..., A[h_k(w)]$ have all already been turned into 1 by a certain amount of words that we inserted previously (due to collisions between the hashes). Even though when we look up the word, it appears as though it has been hashed, in reality it has not.

(B) Consider the first bit in the table. What is the probability that, given a single hash of word w, it ends up in this bit? Assuming that the hash is a uniform hash function, then the chance is $\frac{1}{m}$. So, the chance that it doesn't is $(1 - \frac{1}{m})$. However, each time we hash a word, we are actually hashing it k times with k different hash functions (which we are told are independent). Because they are independent (linearity of expectation), the probability of all k not landing on the first bit is the same as the product of those k individually not landing, or $(1 - \frac{1}{m})^k$. When we do that n times (on independent word hashes), that gets raised to the nth power, leaving us with $(1 - \frac{1}{m})^{kn}$. However, this is the chance that no words get hashed to the first bit, and we want to find the chance that some word was hashed to the first bit. Therefore, we get $1 - (1 - \frac{1}{m})^{kn}$.

(C) We want that $1 - (1 - \frac{1}{m})^{nk}$ be equal to one half, ie $1 - (1 - \frac{1}{m})^{nk} = \frac{1}{2}$, which is $\frac{1}{2} = (1 - \frac{1}{m})^{nk}$. Using the hint, we can see that $\frac{1}{2} \leq e^{-\frac{nk}{m}}$. Taking logs, we get $ln(\frac{1}{2}) \leq -\frac{nk}{m}$, followed by $-ln(\frac{1}{2}) * \frac{m}{n} \geq k$, which we can express again as $k \leq ln(2) * \frac{m}{n}$.

(D) As we discussed earlier, there can only be false positives. We get a false positive when the k bits associated with the word w are all 1. Let us start with a small thought exercise. Imagine we have the m slots in front of us in an array A. We know that the chance of every individual slot being 1 is upper bounded by $\frac{1}{2}$. Imagine we are looking through every element of A. We know that, because of conditional probabilities, our answer as to the probability of each slot being 1 might change if we actually inspect the slots. For example, if we inspect the first slot, and see that it is a 1, then the chance of subsequent slots being 1 is less than one half. Conversely, if the first slot has a 0 in it, then the chance of subsequent slots being 1 is more than one half.

Knowing this, let us now consider the k distinct slots that the word w was mapped to. If

any of them are 0, we know for a fact that the word has not been inserted yet. However, we want to calculate the bound for what the chance is that they are all 1 when the word hasn't really been inserted. So, lets us gather the k slots (that are all 1, but imagine we don't know that yet) that w was mapped to in A and gather them in a second array called B. So, before inspecting, all the elements of B have a 1/2 chance of being 1. If we inspect teh first element, we see that it is 1, and then by our thought experiment above, we know that the subsequent chances of the elements in B being 1 decreases. Therefore, the maximum it will ever be is 1/2. If we consider that the probability that LOOKUP(w) is a false positive is $\frac{1}{2} * (\frac{1}{2} - \epsilon_1) * (\frac{1}{2} - \epsilon_2) * ... * (\frac{1}{2} - \epsilon_{k-1})$, we know that it is bounded from above by $(\frac{1}{2})^k$.

(E) We evaluate k and get $k \leq 20ln(2)$. We are told that the probability of having a collision is $\frac{k^2}{m}$, which we can evaluate in terms of n as $\frac{(20ln(2))^2}{20n} = \frac{20ln^2(2)}{n}$. Therefore, using union bound, we can deduce that the chance of LOOKUP(w) failing is the chance of it failing when there are no collisions plus the chance of it failing when there are collisions. Thus, we proved in part D that the chance of it failing without collisions is $(\frac{1}{2})^k$, and the chance of it failing with collisions is unknown (could be up to 1%) but it only happens $\frac{20ln^2(2)}{n} = O(\frac{1}{n})$ of the time. Therefore, the chance of lookup failing is upper bounded by $(\frac{1}{2})^k + O(\frac{1}{n})$