Problem Set 7 Solutions

This problem set is due **at 11:59pm** on **Wednesday, April 19, 2017.**

EXERCISES (NOT TO BE TURNED IN)

Hashing

- CLRS Exercise 11.3-5.

Probability Review

- CLRS C.2-2
- CLRS 5.4-6

Problem 7-1. Strongly 2-Universal Hashing [50 points]
Let H be a family of hash functions, where each hash function $h \in H$ maps keys from the universe U to $\{0, 1, \ldots, m-1\}$. We call H **strongly 2-universal** if for all $x_1, x_2 \in U$ such that $x_1 \neq x_2$ and for a uniformly randomly chosen h, the pair $(h(x_1), h(x_2))$ is equally likely to be any of the m^2 pairs (y_1, y_2), where y_1 and y_2 are both elements of $\{0, 1, \ldots, m-1\}$.

(a) [12 points] Prove that if H is strongly 2-universal, then it is also universal.
Solution: The strongly 2-universal condition tells us that

$$Pr_{h \in H}[(h(x_1), h(x_2)) = (i, i)] = \frac{1}{m^2}$$

for any i in $\{0, 1, \ldots, m-1\}$.
Given a pair x_1, x_2 where $x_1 \neq x_2$, we see that

$$Pr[h(x_1) = h(x_2)] = \sum_{i=0}^{m-1} Pr[(h(x_1), h(x_2)) = (i, i)] = \frac{m}{m^2} = \frac{1}{m}$$

and therefore H is a universal hash family.

(b) [8 points] Provide a hash family H that is universal but is not strongly 2-universal. Write your answer as a table, where each row corresponds to a hash function and each column corresponds to a key. For simplicity, try to make the values $|H|$, $|U|$, and m as small as possible.
Hint: You can find an answer where $|H|$, $|U|$, and m are at most 3.
Solution: There exists an answer where $|H| = |U| = m = 2$. For the universe $U = \{x, y\}$, consider the following H

	x	y
h_1	0	0
h_2	1	0

If we choose a random hash function, the probability that the two keys collide is the probability that we choose h_1, which is $\frac{1}{2} = \frac{1}{m}$. Thus H is universal.
However, H is not strongly 2-universal because the pair $(h(x), h(y))$ should take on the values $(0, 0), (1, 0), (0, 1)$, and $(1, 1)$ with equal probability, but for this H it will never equal $(1, 1)$ or $(0, 1)$

(c) [15 points] Suppose we are given a universal hash family H. An adversary wants to force a collision, and the adversary knows exactly what H is - that is, the adversary knows the descriptions of each individual hash function $h \in H$. In particular, the following steps take place, in order.

1. We choose a hash function h from H uniformly at random. The adversary does not know which h we pick.
2. The adversary picks a key x and we tell the adversary $h(x)$.
3. The adversary picks a key y in an attempt to force a collision.

The adversary **succeeds** if in step 3 the adversary can choose a key $y \neq x$ such that $Pr[h(x) = h(y)] > 1/m$ (using the knowledge of the value of $h(x)$, but without knowing h).

Describe a universal hash family H for a general m such that the adversary can **succeed** on step 3 above. You may verbally describe your hash family (clearly and concisely), or adopt the the same format as in part (b) - however, note that in contrast with part (b), your table should scale to arbitrarily large m.
Note: You may write down H for a specific value of m to illustrate your idea, but you must clearly explain how to extend your construction to an arbitrarily large m.
Solution: Consider the following hash family H for $m = 2$.

	x	y	z
h_1	0	0	1
h_2	1	0	1

H is universal because the probability of x and y colliding is $1/2$, the probability of x and z colliding is $1/2$, and the probability of y and z colliding is 0.
Now note that the adversary can ask for $h(x)$, and based on that value it can determine which hash function is being used. If $h(x) = 0$, we chose h_1 and the adversary can then give us y to force a collision. If $h(x) = 1$, we chose h_2 and the adversary can then give us z to force a collision.

To extend this to a larger value of m, we use the same construction. We will use m hash functions and $m+1$ unique keys. The first key x will take on the values 0 through $m-1$ when hashed, and it basically serves to identify which hash function is chosen. The next m keys will always hash to the same value (for every possible hash function) and are useful because we can choose the correct key to force a collision based on the value of $h(x)$.

(d) [15 points] Now consider the same set of steps in part (c), except with the assumption that H is strongly 2-universal. Show that the adversary cannot succeed, i.e. force a collision with probability greater than $1/m$ in step 3, on *any* H that is strongly 2-universal.

Solution: With a strongly 2-universal hash family, the adversary cannot force a collision with probability better than $1/m$. Essentially, knowing $h(x)$ gives the adversary no information about $h(y)$ for any other key y.

We can prove this formally using conditional probabilities. Suppose we choose a random hash function $h \in H$, and then the adversary forces us to hash some key x and learns the value $h(x) = X$. Then the adversary gives us any key $y \neq x$, hoping to cause a collision. By definition of strong 2-universality, we have for any x and y with $x \neq y$,

$$Pr_{h \in H}[h(y) = h(x) | h(x) = X] = \frac{Pr_{h \in H}[h(y) = h(x) \text{ and } h(x) = X]}{Pr_{h \in H}[h(x) = X]} = \frac{1/m^2}{1/m} = \frac{1}{m}$$

Therefore, no matter which x the adversary chooses first, and which $h(x) = X$ value it learns, the probability of any particular y colliding with x is only $1/m$.

Problem 7-2. An Enemy of My Enemy [50 points]

Fakebook Inc. is a social media website for the less amicable, where instead of making friends, users can designate "enemies." These relationships are encoded in an undirected graph $G = (V, E)$, where V represents the set of users, and two users v_i and v_j are enemies if and only if $(v_i, v_j) \in E$ (indeed, the enemy relationship must be mutually acknowledged).

Mark, a social analyst, wants to determine whether the old adage "an enemy of an enemy is a friend" applies to Fakebook's social network. In particular, Mark wants to determine whether we can partition the vertices into two sets $V_1, V_2 \subseteq V$ of "mutual friends" where no two users in the same set have an edge between them. In other words, Mark wants to determine if G is bipartite.

Due to a recent surge in hostility, the graph G has suddenly become *dense*, such that $|E| = \omega(|V|)$ (or equivalently, $|V| = o(|E|)$). Unfortunately, this makes it infeasible to store all of G in memory. Your task is to help Mark by coming up with an efficient algorithm to determine whether G is bipartite, given a stream of edges and only $O(|V|)$ space.

Note: For the sake of simplicity, assume each vertex can be represented in $O(1)$ space by an integer in $\{1, ..., |V|\}$ (i.e. ignore the bit-complexity of storing integers).

(a) [18 points]

Warm Up: Let E_k represent the first k edges in the stream, and let $G_k = (V, E_k)$. Devise an algorithm that returns the smallest k such that there exists a cycle in G_k. Be sure to specify which data structures you are using and analyze their space complexity. For full credit your algorithm should run in $o(|V||E|)$, but partial credit will be awarded to slower solutions.

Pre-Solution:

The problem asks us to detect the first instance of a cycle in G in a streaming setting. Since we are given a *stream* of edges, it is a good first step to think about properties of acyclic graphs are broken with the introduction of a new edge. In particular, one property that distinguishes acyclic graphs from cyclic graphs is that acyclic graphs can have *at most 1 path* between any two vertices. Therefore, if we introduce an edge that creates a new path between already-connected vertices, we've introduced a cycle in the graph.

The problem now boils down to coming up with a data structure that keeps track of already-connected vertices (which is a familiar problem we've seen while studying MSTs). There are many ways to implement a data structure that does this, and one particular solution is highlighted below.

Solution:

Maintain a forest which represents subgraphs G_k as you process edges in the stream. By definition of forest, there can be at most 1 path between any two vertices, so if G_k is a forest then it doesn't contain a cycle. Upon encountering a new edge(u, v) in the stream, we check whether it breaks the forest invariant by checking whether there

already exists a path between u and v. This can be accomplished in many ways, one of which is a simple BFS from u and v.

At a first glance, the algorithm seems to take $O(|V||E|)$ due to the BFS for every edge. However, a more careful analysis shows that after $|V| - 1$ edges, we are guaranteed to encounter at least 1 cycle. Therefore, in the worst case, our algorithm for $O(|V|)$ iterations, which gives a total runtime of $O(|V|^2) = o(|V||E|)$.

We could use many data structures for this (a different one is discussed in part c), but a simple graph representation (with nodes/pointers) or adjacency list achieves the $O(|V|)$ space constraint, since we store at most $O(|V|)$ edges at any given time by the same argument above.

(b) [25 points]

Devise an algorithm that determines whether or not G is bipartite using only $O(|V|)$ space. For full credit, your algorithm should run in $O(|V||E|)$.

Hint: Consider modifying your algorithm from (a) to detect when certain properties of bipartite graphs are violated.

Pre-Solution:

As the hint suggests, we should look for a property of bipartite graphs that can be broken upon seeing a new edge. As hinted by the Warm-Up, this probably has to do with cycles. As it turns out, bipartite graphs can only have *even*-length cycles. Therefore, we can boil our problem down to one that detects odd-length cycles – only then have we violated bipartiteness.

This new problem shows strong semblance with our original problem, except with the caveat that we now allow even-length cycles. This makes the space constraint more difficult to satisfy since we cannot throw away information about even-length cycles, and at the same time, cannot terminate on any arbitrary cycle. This should be a signal that maybe we do not need to maintain every even-length cycle in the graph in order to detect the first odd-length cycle. Often, this compromise in "throwing out / ignoring" components while still maintaining adequate informations requires a deeper look into invariants or properties that can be maintained with minimal data. In other words, we want to think about which edges we can *safely* ignore, and argue that it doesn't accidentally transform a non-bipartite graph into a bipartite one or vice versa.

Solution:

As mentioned, we want an algorithm that returns upon encountering an edge that creates a *odd*-length cycle, but without keeping track of all even-length cycles (as this can require $O(|E|)$ space). The algorithm itself is not substantially different from part (a) - the only difference is that we ignore an edge (u, v) in the stream that creates an even-length cycle. We can check the parity of the cycle by simply running a BFS between (u, v) and checking that the path length is odd. This achieves a runtime of $O(|V||E|)$ (note that we are no longer guarantee to terminate after $|V| - 1$ edges). The

main challenge now is to prove that ignoring an edge that creates an even-length cycle does not turn a non-bipartite graph into a bipartite one.

To prove this, we first prove that a non-bipartite G must contain an *odd*-length cycle. If a graph has only even-length cycles, then we can color each vertex with 2 colors so that any adjacent vertices in a cycle have different colors. These vertices can be partitioned into two sets based on their color, and since there exist no edges between vertices of the same color, this graph is bipartite. Hence, a non-bipartite graph must have at least 1 *odd*-length cycle.

We must now prove that eliminating an edge that creates an even-length cycle does not eliminate *all* odd-length cycles in original graph G. Say we ignore an edge (u, v) that is part of an odd-length cycle. Note that because u and v are part of some odd-length cycle, there must exist an even-length path between u and v. We only ignore (u, v) if (u, v) is also part of an even-length cycle–i.e. there also exists an odd-length path between u and v. Therefore, although we ignore (u, v), there must still exist an odd-length cycle in the graph, namely the one composed of the even-length path between u and v and the odd-length path between u and v.

(c) [7 points]

Improve your algorithm from (b) to run in amortized $O(|E| \log |V|)$.

Hint: Think of an amortized data structure that we've studied in the past.

Pre-Solution:

In our implementation for part (b), we focus on mainly two aspects of our data structure. First, it should keep track of the connected components in each subgraph G_k. As we've seen in the past (e.g. with Kruskal's), a great data structure for connected components is the Union-Find data structure. Second, we require a quick way to check odd/even-ness of the cycles. In cases where we run an expensive procedure to determine a fairly simple property, we can look into maintaining some sort of auxiliary information on each node of the graph to help answer our queries faster. The problem then shifts to maintaining this auxiliary information, which lends itself more clearly to amortized analysis.

Solution:

First, we switch our forests/BFS approach to a Union-Find data structure (where each set represents a connected component). We initialize our Union-Find with $|V|$ sets, each containing 1 vertex; however, to help quickly determine whether a cycle is even or odd, we maintain a color $(0, 1)$ on each vertex. When we encounter an edge, we first check whether the two vertices are in the same set. If they are in the same set and have the same color, we know we have created an odd-length cycle (see argument from part b). This reduces the cycle check complexity to $O(1)$. However, if they are in different sets, we union their sets while maintaining the invariant that adjacent vertices have different colors. This requires us to possibly invert every color of one of the two

sets.

We can use the technique of merging smaller sets with larger sets (just like we did for the linked-list implementation of Union-Find) since the problem of updating pointers is analogous to the problem of updating colors. To restate the aggregate analysis: consider the number of times we update the color on a vertex v. Each time we update v's color, we necessarily double the size of v's set. Therefore, we update v's color at most $\log |V|$ times across all operations of the data structure. Summing over all $v \in V$ we get $O(|V| \log |V|)$ color updates across all $|E|$ operations.