Assignment -5

Advanced Algorithms (CPT_S 515)

Name: Coral Jain

WSU ID: 11632780

Problem 1:

We are given in the question a family of hash functions, defined by:

$$H = \{h: 1 \leq i \leq 8\}$$

where each h_i maps an array to eight bits.

As per the criteria for universality, it is defined that a hash function is universal if any two hash functions follow the criteria such that any two keys key1 and key2 are related such that key1 ≠ key2 where both key1 and key2 belong to the universal set.

Also, it is defined that probability of two hash values being the same such that it causes collision is given by $P\big(h(key1) = h(key2)\big) \leq \frac{1}{M}$,where M is the number of indices.

Since, we are mapping a number of given hash functions with a number of indices and it is given in the question that we are mapping to an array of eight bits and an array of bits would be a collection of a number of bits that has only 1s and 0s. therefore, the probability for a collision in this case would be less than $\frac{1}{2}$.

Now, let us consider a case where we have four keys defined by:

Key 1: 00000001
Key 2: 00000011
Key 3: 00001000
Key 4: 00000010

Now, we observe that all the four keys given above differ only by 2 bits. In other words, only one bit is 1 and the rest all are 0.

Now, we calculate the probability of collision for any two keys given above,

P (key1 = key 4) = $\frac{7}{8}$ (since both the keys differ only by 2 bit)
P (key2 = key3) = $\frac{5}{8}$ (since both the keys differ only by 3 bits)

If we look at the above probabilities, we find that both the probabilities are 0.875 and 0.625, which means that probability of both is greater than 0.5 (we learned earlier that probability must be $\leq \frac{1}{2}$).

This contradicts our definition that of the upper bound of probability. Therefore, we can say that H is **NOT** a universal hashing function.

Problem 2:

As per the criteria for universality, it is defined that a hash function is universal if any two hash functions follow the criteria such that any two keys x_1 and x_2 are related such that $x_1 \neq x_2$ where both $x, y \in U$.

Also, it is defined that probability of two hash values being the same such that it causes collision is given by $P\big(h(key1) = h(key2)\big) \leq \frac{1}{M}$,where M is the number of indices.

Let us consider from the given condition that we have a key x which can take up k values. Therefore, we take two keys x_1 and x_2 and we consider a case when two x_1, x_2 collide.

Therefore, $h(x_1) = h(x_2)$ will be the collision condition:

Note: Please note that the values x_{1i} and x_{2i} are two keys and take up values from 0 to k i.e. $k + 1$ values.

$$\sum_{i=0}^{i=k} r_i x_{1i} \bmod M = \sum_{i=0}^{i=k} r_i x_{2i} \bmod M$$

$$\sum_{i=0}^{i=k} r_i x_{1i} \equiv \sum_{i=0}^{i=k} r_i x_{2i} \bmod M$$

$$\sum_{i=0}^{i=k} r_i (x_{1i} - x_{2i}) \equiv 0 \bmod M$$

$$r_i (x_{10} - x_{20}) + \sum_{i=0}^{i=k} r_i (x_{1i} - x_{2i}) \equiv 0 \bmod M$$

$$r_i (x_{10} - x_{20}) \equiv - \sum_{i=0}^{i=k} r_i (x_{1i} - x_{2i}) \bmod M$$

Dividing both sides by $x_{10} - x_{20}$ will give us:

$$r_0 \equiv - \sum_{i=0}^{i=k} r_i (x_{1i} - x_{2i})(x_{10} - x_{20}) \bmod M$$

From the above equation, we can conclude that r_0 can take up $(0 \ldots . M - 1)^k$ values and r can take $(0 \ldots . M - 1)$ values.

Hence, in this case, probability of collision is $\frac{M^k}{M^{k+1}} = \frac{1}{M}$

which satisfies our condition that the given set of hash functions is universal.

Problem 3:

In order to represent a graph using hash value, we need to use Perron's theorem. Perron's theorem takes into account Perron value. A Perron value is the largest eigenvalue in a matrix that has only positive elements. This largest eigenvalue can be obtained from the matrix of a graph and also from the corresponding eigen vectors of a graph. This Perron value is bounded by the largest sum of a row and the lowest sum of a row of the adjacency matrix derived from a given graph. Once eigenvalue i.e. λ is determined, it is easier to hash the graph using concept of locally sensitive hashing.

We can do this with the help of a number of steps:

- Derive the adjacency matrix of a graph using the concept of graph theory, wherein if two nodes are connected by an edge, the value is 1, otherwise it is 0.
- Find the largest value or the Perron value of the graph.
- Use the concept of locally sensitive hashing by diving the adjacency matrix into y bands of x rows.
- On every row, perform this step and put the identical elements into the same bucket. Apply this to each and every row of the given matrix of the graph. This will give us a real number.
- Follow the above steps in order to derive a real number from the graph G.

Problem 4:

In the algorithm called Randomized Quicksort in the problem given to us, we generate a random permutation of an array as input before sorting it using the pivot method like merge sort. However, in our question, we are required to generate a random number using a random generator given to us and use it to create a random graph.

For this, we can assume that the number n is a random number generated by the random generator r(n) and then we use this number to generate a random graph. Now, we go about solving this by considering n nodes in order to generate a graph.

Since, we know that for a given graph, a pair can be formed by taking 2 nodes at a time. Therefore, for n nodes, the number of pairs would be nC_2 which solves to $\frac{n(n-1)}{2}$. Therefore,

for the given 5 nodes, we shall have $\frac{5(5-1)}{2} = 10$ pairs. Take r(10) to generate a random number between 1 to n.

Now we can follow these steps in order to come up with a randomized graph:

- Take 5 nodes given in the question.
- Use pseudo random generator to generate a random number between 1 to n for one pair.
- If we assume that the number generate is k, we shall check the conditions using if-else statements whether k is greater than 5 or not.
- If k < 5, we will not draw an edge between the pair. However, if k > 5, we will draw an edge.
- For each and every pair of nodes for the graph with 5 nodes, we will follow the same procedure and hence, a randomized graph can be obtained.

References:

https://www.geeksforgeeks.org/quicksort-using-random-pivoting/

https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0123.pdf

Problem 5:

We are required to solve this problem using Bloom filter. Bloom filters are used to determine whether a number or a string is present in a set or not. It acts like a data structure which acts in a very efficient manner and returns the result very rapidly. However, there is one problem with such a filter, which is that it can store the elements but removing an element is not possible. Also, another issue with our given problem is this that while Mr. X is travelling from Pullman to New York, we have to take this into consideration that it is a very long journey, and therefore, the number of places that he comes across will be very large in number. Therefore, storing this in a single set would be very time-consuming for the algorithm or maybe impossible, since Bloom filter would take a lot of time to search it in such a large set (if such a large set S is built).

So, for this, we need to remove the elements if we are considering a small set S. Due to this problem, we shall use more than one filter in order to do this. First of all, when Mr. X moves in his car, and as per his GPS details, his location can be traced and the locations of the places which are nearby his location can be kept in S and locations other than near to his location can be pushed out of the set S. However, as it was said above that it is not possible to use Bloom filter here as deletion is not possible with this filter.

This is done with the help of Forgetful Bloom Filters (FBF), where server only needs to remember a moving window of history, so long as that window is longer than the period the client will keep re-trying the operation in i.e. a Forgetful Bloom Filter keeps the total number

of members in check by forgetting old members. In some problems like ours, three filters are used:

1. Past bloom filter
2. Present bloom filter
3. Future bloom filter

Past bloom filter: The places that have been crossed on the way will be stored in this filter.

Present bloom filter: The places that are nearby the current location of Mr. X will be accessed using this filter.

Future bloom filter: The places that Mr. X is yet to visit will be accessed using this filter.

As we talked above that a window is maintained for problems where there is an issue of deletion of elements from the bloom filter, we will use the above filters in following manner:

- ⇒ The past bloom filter is deleted or removed as it contains the places that have been visited by Mr. X.
- ⇒ The present bloom filter can now be converted into past bloom filter after the previous past bloom filter has been removed.
- ⇒ Like the above step, the future bloom filter can now be converted into present bloom filter after the previous present bloom filter has been removed.
- ⇒ Now, we have a new filter with no elements in it.
- ⇒ Please note that the above steps are taken at any fixed location (say l).
- ⇒ The filters keep updating as per the location of Mr. X.

In the steps shown above, the bloom filter can be utilized for this problem.

Reference:

https://hughfdjackson.com/engineering/counters,-idempotence-and-forgetful-bloom-filters/

Problem 6:

The general family for LSH is $H: h: U \rightarrow S$ is (r_1, r_2, p_1, p_2) such that for all $d_1, d_2 \subset U$. If the distance between d_1, d_2 is at most r_1, then the probability of $h(d_1)$ and $h(d_2)$ being equal is greater than or equal to p_1. If the distance between d_1, d_2 is greater than r_2, then the probability of $h(d_1)$ and $h(d_2)$ being equal is less than or equal to p_2. Here each array is 10 bits and hamming distance is used to measure the similarity between two arrays.

S is the set of 10-bit arrays $a\{a_1, a_2, \ldots . . a_n\}$. Let us assume here that the hamming distance is between d_i and d_k, then we have:
$H = [h_1 | h_i(a_1, a_2, \ldots . a_n) = a_i \text{ for } c = 1,2,..n]$ where h_i is a function in the hash family and a_i is a bit array.

Method 1:

Step 1:

First, as per our hash function, we shall gather all the pairs with the distance d_1 and d_2. Now, if they have a small distance, then the collision is allowed by insertion of each bit array as an element of a linked list. This allows similar bit arrays with same hash value to be kept together. This is called chaining. This generates a hash table M_1.

Step 2:

Second, create a large hash table M_2 and making the load factor $1 - \frac{1}{m}$. Scan the current table M_1 for each slot with and without linked list values. If the chained items in a slot have a consecutive distance of 1, them items are inserted into M_2 using the new function.

Step 3:

Finally, after placing each item into M_2, we get one-to-one mapping of all bit arrays with mapping to hash values done based on how small their hamming distance is.

Reference:

http://pgraycode.wordpress.com/2009/09/21/uniformity/

Method 2:

The method of doing this it along with hamming distance concept will not be able to satisfy one-to-one mapping condition. The same condition, which is not satisfied by using hash function, which may not be one-to-one. However, the concept of weighted distance can be used for this. Lei Zhang et. al have defined this in his paper titled, *"Binary Code Ranking*

with Weighted Hamming Distance", where the authors have described the way to handle ambiguity that arises from results or the values having same Hamming Distance. In this method, hash bits are mapped to bit-level weights (they have to be different). Later, the hash bits that have similar hamming distance are given the ranks. As per the paper, "the returned binary codes are ranked at a finer-grained binary code level". In order to apply this to our problem, we can choose an arbitrary value, from which we can obtain weighted value of the hamming distance when we input the array of 10 bits that we discussed in the problem.

Reference:

Link: https://ieeexplore.ieee.org/document/6619052/authors#authors

Paper Citation: L. Zhang, Y. Zhang, J. Tang, K. Lu and Q. Tian, "Binary Code Ranking with Weighted Hamming Distance," *2013 IEEE Conference on Computer Vision and Pattern Recognition*, Portland, OR, 2013, pp. 1586-1593.
doi: 10.1109/CVPR.2013.208