1 Advanced Algorithms Homework 5

Q1

H is **not universal**: To determine whether the family $H = \{h_i : 1 \leq i \leq 8\}$ of hash functions is universal, we carefully examine the definition of universal hashing and apply it to the given family H – a family H of hash functions mapping a universe U to a range $\{0, 1, \ldots, M-1\}$ is universal if, for any two distinct elements $x, y \in U$ (where $x \neq y$), the probability that a randomly chosen hash function h from H maps both x and y to the same value is at most $\frac{1}{M}$. Mathematically:

$$\forall x \neq y \in U, \quad \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{M}.$$

So Given:

- $H = \{h_i : 1 \leq i \leq 8\}$,
- Each h_i maps an 8-bit array $a_1 a_2 \ldots a_8$ to its i-th component: $h_i(a_1 a_2 \ldots a_8) = a_i$.

The range of the hash functions is as follow , each h_i outputs a single bit (either 0 or 1), so $M = 2$. For two distinct 8-bit arrays $x = x_1 x_2 \ldots x_8$ and $y = y_1 y_2 \ldots y_8$, let $D(x, y)$ denote the set of indices where x and y differ:

$$D(x, y) = \{i \mid x_i \neq y_i\}.$$

Let $d = |D(x, y)|$ be the number of differing bits between x and y. The number of hash functions h_i that do not cause a collision (i.e., $h_i(x) \neq h_i(y)$) is exactly d. Conversely, the number of hash functions that **do** cause a collision (i.e., $h_i(x) = h_i(y)$) is $8 - d$. Thus, the probability that a randomly chosen h_i from H causes a collision is:

$$\Pr[h_i(x) = h_i(y)] = \frac{8 - d}{8}.$$

To satisfy universality, we require:

$$\Pr[h_i(x) = h_i(y)] = \frac{8 - d}{8} \leq \frac{1}{M}.$$

Substituting $M = 2$:

$$\frac{8 - d}{8} \leq \frac{1}{2}.$$

Solving for d:

$$\frac{8 - d}{8} \leq \frac{1}{2} \implies 8 - d \leq 4 \implies d \geq 4.$$

However,

- **If $d \geq 4$:** The probability $\Pr[h_i(x) = h_i(y)] \leq \frac{1}{2}$, satisfying the universal condition.
- **If $d < 4$:** The probability $\Pr[h_i(x) = h_i(y)] > \frac{1}{2}$, violating the universal condition.

Counter example Demonstrating Non-Universality Consider the specific case where:

$$x = 00000000 \quad \text{and} \quad y = 00000001.$$

Here, x and y differ only in the last bit ($d = 1$). Applying our probability formula:

$$\Pr[h_i(x) = h_i(y)] = \frac{8-1}{8} = \frac{7}{8} = 0.875.$$

Since $0.875 > 0.5 = \frac{1}{M}$, the probability of collision exceeds the universal threshold. This counterexample clearly shows that the family H does not satisfy the universal hashing condition for all pairs of distinct inputs.

Thus, the family H is **not universal** because there exist distinct input pairs (e.g., $x = 00000000$ and $y = 00000001$) for which the probability $\Pr[h_i(x) = h_i(y)] = \frac{7}{8}$ exceeds $\frac{1}{2}$. Therefore, H fails to meet the criteria required for universality.

Q2

H is a **universal** family of hash functions. The family H is defined as:

$$H = \{h_r : r \in [M]^k\},$$

where each hash function h_r maps a k-dimensional vector $x = (x_1, x_2, \ldots, x_k)$ to:

$$h_r(x) = \left(\sum_{i=1}^{k} r_i x_i \right) \mod M.$$

Here:

- M is a prime number.
- $x = (x_1, x_2, \ldots, x_k)$ and $r = (r_1, r_2, \ldots, r_k)$ are vectors in $[M]^k$, i.e., each component $x_i, r_i \in \{0, 1, \ldots, M-1\}$.

As stated in Q1, the universality can be shown in the following way :

$$\forall x \neq y \in U, \quad \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{M}.$$

To prove that H is universal, we need to show that for any two distinct vectors $x, y \in [M]^k$ with $x \neq y$:

$$\Pr_{r \in [M]^k}[h_r(x) = h_r(y)] \leq \frac{1}{M}.$$

A collision occurs if:

$$h_r(x) = h_r(y) \implies \left(\sum_{i=1}^{k} r_i x_i\right) \equiv \left(\sum_{i=1}^{k} r_i y_i\right) \pmod{M}.$$

Subtracting the right-hand side from the left-hand side, we obtain:

$$\sum_{i=1}^{k} r_i(x_i - y_i) \equiv 0 \pmod{M}.$$

Let $\Delta_i = x_i - y_i$ for each $i = 1, 2, \ldots, k$. Since $x \neq y$, there exists at least one index j such that $\Delta_j \neq 0$. Without loss of generality, assume that $\Delta_1 \neq 0$. The collision condition becomes:

$$r_1\Delta_1 + \sum_{i=2}^{k} r_i\Delta_i \equiv 0 \pmod{M}.$$

Solving for r_1:

$$r_1\Delta_1 \equiv -\sum_{i=2}^{k} r_i\Delta_i \pmod{M}.$$

$$r_1 \equiv -\Delta_1^{-1}\left(\sum_{i=2}^{k} r_i\Delta_i\right) \pmod{M},$$

where Δ_1^{-1} is the multiplicative inverse of Δ_1 modulo M. Since M is prime and $\Delta_1 \neq 0$, Δ_1^{-1} exists.

The equation

$$r_1 \equiv -\Delta_1^{-1}\left(\sum_{i=2}^{k} r_i\Delta_i\right) \pmod{M}$$

determines a unique value for r_1 given specific values of $r_2, r_3, \ldots, r_k$.

So, since each r_i for $i = 2, 3, \ldots, k$ is chosen uniformly at random from $[M]$, the right-hand side of the equation is uniformly distributed over $[M]$. Therefore, for each fixed choice of $r_2, r_3, \ldots, r_k$, there is exactly one value of r_1 that satisfies the collision condition. The probability that a randomly chosen r_1 satisfies the equation is:

$$\Pr\left[r_1 \equiv -\Delta_1^{-1}\left(\sum_{i=2}^{k} r_i\Delta_i\right) \pmod{M}\right] = \frac{1}{M}.$$

Since the probability of collision $\Pr[h_r(x) = h_r(y)]$ is exactly $\frac{1}{M}$ for any two distinct vectors x and y, the family H satisfies the universality condition:

$$\forall x \neq y \in [M]^k, \quad \Pr_{h_r \in H}[h_r(x) = h_r(y)] \leq \frac{1}{M}.$$

Therefore, H is a **universal** family of hash functions.

Q3

To hash a graph into a single number, we first represent the graph using its Laplacian matrix, which captures its structural properties. We then vectorize this matrix into a one-dimensional vector and apply a universal hash function to this vector to obtain the final hash value. This approach ensures that the probability of two distinct graphs colliding is bounded by $\frac{1}{M}$, where M is a chosen prime number.

To begin let G be an undirected, unweighted graph $G = (V, E)$ with n vertices and A be an $n \times n$ matrix where $A_{ij} = 1$ if there is an edge between vertices i and j, and 0 otherwise. Then degree Matrix D: An $n \times n$ diagonal matrix where D_{ii} equals the degree of vertex i.

Laplacian Matrix L**:** Defined as $L = D - A$.

The matrix properties is as follows :

- L is symmetric and positive semi-definite.
- L encapsulates both the connectivity and degree information of G.

Next we want to vectorize the Laplacian Matrix as this transformation allows the application of standard hashing techniques designed for one-dimensional data and because the vector $\mathbf{v}$ retains all structural information from the Laplacian matrix:

To convert the Laplacian matrix L into a one-dimensional vector $\mathbf{v} \in \mathbb{R}^{n^2}$ by concatenating its rows sequentially:

$$\mathbf{v} = [L_{11}, L_{12}, \ldots, L_{1n}, L_{21}, \ldots, L_{nn}]^\top$$

Next, we then can apply the universal hash function $\mathcal{H}$. Universal hashing ensures that the probability of collision between distinct inputs is low. The linear combination followed by a modulo operation provides a compact and uniform hash distribution:

$$\mathcal{H} = \{h_{\mathbf{r}} : \mathbf{r} \in [M]^k\}$$

where:

- M is a large prime number.
- $k = n^2$ is the dimensionality of $\mathbf{v}$.
- Each $h_{\mathbf{r}}$ is defined as:

$$h_{\mathbf{r}}(\mathbf{v}) = \left(\sum_{i=1}^{k} r_i v_i \right) \mod M$$

- $\mathbf{r} = [r_1, r_2, \ldots, r_k]^\top$ is a random vector with elements uniformly chosen from $[M]$.

To prove Universality, we want to show that for any two distinct graphs G and H, the probability that $h_{\mathbf{r}}(\mathbf{v}_G) = h_{\mathbf{r}}(\mathbf{v}_H)$ is at most $\frac{1}{M}$.

Proof:

1. **Distinct Vectors:** Since $G \neq H$, their Laplacian vectors $\mathbf{v}_G$ and $\mathbf{v}_H$ are distinct. Thus, $\mathbf{v}_G \neq \mathbf{v}_H$.

2. **Collision Condition:** A collision occurs if:

$$h_{\mathbf{r}}(\mathbf{v}_G) = h_{\mathbf{r}}(\mathbf{v}_H)$$

   Subtracting both sides:

$$\left(\sum_{i=1}^{k} r_i v_{G,i}\right) \equiv \left(\sum_{i=1}^{k} r_i v_{H,i}\right) \mod M$$

$$\sum_{i=1}^{k} r_i (v_{G,i} - v_{H,i}) \equiv 0 \mod M$$

   Let $\Delta_i = v_{G,i} - v_{H,i}$. Since $\mathbf{v}_G \neq \mathbf{v}_H$, there exists at least one $\Delta_j \neq 0$. Without loss of generality, assume $\Delta_1 \neq 0$.

3. **Isolating r_1:**

$$r_1 \Delta_1 + \sum_{i=2}^{k} r_i \Delta_i \equiv 0 \mod M$$

   Solving for r_1:

$$r_1 \equiv -\Delta_1^{-1} \left(\sum_{i=2}^{k} r_i \Delta_i\right) \mod M$$

   Here, Δ_1^{-1} exists since M is prime and $\Delta_1 \neq 0$.

4. **Probability Calculation:** - For fixed $r_2, r_3, \ldots, r_k$, the right-hand side of the equation defines a unique value for r_1. - Since r_1 is chosen uniformly at random from $[M]$, the probability that it equals this specific value is:

$$\Pr[r_1 = \text{specific value}] = \frac{1}{M}$$

5. Therefore, the probability that $h_{\mathbf{r}}(\mathbf{v}_G) = h_{\mathbf{r}}(\mathbf{v}_H)$ is:

$$\Pr[h_{\mathbf{r}}(\mathbf{v}_G) = h_{\mathbf{r}}(\mathbf{v}_H)] = \frac{1}{M}$$

   This satisfies the universality condition:

$$\Pr[h_{\mathbf{r}}(G) = h_{\mathbf{r}}(H)] \leq \frac{1}{M}$$

Q4

To implement S such that Mr. X can query it effectively, we construct a random graph with 5 nodes by first determining a random number of edges M. Next, we select M unique edges uniformly at random from the set of all possible edges between the nodes. This process ensures that every graph with exactly M edges has an equal likelihood of being generated. By selecting edges without replacement, we avoid duplicates, maintaining the integrity of the graph. Adjusting M allows for flexibility in generating graphs of varying densities, ranging from sparse to fully connected.

Walk through : 1. Create a graph G with 5 nodes labeled from 1 to 5. 2. Determine the total number of possible edges is:

$$E = \binom{n}{2} = \frac{n(n-1)}{2}$$

3. For $n = 5$, $E = \frac{5\times4}{2} = 10$.

4. Next, we decide on Number of Edges M generate $M = r(10)$, where $1 \leq M \leq 10$.

5. List all possible edges by creating a list L of all possible edges between the nodes:

$$L = \{(i, j) \mid 1 \leq i < j \leq 5\}$$

Initially, L contains 10 edges.

Randomly Select M Edges**:**

- For $k = 1$ to M:
    1. Let $N =$ length of L.
    2. Generate $x = r(N)$, where $1 \leq x \leq N$.
    3. Select the x-th edge $e = (i, j)$ from L.
    4. Add edge e to graph G.
    5. Remove edge e from list L to avoid duplicates.

After selecting M edges, G is the desired random graph.

Algorithm in Pseudocode:

```
G = empty graph with 5 nodes
E = [(1,2), (1,3), ..., (4,5)]  # All possible edges
M = r(10)  # Number of edges to include

for k = 1 to M:
    N = length of E
    x = r(N)
    e = E[x]
    Add edge e to graph G
    Remove e from E

return G
```

Q5

To implement the dynamic set S, we can use an Invertible Bloom Filter (IBF). Unlike standard Bloom filters, IBFs support both insertions and deletions, which is essential as S is continuously updated as Mr. X drives. Additionally, IBFs associate keys with values, allowing efficient queries for specific types of places (e.g., "Is there a restaurant nearby?"). This approach offers a space-efficient, probabilistic data structure that supports fast updates and queries, making it ideal for Mr. X's in-car device.

1. Initialization

- Define parameters:
  - m: Size of the IBF array (number of cells).
  - k: Number of independent hash functions.
  - $h_1, h_2, \ldots, h_k$: Hash functions mapping keys to indices $[0, m-1]$.
- Initialize the IBF:
  - Each cell IBF$[i]$ contains:

$$\text{IBF}[i] = (\text{count} = 0, \text{keySum} = 0, \text{valueSum} = 0)$$

  - For all i from 0 to $m-1$, set the initial values to zero.

2. Insertion

The key-value pair is hashed k times. For each hash function h_i, update the corresponding cell in the IBF by incrementing the count and applying XOR operations on the key and value.

When a new place becomes nearby, insert it into S as follows:

```
function Insert(key, value):
    for i from 1 to k:
        index = h_i(key) mod m
        IBF[index].count += 1
        IBF[index].keySum ^= key
        IBF[index].valueSum ^= value
```

3. Deletion

Deletions use the same key and value as in the insertion step. The count is decremented, and the XOR operation removes the key and value from the IBF.

When a place is no longer nearby, remove it from S as follows:

```
function Delete(key, value):
    for i from 1 to k:
        index = h_i(key) mod m
        IBF[index].count -= 1
        IBF[index].keySum ^= key
        IBF[index].valueSum ^= value
```

4. Query

Scan the IBF for cells that have a positive count and a value sum matching the query. This operation returns `True` if the queried place type is possibly nearby.

To check if a specific type of place (e.g., "restaurant") is nearby:

```
function Query(value):
    for i from 0 to m-1:
        if IBF[i].count > 0 and IBF[i].valueSum == value:
            return True  # Possibly nearby
    return False  # Not nearby
```

The XOR operation ensures that insertions and deletions cancel each other out:

- $a \oplus a = 0$ (XOR of the same value removes it).
- $a \oplus 0 = a$ (XOR with zero retains the value).
- Insertion:

$$\text{count} + 1, \quad \text{keySum} \oplus key, \quad \text{valueSum} \oplus value$$

- Deletion:

$$(\text{keySum} \oplus key) \oplus key = \text{keySum}, \quad (\text{count} + 1) - 1 = \text{count}$$

Collisions in an Invertible Bloom Filter (IBF) can lead to false positives, where hash collisions cause unrelated key-value pairs to incorrectly match a query. However, this issue can be mitigated by carefully selecting the IBF parameters, specifically the array size (m) and the number of hash functions (k), to minimize the likelihood of collisions. The IBF is highly space-efficient, requiring only $O(m)$ space, which is significantly less than explicitly storing all elements in a set. Furthermore, the time complexity for insertions, deletions, and queries is $O(k)$, where k is the number of hash functions, making the IBF computationally efficient while maintaining its probabilistic guarantees.

Q6

The goal of this problem is to design a hash function H that maps 10-bit binary arrays $v \in \{0,1\}^{10}$ to integers while ensuring two key properties: one-to-one

mapping and locality sensitivity. A one-to-one mapping guarantees that each unique array is assigned a distinct hash value, while locality sensitivity ensures that arrays with small Hamming distances produce hash values that are numerically close. To achieve these objectives, we propose a two-step hashing method.

First, locality-sensitive hashing (LSH) is used to group similar arrays into the same or nearby bins based on their Hamming distance. Then, within each bin, a perfect hashing technique is applied to assign a unique value to each array. The objective is to map similar arrays (small Hamming distance) to the same bin with high probability.

So, we begin by defining a hash function $h(v)$ by selecting k random positions $\{i_1, i_2, \ldots, i_k\}$ from the 10-bit array v:

$$h(v) = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$$

Where the output of $h(v)$ is a k-bit binary value, which serves as the bin index. Arrays with similar values at the selected positions are more likely to hash to the same bin. k: Number of bits used in the hash function. Smaller k increases collisions (larger bins), while larger k reduces collisions. Arrays are grouped into 2^k bins, with similar arrays more likely to share a bin.

Next we can ensure that each array within a bin is assigned a unique hash value. So for each bin B, collect all arrays that hashed to that bin using $h(v)$. Then we construct a perfect hash function h_B for each bin:

$$h_B : B \to \{0, 1, \ldots, |B| - 1\}$$

Use minimal perfect hashing algorithms or lookup tables to uniquely map each array within B.

Finally, we combine the results from the two steps to define the overall hash function:

$$H(v) = \text{concat}(h(v), h_B(v))$$

where:

- $h(v)$: The bin index (output from LSH).
- $h_B(v)$: The unique index within the bin.

Thus, the hash function satisfies the following properties:

1. **One-to-One Mapping:**
    - Each array v is assigned a unique hash value because h_B ensures uniqueness within each bin.
2. **Locality Sensitivity:**
    - Arrays with small Hamming distances are likely to hash to the same bin or nearby bins, resulting in hash values that are numerically close.

For example, lets assume the following:

- 10-bit arrays $v \in \{0,1\}^{10}$.
- $k = 4$: Select 4 random bit positions for LSH.

Steps:

1. For $v = 1101001010$, use LSH to compute the bin index:

$$h(v) = (v_2, v_5, v_7, v_9) = (1, 0, 1, 0)$$

   Bin index: $h(v) = 1010$ (decimal 10).

2. Within bin 10, compute the unique index $h_B(v)$:

$$h_B(v) = 3 \quad \text{(assuming } v \text{ is the 4th unique array in bin 10).}$$

3. Final hash value:

$$H(v) = \text{concat}(1010, 3) = 10100011.$$