Cpt S 515 Homework #5

Instructor: Zhe Dang

Student: Sheng Guan

11593929

1. Consider a family H of hash functions:

$$H = \{h_i : 1 \le i \le 8\}.$$

Each h_i is to map an array of eight bits into its i-th component: $h_i(a_1 \dots a_8) = a_i$. Is H universal? why or why not?

Answer:

H is not universal.

Let's recall the universal hash function H's definition first. Let H be a finite set of hash functions from U to [M]. If $\forall\ x \neq y \in U$, $Pr(h(x)=h(y)) \le 1/M$, where h is randomly chosen from H, then H is universal.

$|U| = 2^8$,

[M]=8

1/M=1/8

In the family H of hash functions, h is random function in H.

Pr(h(x)=h(y) |h(x))= 1/2

Then Pr(h(x)=h(y) |h(x))* Pr(h(x))= ½ *½ + ½*½ >1/M, that violates the universal definition.

Hence, H is NOT universal.

2. Here is a classic example of universal family of hash functions. Let M be a prime number and, as usual, $[M] = \{0, 1, \cdots, M - 1\}$. Consider the following family of hash functions:

$$h_r(x) = (r \circ x \bmod M);$$

where $r,x \in [M]^k$ (where k is a given constant like 10), and $r \circ x = \sum_i r_i x_i$. Show that the family of hash functions (for the given k) is universal.

Answer:

$h_r(x) = (r \circ x \bmod M)$

The result of h_r (x) will from 0 to M-1, no matter what the value of k is, [M]=M.

h is randomly chosen from H since r,x $\in [\mathrm{M}]^k$

Let H be a finite set of hash functions from U to [M].

$\forall$ x≠y$\in$U, Pr(h(x)=h(y))=1/M≤1/M

By the definition, the family of hash functions (for the given k) is universal.

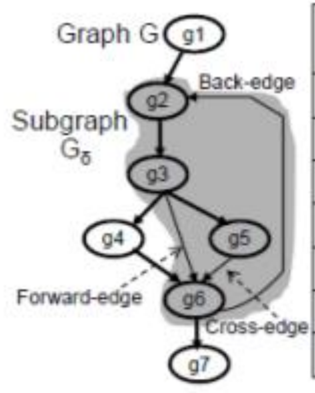
3. So far, what we have learned about hasing is to hash an array of numbers into one number (e.g., locality sensitive hashing). Can you suggest a way to hash a graph into a number (which could be a real number)?

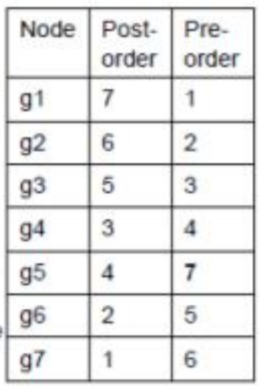
Answer:

A directed graph G(V, E) is a set of nodes (or vertices) V and a set of edges E between these nodes: e(x, y) is an edge from x to y, (x, y) $\in$ V × V . Undirected graphs can be represented as directed graphs.

The major challenges in hashing graphs are: (1) a graph may have cycles, (2) changes to the nodes affects hashes of multiple other nodes.

Graph Traversal:





Node	Post-order	Pre-order
g1	7	1
g2	6	2
g3	5	3
g4	3	4
g5	4	7
g6	2	5
g7	1	6

A graph G(V,E) can be traversed in a depth-first search manner and get Post-order, Pre-order defined. The post-order number of a node is smaller than that of its parent. The pre-order number of a node is greater than that of its parent. The pair of post-order and pre-order number for a node in a non-binary tree correctly and uniquely determines the position of the node in the structure of the tree, where the position of a node is defined by its parent and its status as the left or right child of that parent. Then the idea is to represent the graph as a tree. We do a depth-first search of the graph G(V,E), which gives us one or more depth-first trees.

We use the post-order number o_x and pre-order number q_x of node x to define tree-edge, forward-edge, cross-edge and back-edge. (1) e(x,y) is a tree edge Iff $o_x > o_y$ and $q_x < q_y$; (2) e(x,y) is a forward-edge iff there is a path from x to y consisting of more than one tree-edges, $o_x > o_y$ and $q_x < q_y$; (3) e(x,y) is a cross-edge iff $o_x > o_y$ and $q_x > q_y$; (4) e(x,y) is a back-edge iff $o_x < o_y$ and $q_x > q_y$

A hashing scheme for graphs requires a hash function algorithm.

HASHING: The hash algorithm gH, output a real value $rH_{G(V,E)}$

gH:

1. Input: a graph G(V, E)and its efficient tree-representation.
2. Sort the source nodes of the graphs in the non-decreasing order of their contents or label.
3. Carry out depth-first traversal of the graph G(V, E) on the first source node x in the sorted order. If there are no source nodes in G(V, E), choose x randomly.
4. If node y is visited in DFS, assign its (post-order, pre-order) number to it.
5. If the edge e(y, z) is not a tree-edge and is cross-edge, then create an edge-node yz having the following content $ce((p_y, r_y),(p_z, r_z))$; if forward-edge: $fe((p_y, r_y),(p_z, r_z))$, and back-edge: $be((p_y, r_y),(p_z, r_z))$
6. Add an edge from y to the new node yz, and remove the edge e(y, z).
7. Remove x from the sorted order of source nodes if exists.
8. If there are nodes in G(V, E) that are yet to be visited, then repeat from 2.
9. Compute hash of each node x as follows: $H_y \leftarrow \text{Hash}((p_y, r_y)||y)$.
10. To each edge-node yz, assign $H_{yz} \leftarrow \text{Hash}(H_y||H_z)$.
11. Compute the hash of graph gHG(V, E) as follows:

$gH_{G(V,E)} \leftarrow \text{Hash}(H_{y1}||H_{y2}|| \ldots ||H_{ym})$

where yi refers to the i'th node in the increasing order of the post-order numbers of the nodes, and of the originating nodes of the edge-nodes, and m is the total number of nodes in the efficient-tree including original nodes and the edge-nodes.

12.Transfer this string $gH_{G(V,E)}$ to a real value $rH_{G(V,E)}$

This construction uses graph traversal techniques and is highly efficient with respect to updates to graphs: it requires as little as two (O(1)) hash values to be updated to refresh the hash of the graph.

The algorithm idea refers to the Paper:
Kundu, Ashish, and Elisa Bertino. "On Hashing Graphs." IACR Cryptology ePrint Archive 2012 (2012): 352.

4. Randomized quicksort is a Las Vegas algorithm where the first step is to create a random permutation of the input array of numbers before the second step of running quicksort. Now, we assume that we have a high quality psuedo random generator r(n) that will generate a random number in 1..n. Please show how to generate a "random" graph with 5 nodes.

Answer:

A graph will be randomized graph if we insert the node in random order. Or equivalently we could apply the random permutation algorithm from last time to "unsort" the elements, and then insert them one at a time.

```
Step1: Use random generator r(n) to generate 25 numbers in 1..n and store them in
an array R[1..25].
Step2: For (i=0;i<5;i++){
        For(j=0;j<5;j++){
            (1)Create a random permutation of the array R[1..25] and the pivot
            A[p] is marked.
```

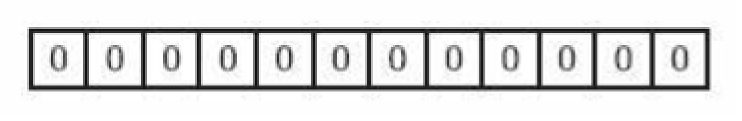
(2)Assign A[p] as G[i][j], we can further set some threshold:

$$G[i][j] = \begin{cases} 0, & A[p] < n/2 \\ 1, & A[p] \geq n/2 \end{cases}$$

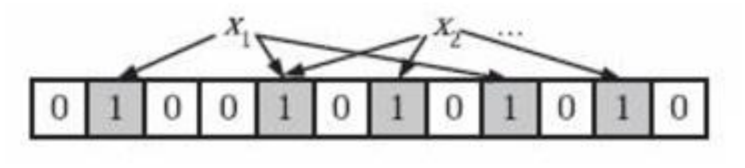
Step3: We will get a graph adjacency matrix G whose 0 and 1 elements can represent a “random” graph with 5 nodes.

5. Mr. X drives on I-90 all the way from Pullman to New York (Let's assume that Pullman is Spoakne). On his car, there is a device that can suggest all the interesting places nearby that Mr. X might visit (and spend some money at these places of course). These places are stored in a set S and will be updated automatically while Mr. X is driving. Please suggest a way to implement the S so that Mr. X can query (e.g., "Is there a restaurant nearby?", etc.). You shall use Bloom filter to store S. Feel free to look up papers on the Internet.

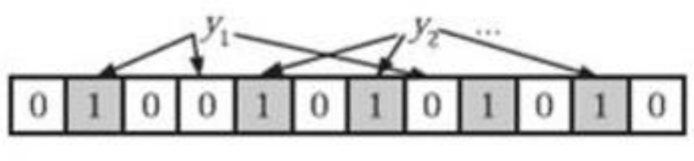
Answer:
Bloom Filter is a space-efficient random data structure that uses bit arrays to represent a set concisely, and it can quickly determine whether one element belongs to this set or not. As its initial state, Bloom Filter is an array of bits containing m bits, each bit is set as 0 as shown below:



In order to represent a set of n elements like S ={x_1, x_2, …, x_n}, Bloom Filter uses k independent hash functions that respectively map each element x_i into the set {1,…,m} Here we store the S with interesting places that are within certain radio threshold of Mr. X’s current location. The set S will be updated based on Mr. X’s current location and S can be pre-store in the database corresponding to location index for Mr. X to query.
For any element x_K, the location that the i-th hash function $h_i(x_k)$ maps will be set to 1 (1 ≤i≤k). Note that if a position is set to 1 many times (i-th hash function and j-th hash function both map this position as 1, i<j≤ k), then only the first time i-th hash function will work. The rest hash functions have no effect. In the below example, two hash functions select the same position to set as 1.(fifth from the left)



When Mr X queries “Is there a restaurant y nearby?”,
In determining whether y belongs to this set S, here S can be a dynamic set of restaurants’ names near Mr X’s location.
We apply k hash functions to y, and if all the positions of $h_i(y)$ are 1 ($1 \le i \le k$) then we consider y to be an element in the set S, else determine y is not an element in the set S.
For example, if y= y_1, then y does not belong to the set S. If y= y_2, y can belong to the set S or y can just happen to be a false positive and we can compute false positive rate.

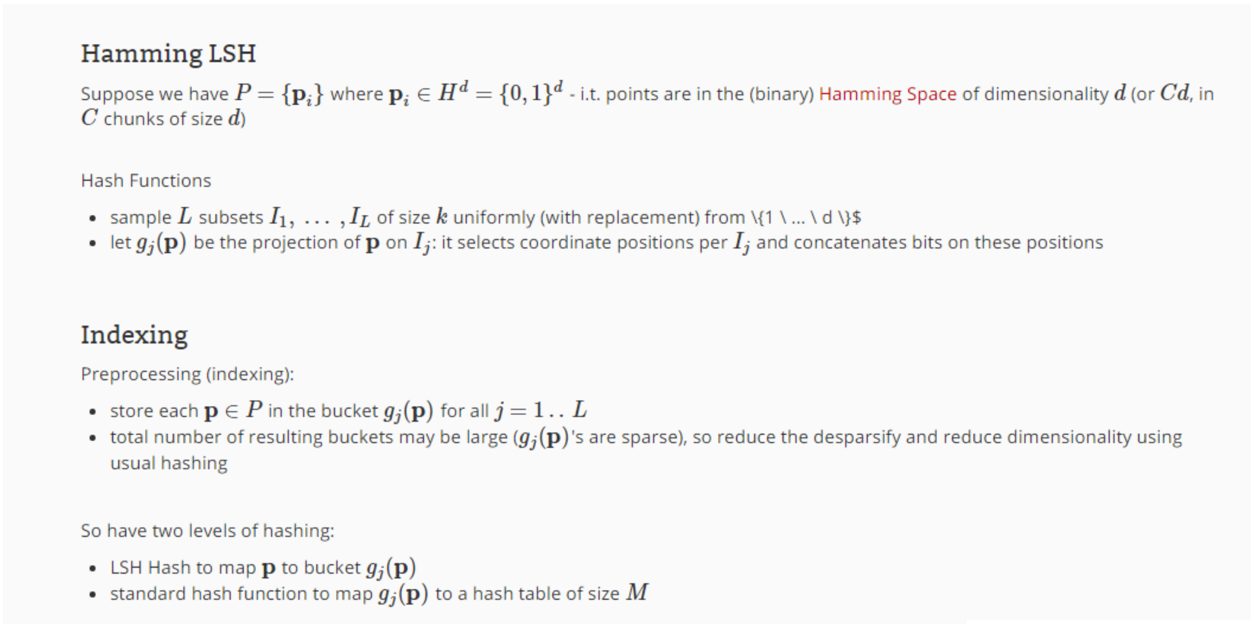


When Mr X queries “Is there a restaurant nearby?”, here S can be a dynamic set of interesting place attributes such as museum, theatre, etc. near Mr X’s location. We apply k hash functions to y=”restaurant”, and if all the positions of hi(y) are 1 ($1 \le i \le k$) then we consider y to be an element in the set S with certain error rate, else determine y=”restaurant” is not an element in the set S, then no restaurant nearby.

6. We know many ways to hash an array of integers into a number. However, hash itself is loosy --- that is, the function many not be one-to-one. Can you suggest a way to hash an array of 10 bits into a number such that
a. the hash is one-to-one, and,
b. the hash is locality sensitive (i.e., when the Hamming distance between two such arrays of 10 bits is small, then so is the distance between their hash values).

Answer:

Suppose that our points are d-bit (in our example is 10) vectors and that we use Hamming distance for our metric. In this case, using the family of one-bit projections
$\{h_i \mid h_i(x) = x_i\}$ gives a locality-sensitive hash family.
We can show this family is (r, r(1 + ϵ), $1 - \frac{r}{d}$, $1 - \frac{r(1+\epsilon),}{d}$) –sensitive
This conclusion is obtained from:
http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf
There is a technique called Bit Sampling LSH. The main idea is that we don't need to know the exact value of each hash, but only that the hashes are equal at their respective positions in each hash vector. With this insight, let's only look at the least significant bit (LSB) of each hash value.



Hamming LSH

Suppose we have $P = \{\mathbf{p}_i\}$ where $\mathbf{p}_i \in H^d = \{0,1\}^d$ - i.t. points are in the (binary) Hamming Space of dimensionality d (or Cd, in C chunks of size d)

Hash Functions

- sample L subsets $I_1, \ldots, I_L$ of size k uniformly (with replacement) from \{1 \ ... \ d \}$
- let $g_j(\mathbf{p})$ be the projection of $\mathbf{p}$ on I_j: it selects coordinate positions per I_j and concatenates bits on these positions

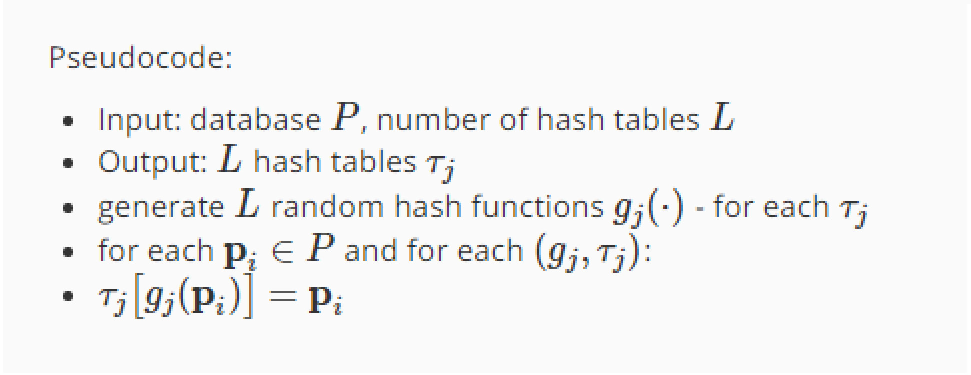
Indexing

Preprocessing (indexing):

- store each $\mathbf{p} \in P$ in the bucket $g_j(\mathbf{p})$ for all $j = 1..L$
- total number of resulting buckets may be large ($g_j(\mathbf{p})$'s are sparse), so reduce the desparsify and reduce dimensionality using usual hashing

So have two levels of hashing:

- LSH Hash to map $\mathbf{p}$ to bucket $g_j(\mathbf{p})$
- standard hash function to map $g_j(\mathbf{p})$ to a hash table of size M



Pseudocode:

- Input: database P, number of hash tables L
- Output: L hash tables τ_j
- generate L random hash functions $g_j(\cdot)$ - for each τ_j
- for each $\mathbf{p}_i \in P$ and for each (g_j, τ_j):
- $\tau_j\left[g_j(\mathbf{p}_i)\right] = \mathbf{p}_i$

(Refers to http://mlwiki.org/index.php/Bit_Sampling_LSH)
Based on Bit Sampling LSH, if we design a data structure LshTable to index the bit vectors to reduce the number of bitSimilarity comparisons drastically (but increase memory

consumption) , by building multiple hash tables on binary code substrings, we can do exact K-nearest neighbor search in Hamming space and at the same time to try to make the hash to achieve one-to-one.
http://www.cs.toronto.edu/~norouzi/research/papers/multi_index_hashing.pdf
Binary codes (an array of 10 bits) are indexed m times into m different hash tables, based on m disjoint binary substrings. 10 bits is partitioned into m disjoint substrings, $h^{(1)}$, . . . , $h^{(m)}$, For convenience, we assume that b is divisible by m. The key idea rests on the following proposition: When two binary codes h and g differ by r bits or less, then, in at least one of their m substrings they must differ by at most r/m bits.
Given a dataset, one hash table is built for each of the m substrings of the codes. For a query q with substrings $\{q^{(i)}\}^{m}_{i=1}$, we search the i th substring hash table for entries that are within a Hamming distance r/m of $q^{(i)}$, thereby retrieving a set of candidates, denoted $N_i(q)$. According to the above proposition, the union of the m sets, $N(q) = \cup_i N_i(q)$, is necessarily a superset of the r-neighbors of q. The last step of the algorithm computes the Hamming distance between q and each candidate in N (q), retaining only those codes that are true r-neighbors of q.