Comp 480/580: Assignment #1 (80 points)

Rice University — Due Date: Tuesday, 09/30/2025

1 Testing Hash Functions 20 Points

Avalanche Analysis: A common test of hash function performance is whether or not it achieves "avalanche." This refers to the desirable characteristic that

$$P(Output\ bit\ i\ changes \mid Input\ bit\ j\ changes) = 0.5\ \ \forall i,\ j$$

We are going to analyze three hash functions. Lets fix the range to 10 bits, and $P = 1048573$ Pick, a,b,c,d uniformly between $[1 - 1048573]$, and store them. (you need to report the numbers you generated) Your hash functions $h(x)$ are
1. $((ax + b)$ mod P) mod 1024 (2-universal)
2. $((ax^2 + bx + c)$ mod P) mod 1024 (3-universal)
3. $((ax^3 + bx^2 + cx + d)$ mod P) mod 1024 (4-universal)
4. murmurhash3 with a fixed seed (use murmurhash from sklearn.utils import murmurhash3_32. Feel free to take extra mod to constrain it in range [0-1023])

Randomly generate 5000 positive integers 31-bit keys (values of x) to start with. Design your experiment to estimate the probability values (you need to flip bits now).

Create a 2 dimensional heatmap : On x-axis, we have bits in the input. For every input bit j, we have to calculate 10 values, which is the probability

$$P(Output\ bit\ i\ changes \mid Input\ bit\ j\ changes) = 0.5\ \ \forall i,\ j$$

Find the most convenient way to have this heatmap. (value of 0.5 should be dark for best visualization).

Write a paragraph on the plot and your conclusion.

1.1 What to Submit:

Submit codes, plots, report, makefile (if any) and a main script to run the code that generate the heatmap. Submit one compressed file via Canvas. We will only type the scripts in the command line, and it should generate all the outputs. Make sure you fix the random seeds so that multiple runs of the code produces the same output.

2 Counting Turtle Confidence 20 points

Refer to the notes `https://www.cs.rice.edu/~as143/COMP480_580_Spring21/lect1.pdf`

We showed that

$$E(M) = \frac{k_1 k_2}{n} \text{ OR } n = \frac{k_1 k_2}{E(M)}$$

So now, let us say we repeat the experiment R-times independently and generate R measurements $M_1, M_2, M_2, ...M_R$. We report

$$\hat{n} = \frac{k_1 k_2}{\frac{1}{R}\sum_{i=1}^{R} M_i}$$

as our estimator of n. The goal is to find a high probability confidence interval for $\hat{n}$. A good idea is to find confidence interval for $\frac{1}{R}\sum_{i=1}^{R} M_i$, i.e. we want to say with probability $1-\delta$, $\frac{1}{R}\sum_{i=1}^{R} M_i \in [E(M)-a, E(M)+a]$, here a is a function of δ (uncertainty), variance $Var(M)$, and of course R (more repetitions are sharper).

- Can you find an interesting a (use Chebyshev). It could be of the form $contant \times std$ (where std is the standard deviation or square root of some variance) 15 points

- Comment on how you will use the formula to find the number of repetitions to get to a particular fraction f of error, compared to $E(M)$, with a probability of failure less than 0.05. 2 points

- How does the above translate into an interval for $\hat{n}$. Can we argue some cases where estimation is really hard. 3 points

3 Inequalities 20 points

It is very hard to prove completely that linear probing is O(1) expected cost, but we will prove it assuming a theorem, to highlight why we need higher level of independence, in this case at lease 5-independent hash functions. Prove that linear probing has O(1) expected cost for searching with load factor $\alpha = \frac{m}{n} = 1/3$.

Assume that for linear probing (someone proved that) the expression for expected cost of searching is bounded by $E(cost\ of\ searching) = O(1)\sum_{i=1}^{\log n} 2^s \times Pr[B_s \geq 2 * E[B_s])$, where B_s is a random variable that denotes the number of elements (out of the total m elements inserted) that maps into a given region (the arc) of size 2^s, using the 5-independent hash function.

Can you now prove that this bound implies that the expected cost of searching is at most a constant?

Hint 1: Use 4^{th} Moment Bound:

$$Pr(|B_s - E[B_s]| \geq a) \leq \frac{4thMoment(B_s)}{a^4},$$

where

$$4thMoment(B_s) = E[(B_s - E(B_s))^4]$$

Hint 2: 5-Independence implies, with X_i as defined in class, that $E[X_iX_jX_kX_l] = E[X_i]E[X_j]E[X_k]E[X_l]$ for any quadruples $[X_i,\ X_j,\ X_k,\ X_l]$

4 Implement and Test Bloom Filters 40 points

Build your own Bloom Filter

The instructions are given in python for demonstration. If using any other language, feel free to choose equivalent alternatives. Never hesitate to ask question, when in doubt

In class we saw the basic Bloom filter, where we used k independent random hash-functions $h_1, h_2, ..., h_k$ to hash a set S of N elements into an array A of R bits. Recall that the formulas for computing the best k to use for a given key set size M and maximum false positive rate at given range,

$$k = \frac{R}{N} ln2 \tag{1}$$

with false positive rate

$$fp = 0.618^{\frac{R}{N}} \tag{2}$$

4.1 Warmup Tasks

- 1. Implement a simple hash function factory. Given an argument m, the desired hash table size, the factory should return a hash function that maps integers into a table of that length. (Or use murmurhash from sklearn.utils import murmurhash3_32)

- 2. Implement a Bloom filter as a class. A Bloom filter's primary constructor receives two arguments: the desired false positive rate, c, and the expected number of keys to store, n. You should not use arrays but instead use bitmaps. You can make use of an example of bitarrays in python from here **url** and make sure that your range is power of 2.
- 3. You will generate two dataset: a membership set: a list of 10,000 unique integers selected randomly from the range $[10000..99999]$, and a test set: a list of 1000 unique integers not in the membership set and 1000 selected randomly from the membership test. Demonstrate your Bloom filter for each of these false positive rates:
  0.01
  0.001
  0.0001
  In all cases, insert the items in the membership set into a Bloom filter using its primary constructor. Then look up all the items in the test and compute the false positive rate.

4.1.1 Deliverables

BF.py
```
• def hashfunc(m):
  ...

  class BloomFilter():
          def __init__(self, n, fp_rate):
          ...

      def insert(self, key):
          ...

      def test(self, key):
          ...
```

Results.txt
```
• Theoretical FP                          Real FP
  0.01
  0.001
  0.0001
```

4.2 Extended Practices

- Download Aol dataset which includes anonymized user ids and click data. **(url)**
- Data prepossessing: get the unique urls from the data file.

```
import csv
import pandas as pd

data = pd.read_csv('user-ct-test-collection-01.txt', sep="\t")
urllist = data.ClickURL.dropna().unique()
```

- Treat urllist as the membership set and add them to your bloom filter object.

- Sample 1000 urls from urllist as the test set and 1000 random strings as false urls. Report the false positive rate and memory usage of your design of bloom filter. Plot the false positive rate with memory by varying R. (use k=0.7R/N, N=377871 in this dataset).(memory usage of a object can be check with sys.getsizeof()).
- Insert the membership set to a python hashtable and compare the memory usage of it and that of the bloom filter. Also estimate the size of bloom filters using theoretical bit calculations and compare it with returned by code. Comment on findings.

In less than a page report, describe your conclusions.

4.3 What to Submit:

Submit codes, plots, report, makefile (if any) and a main script to run the code that generate the plots. Submit one compressed file via Canvas. We will only type the scripts in the command line, and it should generate all the outputs. Make sure you fix the random seeds so that multiple runs of the code produces the same output.