CS358: Applied Algorithms

Assignment 4: Locality-Sensitive Hashing (due 10/30/25 10PM)

Instructor: Sam McCauley

Instructions

Code can be submitted by committing and pushing them to gitlab. I strongly suggest that you access `evolene.cs.williams.edu` through a web browser to make sure everything was uploaded as you expected. You may collaborate on code with your classmates, as well as the instructor and TA, but you may not use any LLM assistance. Please contact me at srm2@williams.edu if you have any questions or find any problems with the assignment materials.

Problem Description

In this problem, we will be trying to find the closest pair of items in a large, high-dimensional dataset.

In particular, you will receive as input a large number of random 128 bit numbers (each broken up into four 32-bit chunks). There will also be a single planted pair of numbers, which are close in terms of Jaccard similarity. The goal of your program is to return the index of these items.

The Jaccard similarity is a set similarity measure. In the context of bit strings `a` and `b`, the Jaccard similarity can be defined (using C notation for bitwise operations & and `|`) as

$$\frac{\text{number of bits in } \texttt{a} \ \& \ \texttt{b}}{\text{number of bits in } \texttt{a} \ \texttt{|} \ \texttt{b}}.$$

In this assignment, you will use locality-sensitive hashing to efficiently find the pair of items in the list with similarity .8 or greater.

INPUT: `test.out` is given two arguments; each is a text file containing any number of problem instances. A problem instance begins with three numbers on a line. The first number represents the size of the instance; this is equal to the number of subsequent lines in the file that are a part of this instance. The next two numbers indicate the two indices of the close pair (these indices assume the array is 0-indexed).

Each of the following lines represents a 128 bit number. Each line consists of four signed 32-bit integers, separated by a space. Concatenating the bits representing these integers results in a single signed 128 bit number. It may be useful to represent each number as an array of four 32 bit integers, or as two 64 bit integers in a struct (this is what I used, and this is how the data will be passed to your function). It may be possible to store the number in a single 128 bit data type, but I have not experimented with this.

In each problem instance, exactly one pair of numbers has similarity .8 or greater.

After all inputs are completed, the file may end, or another problem instance may be immediately concatenated onto the end. For example, `largeInput.txt` contains 8 problem instances.

The functionality in `test.c` will read the file, and store each number in an array (in order) of objects of type `Item`. A `Item` is a struct containing two unsigned 64-bit integers. `test.c` will, for each problem instance, call the function `find_close(Item* input, int length)`—the arguments to this function are the array of `Item`s, and the length of the array.

I have included two files for testing: `simpleInput.txt` and `largeInput.txt`. Unfortunately, we have reached the lab where "big data" is beginning to get a bit annoying: `largeInput.txt` is nearly 300MB, and cannot be stored in a github repo. Therefore, this input is available in two places: on the website, and in my scratch drive where you can access it from the lab computers (the location of the file is given in the example below).

Be sure to not add `largeInput.txt` to your git repo! Git does not handle large files well and it is a pain to remove. Your `.gitignore` file contains `largeInput.txt`, which should mean that it should not be added unless you override the ignore file.

You may also generate your own input; `largeInput.txt` was generated using 14 instances of size 500,000, so doing the same should result in an almost-identical test file without any downloads required. (Make sure you don't commit those either.)

A simple run of the program can proceed as follows, accessing the version of `largeInput.txt` on my scratch drive (this should work as-is if you run it on a lab computer, from your account):

```
./test.out simpleInput.txt /home/scratch/srm2/Assignment4/largeInput.txt
```

or, if you have `largeInput.txt` stored locally:

```
./test.out simpleInput.txt largeInput.txt
```

Output: The function should output the indices of the close pair of elements, i.e. the pair with similarity `.8` or greater. To make these easier to pass around, we assume that these indices are 32-bit numbers, and concatenate them together to create one 64-bit number to pass back to the calling function. That is to say: the return value of function `find_close` is an unsigned 64-bit number where the first 32 bits represent one index of the close pair, and the last 32 bits represent the other index of the close pair. **The order of the solution pair does not matter!**

Note on Output Times: This assignment is likely to take a bit longer to run than previous assignments—a relatively simple implementation seems to take 20–120 seconds to solve `largeInput.txt`.

Questions

Implementation 1. Implement MinHash to find the most similar pair of items, as described above. You do not need to describe your implementation.

Implementation 2 (Extra Credit)**.** Optimize the code so that it consistenly (say, roughly half the time) runs in 10 seconds or less. Some suggestions:

- Experiment to find an effective value for k
- How are you storing your buckets? Consider storing them in a way with less overhead. (In fact, with a little preprocessing we can store all buckets in a single array without any metadata at all.)
- Make sure the compiler flags ensure that your code is optimized.
- It may be that unusually large buckets are slowing you down. How can you avoid this? (There are many good answers to this question.)

If you choose to do this extra credit, please briefly explain your optimizations below.

Solution. □

Problem 1. In class, we calculated R, the number of expected repetitions to find the close pair.

Let's say you have a dataset of n sets that may or may not have a close pair in it. You want to make sure that you don't loop infinitely, so you place a bound on the number of repetitions. Let's call it R_{MAX}.

Let j_1 be the similarity of the close pair, and j_2 be the similarity of all other pairs. In class, we saw that since $k = \log_{1/j_2} n$, the expected number of repetitions is $R = O(n^{\log_{1/j_2}(1/j_1)})$. In fact, we can show a similar result: after R' repetitions, we find the close pair with probability exactly $1/2$, with $R' = O(n^{\log_{1/j_2}(1/j_1)})$.

What should we set R_{MAX} to be so that we find a close pair (if it exists) with high probability? I am looking for an asymptotic answer, i.e. big-O notation.

Hint: You want to use "basic" probability calculations here—by which I mean not union bound, linearity of expectation, or anything like that. Start with R' and use it to help you obtain R_{MAX}.

Solution. □

Problem 2. Assume we have an instance of n items. The Jaccard similarity between any two of these items is exactly .25, except for one pair which has similarity exactly .5.

Let's say you have a working implementation; this question asks how perturbations

in your implementation are likely to change its behavior. Let k be the number of hash functions you concatenate in your implementation to obtain the final hash of your element (note that this variable just stores the number of concatenated hashes in *an implementation*: it may not be $\log_4 n$, or any other particular function of n and the similarities).

Assume that with k concatenated hashes, the expected size of each hash bucket is B. Furthermore, as above, let R' be a fixed number of repetitions, where exactly $1/2$ of the time, your implementation finds the close pair in R' or fewer repetitions.

(a) Let's say we (again) increase the number of concatenations in your hash function: you concatenate $k' = k+1$ hash functions instead of k. How does this affect the expected size of each bucket (i.e. if B' is the expected size of each bucket when concatenating k' hashes, what is the relationship between B' and B)? Please briefly justify your answer.

Solution. □

(b) Let's say we increase the number of concatenations in your hash function: you concatenate $k' = k + 1$ hash functions instead of k. What fraction of the time will your implementation now find the closest pair after R' repetitions? Please briefly justify your answer.

Hint for (b): First, write out what we're looking for as a function of R' and k as in part (a). This will be worth substantial partial credit. For full credit, use $(1-1/x)^x = 1/e$ (you can assume they are exactly equal) to obtain a number—i.e. a single constant rather than an equation—for the answer.

Solution. □