CS358: Applied Algorithms

Assignment 5: Locality-Sensitive Hashing (due 4/30/2020 10PM EDT)

Instructor: Sam McCauley

Instructions

All submissions are to be done through github. This process is detailed in the handout "Handing In Assignments" on the course website. Answers to the questions below should be submitted by editing this document. All places where you are expected to fill in solution are marked in comments with "FILL IN."

Please contact me at sam@cs.williams.edu if you have any questions or find any problems with the assignment materials.

We will have a leaderboard for this assignment. This will allow some automatic testing and feedback of your code, as well as giving some opportunity for friendly competition and extra credit.

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 (starting with 0) of the close pair.

Each of the following lines represent 128 bit numbers. 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). It may even be possible to keep them as 128 bit numbers—I have not experiemented with this.

In each problem instance, exactly one pair of numbers has similarity .8 or greater. Please email me if you find that this is not the case.

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 `Pair`. A `Pair` is a struct containing two unsigned 64-bit integers. `test.c` will, for each problem instance, call the function `find_close(Pair* input, int length}`—the arguments to this function are the array of `Pair`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 over 100MB. Therefore, I have put `largeInput.txt` on the website (in compressed format); you should download it separately to your repository. I have put `largeInput.txt` into your .gitignore file so that you do not download or upload it accidentally. You may also generate your own input; `largeInput.txt` was generated using 8 instances of size 300,000, so doing the same should result in an almost-identical test file without any downloads required.

A simple run of the program can proceed as follows:

```
./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! The functions in `test.c` will try both orders when determining if your answer is correct.

Generating new inputs: I have included a file `generateNew.c` that can be compiled and run to generate a new, random instance of the problem meeting the above requirements. It takes a simple command-line argument, which is the length of the new instance. Feel free to use this to generate tests for your code. It outputs the instance to the standard output, so you should redirect the output to a text file in order to be able to use it.

Note that because of the way this works, this program will, sometimes, fail to generate an instance.[1] It will output a warning to stderr—which mean that you'll still see it if you redirect output to a file. It won't output anything else (and won't write anything to the file) in this case. In general, this program was hacked together and should probably not be taken as an example of quality code.

[1] This has to do with how it generates the "close" pair of items, while ensuring that each item still looks random and does not have any other unusual qualities (for example, the number of 1s in each close-pair item should be similar to the number of 1s in any other item).

Here is one example of how to compile the code, then run it twice to generate two new instances, each of size 10:

```
gcc -o generateNew.out generateNew.cpp
./generateNew.out 10 > newInstance.txt
./generateNew.out 10 >> newInstance.txt
```

Note the `>>` in the second run; this means that the output should be *appended* to `newInstance.txt` rather than overwriting it.

OUTPUT TIMES: The inherent randomness in this assignment means that execution times are likely to be very inconsistent. This has a few effects. First, this assignment is likely to take a bit longer to run than previous assignments—a relatively simple implementation seems to take 5-12 seconds to solve `largeInput.txt`.

Second, "ties" will be considered more generously for this assignment. **Two submissions will be considered "tied" if they their final running times are within one second of each other.**

Third, it is likely that there will be some sense of "average" running time implemented in the testing script. The details of this will be announced soon.

Finally, please bear in mind that your best running time given by the testing scripts is likely to decrease somewhat significantly, as repeated runs mean you get more lucky instances. I will try to mitigate this when possible when assigning final grades, and I believe that the 3-4 tests run on the last day will help with this as well. But, as before, this is an incentive to submit fairly early if you are interested in getting the extra credit points.

Questions

Code (80 points). Your code is worth 80 points. (This is a bit more in previous weeks, mostly because this assignment seems to require a bit more to get working.)

Problem 1 (10 points)**.** Please describe your implementation at a high level. What techniques and data structures did you use (i.e. how did you store the buckets)? What optimizations did you use?

Solution.

Problem 2 (10 points)**.** 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. Assume that, currently, you concatenate k hash functions to obtain the final hash of your element. You find that 1/2 of the time, your implementation finds the close pair in R or fewer repetitions.

(a) Let's say we 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? Please briefly justify your answer.

(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.

Solution.