CS358: Applied Algorithms

Assignment 5: Locality-Sensitive Hashing (due 10/27/21 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 srm2@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. Leaderboard testing will start Friday morning.

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.

Your code is **required to use randomness**. That is to say, each run of the algorithm should use a different permutation for the MinHashes (you may not just hard code manually-selected arrays as was done for h in Assignment 3). In your starter code you are given an array-shuffling function `shuffle_array()` that satisfies this requirement (so long as your submitted version uses `srand(time(NULL))` and not `srand(HASH_SEED)`).

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 (performing SIMD operations on them directly), but I have not experimented 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 `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 over 100MB, and cannot be stored in a github repo. Therefore, your repo will contain a file `largeInput.zip`; uncompressing it will give `largeInput.txt`.

I have put `largeInput.txt` into your .gitignore file so that you do not add it to git 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. (Make sure you don't commit those either.)

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.

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 .4 seconds of each other.**

Third, each testing day, your score for the leaderboard will be calculated as the median of three runs.

Finally, please bear in mind that your best running time given by the testing scripts is likely to decrease somewhat significantly, as (even with the above) repeated runs mean you get more lucky instances. As before, this is an incentive to submit fairly early if you are interested in getting the extra credit points.

Questions

Code (50 points). Implement MinHash to find the most similar pair of items, as described above. You do not need to describe your implementation.

Problem 1 (20 points)**.** Most of the time in your implementation is spent on finding the similarity between all pairs of items in a bucket. Implement a way to find these similarities using SIMD instructions for buckets of size > 2. (For buckets of size 2 you can use the normal method.)

Your code should work both with and without SIMD instructions. Near the beginning of `minHash.c` there is a constant `USE_SIMD`; your implementation should not use SIMD instructions if this is 0, and should use them if it is 1.

[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).

Specifically, your method will need to use the following intrinsics to calculate the Jaccard similarity between elements:

```
c = _mm256_and_si256(a, b);
c = _mm256_or_si256(a, b);
```

Where the above function calls take the AND (respectively OR) of two `__m256i` variables `a` and `b` and store the result in an `__mm256i` variable `c`.

You also need to calculate the number of 1s in the result (after you take the AND and OR between the elements). Unfortunately, `AVX2` does not have a way to calculate the number of 1s in a vector. Instead, use `__builtin_popcount()` to calculate the number of 1s, 64 bits at a time.

Is your code faster with the SIMD instructions? Test it with and without the SIMD instructions on the given data; write both times below.

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. 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, 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 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.

(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.

Problem 3 (10 points)**.** 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 the expected number of repetitions is $R = O(n^{\log_{1/j_2}(1/j_1)})$. 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 (not union bound, nor the bound from Assignment 4 Problem 1).

Solution.

Problem 4 (10 points)**.** In class I mentioned that you may not need to store the entire permutation for MinHash. In particular, if you want $\hat{k}$ hash values from each permutation, storing $O(\hat{k})$ (for a large enough constant) is likely to be enough. (To emphasize: this is optional; you can get full credit on the coding portion while storing each permutation entirely.)

Let's say $\hat{k} = 2$, and you store the first 16 items from each hash.

Consider the data from this assignment: each item has 128 bits, and each bit is a 1 with probability $\frac{1}{2}$. For a given item, what is the probability that after looking at 16 bits (randomly selected from the permutation) that fewer than 2 of them will be 1?

Solution.