Programming Test for Engineering Candidates

MatrixOrigin Technologies.

June 8, 2021

Contents

1 Overview

There are two problems in this test. The first problem contains two subproblems, while the other one is an independent problem itself. For each problem, please provide source code files and a makefile. All programs should compile in Linux at the assigned test servers. You must use C++ programming language to solve the problems. You can use standard C++ libraries. No other third party code is allowed in this test. During the test, you can use your books, use Internet for research, or any other means to come up with proper solutions. However, you should not discuss or share any code, ideas or specific solutions of any problems in this test with other contestants. Any such activities will entirely disqualify the solutions of the involved.

2 Evaluation

Each problem in this test has a maximum score you can get. Here is how we grade your solution. First, if your program compiles correctly by simply typing make at a prompt in any directory after extracting your solution files, and produces expected output without a bug, you get the maximum score on correctness. If your Makefile does not work, you get 0 point on that problem. Please make sure that your Makefile works without a fail!

Secondly, after your program passes the correctness, it would be evaluated on performance. That is, we will measure the average time it takes to execute on random, fairly large input files. If there are N number of contestants that passed correctness test, their solutions will be sorted in terms of their speed according to their average time performance. The first ranked solution will get the maximum performance score allocated, and the last ranked solution will get 0 point, while the rest will get corresponding scores according to their ranks linearly decreasing from maximum performance score to 0. If your program fails on the correctness test, you automatically get 0 point on the performance evaluation.

Also please note that there is a sample test input and output files. If your program passes this basic test case provided, you will get 10% of the problem maximum score, even though your program fails on other more complex test cases.

Because the test involves a lot of reading and design within a limited amount of time, there is no requirement for documentation of your source code files or putting together a readme file. The evaluation is strictly based on the correctness and speed of your program.

3 Submission

Please use Linux tar command to archive your source codes and Makefile. No object files or executables should be included in the archive. An archive name should be firstname.lastname.tar. If you name your archive name anything other than firstname.lastname.tar, you will get 10 points deduction!

Please submit your solution (via email to yingfeng.zhang@matrixorigin.cn or yingfeng.zhang@gmail.com). Please note that you can submit your solution only once. Any subsequent submissions will be ignored.

4 Probabilistic Simhash Matching

In this test, you need to implement a near-duplicate detection system. The approach you have to use is based on the well-known Simhash. Some details of the algorithm is described below. You should also read the related paper "Probabilistic Simhash Matching" to get further details, while it would suffice reading material below.

4.1 Simhash and Hamming distance

Simhash was developed by Moses Charikar and is described in his paper "Similarity Estimation Techniques from Rounding Algorithms". The main purpose of Simhash is that similiar items are hashed to similiar hash values. The similarity can be judged by the bitwise Hamming distance between hash values.

Below is the algorithm to map an n-dimensional vector with real elements to an f-bit Simhash.

Algorithm 1 Calculating Simhash for a real vector

Input: $\mathbf{v}[1..n]$: input real vector
Output: s: f-bit Simhash

```
initialize a real vector t[1..f] to be 0
generate n f-bit random integers r[1..n]
for i ← 1 to n do
    for j ← 1 to f do
        if the j-th bit of r[i] is 1 then
            t[j] ← t[j] + v[i]
        else
            t[j] ← t[j] − v[i]
        end if
    end for
end for
for i ← 1 to f do
    if t[i] ≥ 0 then
        set the i-th bit of s to 1
    else
        set the i-th bit of s to 0
    end if
end for
```

4.2 Algorithm

4.2.1 Near-duplicate detection

In general, we can consider two Simhashes as near-duplicates if their Hamming distance is not more than h, where h is an empirical threshold. To find all near-duplicates for a Simhash among a database of m Simhashes, the most trivial approach is to do a full linear scan which needs m times of Hamming distance calculations.

The time cost of a full linear scan will become intolerable while m growing. In fact, most of the Hamming distance calculations are unnecessary since only a few near-duplicates can be found for each queried Simhash. Several solutions have been proposed to reduce the time of comparisons, such as Manku et al, 2007.

4.2.2 Bit-flip combinations

In this programming test, we consider a new method proposed by Sood and Luguinov, the Probabilistic Simhash Matching (PSM) system which does not need to maintain multiple copies of data for near-duplicate search while still maintaining a good recall percentage. For a given query fingerprint, we explore different existing fingerprints based on the probability of finding a near-duplicate match and limit our number of attempts to achieve good query speed.

First, we can make the Simhash database sorted and build a index. To better understanding the "index" here, suggesting that all the Simhashes are f-bit integers and sorted in a list, we split them into sublists according to the first p bits of each Simhash, then all the positions of first elements in each sublist compose an index. For example, the 32767-th element of the index indicates the position of the least element whose first p bits equals to `0x7fff` in the Simhash list. Since the Hamming distance threshold h is often empirically 2 or 3, we can easily build the bit-flip combinations which means all the possible first 16-bits of a near-duplicate to the queried Simhash. For $h = 3$ and $p = 16$ index, we need to scan for all the 65536 (2^{16}) sublists in a full linear scan, but only $C_p^0 + C_p^1 + C_p^2 + C_p^3 = 697$ sublists benefiting from the bit-flip combinations.

In the next section, we would like to show that the number of sublists to be scanned can be further reduced by determining the weak bits, while still retaining a high recall.

4.2.3 Bit-flip probability

To compute a f-bit simhash, the intermediate step is to compute a f dimension weight vector $\mathbf{W_d}$, each of whose dimension is initialized to zero. The hash function θ is applied on each feature $t_{i,d}$ iteratively and $w_{i,d}$ is added

Feature	Hash	weight				
$word_1$	0101	0.05	-0.05	$+0.05$	-0.05	$+0.05$
$word_2$	1101	0.02	$+0.02$	$+0.02$	-0.02	$+0.02$
$word_3$	0001	0.01	-0.01	-0.01	-0.01	$+0.01$
$word_4$	1110	0.03	$+0.03$	$+0.03$	$+0.03$	-0.03
$word_5$	0100	0.05	-0.05	$+0.05$	-0.05	-0.05
$word_6$	0011	0.09	-0.09	-0.09	$+0.09$	$+0.09$
$\sum weight$			-0.15	$+0.05$	-0.01	$+0.09$
`simhash`			0	1	0	1

Table 1: An example computation of simhash on a document containing six words using four bit hash values.

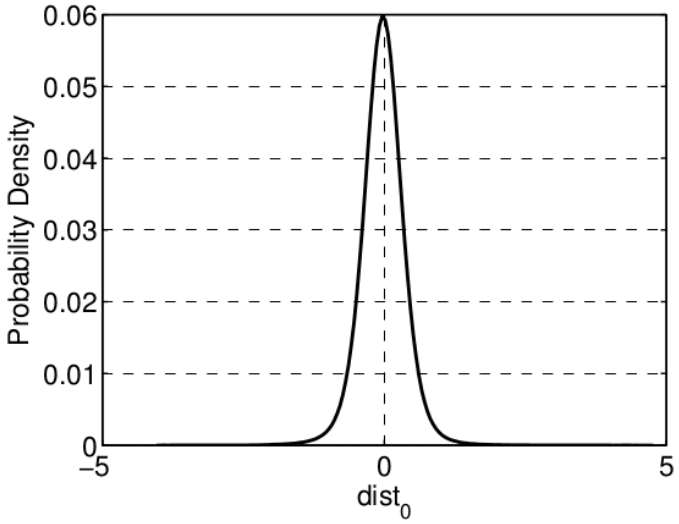

Figure 1: Distribution of $dist_j$ calculated from 45M document pairs for $j = 0$.

or subtracted from each dimension of $\mathbf{W_d}$ depending on the bit values of hash $\theta(t_{i,d})$ as follows: if the j-th bit of hash is 1, then j-th dimension of $\mathbf{W_d}$ is incremented by $w_{i,d}$ else j-th dimension of $\mathbf{W_d}$ is decremented by $w_{j,d}$. When all n weights have been added/subtracted based on the bits at specific index for each feature hash, the intermediate vector $\mathbf{W_d}$ has been generated.

For two near-duplicate documents d and d' , the distance $dist_{j,d-d'}$ between their weight vectors $\mathbf{W_d}$ and $\mathbf{W_{d'}}$ in j-th dimension can be given:

$$W_{j,d'} = W_{j,d} + dist_{j,d-d'} \tag{4.1}$$

The probability that q' flips the j-th bit of q is directly dependent on how far away $dist_{j,d-d'}$ is from $W_{j,d}$ on the number line. We can prove this

by plotting the distribution of $dist_{j,d-d'}$ which intuitively should follow a normal distribution with mean 0. Figure 1 shows the plot of $dist_0$ for a random sample of 45M document pairs which proves the correctness of our assumption. We can say that the probability of a bit-flip is greater for a bit whose weight $W_{j,d}$ is closer to 0. Mathematically, the probability of bit-flip for j-th bit ($j \in [1, f]$) can be estimated as:

$$Pr(b_j(f) \neq b_j(f')) = 1 - \left| \frac{W_{j,d}}{|\mathbf{W_d}|} \right| \tag{4.2}$$

where we divide every weight component $W_{j,d}$ by size of the vector $|\mathbf{W_d}|$ to get a value between 0 and 1.

Now we can order the bits by the absolute value of $W_{j,d}$. For the example in Table 1, we can sort the bits in order $(3, 2, 4, 1)$, as $W_{3,d}$ has the least absolute value 0.01 and $W_{1,d}$ the largest 0.15. With this bit order, we can generate top-k weakest bit-combinations using a method introduced in PSM paper, then further reduce the number of to-be-scanned sublists from $\sum_{i=0}^{h} C_p^i$ to k.

5 Problem A: Online mode of PSM (100 scores)

5.1 Description

In this problem, you should implement the online PSM algorithm. Notice that for image search, the empirical Hamming distance threshold h for near-duplicates can be assumed as **2**. There are 20 individual tests, each of which is *5* scores.

This problem can be divided into following *two* subproblems.

5.1.1 Problem A.1 (50 scores)

In this subproblem, you should implement the index strategy for Simhash matching. This is the basis for PSM. You should first map each *feature* to a 64-bit Simhash, then find near-duplicates out of the *image feature database* for each *feature* in the queried *feature* list.

Your score will be evaluated by the performance. For each test point, if the time cost of our sample solution is t_{judge}, your program costs $t_{contestant}$ (in average), your score will be evaluated as

$$\frac{score}{score_0} = max\{0,\ 1 - 0.2 \times \lg \frac{t_{contestant}}{t_{judge}}\}$$

in which $score_0 = 5$ is the full score for each test point.

5.1.2 Problem A.2 (50 scores)

In this subproblem, you should complete your PSM algorithm. The score is evaluated according to the performance improvement and recall rate against your `Problem A.1` program.

The score for this subproblem is related to your recall rate and performance improvement against your program for `Problem A.1`. Assuming that your recall rate is r_p and your time costs are $t_{non-psm}$ and t_{psm} respectively, then your score will be evaluated like this:

$$\frac{score}{score_0} = (1 - 0.1 \times (3 - \log_2 \frac{t_{non-psm}}{t_{psm}})) \times \min\{1,\ 1 - 5 \times (0.95 - r_p)\}$$

where $score_0 = 5$.

Warning: you will get a *0* score for a test point if your PSM algorithm detects any near-duplicates that is not in your result of `Problem A.1`, because a correct PSM can only find a subset of the results of a non-PSM. To ensure this, you should use the same random hashes for these two subproblems.

5.2 Input

The two subproblems share the same input files. You get 20 test points, each of which contains two input files, an *image feature database* and a list of queried *features*. There are 5 samples under the `sample_input` subdirectory, named as `ProblemA_[00-05].dat` and `ProblemA_[00-05].que`.

The *image feature database*: the first line is an integer n, following by n lines each of which consists of a string representing an image file name and 128 float numbers representing a *feature*. Beware that one file name can correspond to multiple *features*.

The queried *feature* list: first line is an integer m, and then m lines each of which is a *feature* (128 float numbers).

Warning: For the ease of performance evaluating, your programs should load and process the *image feature database* before touching the queried *feature* list, then print a timestamp, and print another timestamp at the end of your program. This is because the database loading process probably occupies most of the total time cost while we only care about the performance of query handling. Never forget to print a timestamp before and after processing the queries. There is a function named `print_current_timestamp()` in the given `util/psm-helper.hpp`, to easily print a timestamp.

5.3 Output

For each subproblem, you should output your results into a single file. Corresponding to the 5 sample inputs, there are 5 sample result files named as `ProblemA1_[00-05].out` and `ProblemA2_[00-05].out` under the subdirectory `sample_output`. For each queried *feature*, you should output all the file names of near-duplicates detected by your program, then an extra `'\n'` to separate matches for different queries.

5.4 Sample input and output

For each subproblem, your executable should accept 3 parameters, corresponding to file names of the *image feature database*, the queried *feature* list and your results respectively. During evaluating, we will use command like this:

```
./your_program 00.dat 00.que 00.out
```

and Table 2 shows a sample input and output:

5.5 Hints

- The order of your output must be the same as the queries!
- The `MurmurHash64A()` hash function in `psm-helper.hpp` may help you to map a feature (128-dimensional real vector) to a 64-bit Simhash.

00.dat	00.out
5 foo.jpg 0.1 -0.2 0.3 -0.2 ... *(128 floats)* foo.jpg 0.3 0.4 -0.5 0.1 ... *(128 floats)* foo.jpg 0.4 0.5 -0.6 0.3 ... *(128 floats)* bar.png -0.2 0.7 -0.5 -0.2 ... *(128 floats)* bar.png 0.3 0.5 -0.4 0.2 ... *(128 floats)*	2 foo.jpg bar.png 1 foo.jpg
00.que	
2 0.4 0.3 -0.5 0.1 ... *(128 floats)* 0.1 -0.3 0.2 -0.6 ... *(128 floats)*	

Table 2: Sample input and output format for online PSM

- In our sample PSM solution, it costs 450MB memory and 15 seconds to finish querying *100,000* features from an *image feature database* of size *800,000*. Try your best to beat this performance!

6 Problem B: Batch mode of PSM

6.1 Description

In this problem, you should implement the batch mode introduced in the PSM paper. Your score will not get a penalty if your output has a recall rate of not less than 90%, so please concentrate more on the performance.

In the *image feature database*, each image file corresponds to one or more *feature*. And two images are assumed as near-duplicates if any *feature* in the first image can find a near-duplicate in the second one. Your task is to efficiently detect the near-duplicate pairs in a batch growing *image feature database*.

There are 10 test points for this problem, each of which values 10 scores. For each test point, the fastest program with a recall rate of not less than 90% will get a full score. And the score for other contestants will be evaluated in the following way: assuming that the time cost of the fastest program is $t_{fastest}$ and your cost is t_{yours}, and your recall rate is r_p, then

$$\frac{score}{score_0} = (1 - 0.1 \times \log_2 \frac{t_{yours}}{t_{fastest}}) \times \min\{1,\ 1 - 5 \times (0.9 - r_p)\}$$

where $score_0 = 10$.

6.2 Input

There are multiple input files, each of which is an *image feature database*. Refer to 5.2 for details about the file format.

00.out
...
0.jpg 1.jpg
0.jpg 2.jpg
0.jpg 3.jpg
4.jpg 5.jpg
4.jpg 6.jpg
...

Table 3: Sample output format for batch PSM

And your program should be able to accept parameters of various length. The command will be like this:

```
./your_program 00.in 01.in 02.in ... 00.out
```

There will be no more than **50** parameters in the real test. And beware that the last argument is the output file name.

6.3 Output

Your output should be a single file named `00.out`. This file should contain multiple lines, each of which contains *two* image file names of a near-duplicate pair. For example, if you have 3 near-duplicates for `0.jpg`: `1.jpg`, `2.jpg` and `3.jpg`, and 2 for `4.jpg`: `5.jpg` `6.jpg`, then your output file should be like in Table 3:

6.4 Hints

- Don't forget to find near-duplicate pairs among the first *image feature database*.