Team Project 4: Plagiarism Detection

June 26, 2016

1 Author Introduction

This project is finished individually by Zhao Zishuo in class IIIS-50.

I attended NOIP2013 and got provincial first prize in senior high school. In the first semester in Tsinghua University, I was a student tutor in the course "Fundementals of Programming" and made out problems for exercises and exams. In the second semester, I took part in the "El Dorado" AI competition and won Top-16.

I had been initially in class CST-52 by May 12, 2016, on which I entered IIIS.

2 Project Background

This project is based on Homework 11, about ways to find match strings and detect plagiarism. This project implements the winnowing algorithm, which finds charateristic substrings of documents as fingerprints. To check plagiarism, it is needed to match those fingerprints and check similarity. To make the fingerprints not too "heavy", the KR algorithm is used to transform substrings into hashes, and only record part of them. The simplest idea to take those $i \mod p$, but in some cases, they can be very far from one another, or even nothing at all. The winnowing algorithm improves it to make fingerprints distributed more regularly.

3 Project Introduction

This project is divided into two parts—generator and judge. Far more time is needed to extract fingerprints from a whole document than compare fingerprints. To detect plagiarism, it is frequently necessary to compare a documents with many other ones. Therefore, the time complexity of comparation is more important.

If we generate fingerprints of a single document each time we compare, then much unnecessary time is wasted. So I decided to extract the fingerprints and save them in an *.index* file with the generator, and compare them with the judge. It can save much time in practice, especially when there are a great number of documents to check.

Although I wrote the judge in a simple way and it can only compare one pair of fingerprints, it can be extended easily to meet different needs (to compare many documents pairwise or to compare one with a bunch, etc). Above all, the key point is to generate the fingerprints, and make generation and comparation more efficient.

4 Project Architecture

4.1 Generator

4.1.1 The input of files

This part is in *input.h/cpp*.

To input a document, the base class *source* is used to get data from a string or a file. It removes spaces and turn all letters into lower case, for further process.

This class is like a "string" and we can use *operator* [], *operator* << on it. It is an abstract base class of *cmp_base*, which deals with string matching via different algorithms. In this project, though, KMP is not used due to lack of efficiency.

4.1.2 The implementation of KR Algorithm

This part is in *cmp_kr.h/cpp*.

This algorithm uses the polynomial in the finite field F_p as the hash of a $k-gram$, a substring with length k. The advantage is that it only takes $O(1)$ time to generate a following hash, better than most of other algorithms with that of $O(k)$. The hash of a $k-gram$ of $str[]$ beginning with place c is defined by (in F_p):

$$H(c) = \sum_{i=0}^{k-1} str[c+i] \cdot q^{k-1-i}$$

Then we have

$$\begin{aligned} H(c+1) &= \sum_{i=1}^{k} str[c+i] \cdot q^{k-i} \\ &= q \cdot \left(\left(\sum_{i=0}^{k-1} str[c+i] \cdot q^{k-1-i} \right) - str[c] \cdot q^{k-1} + str[c+i] \right) \\ &= q \left(H(c) - str[c] \cdot q^{k-1} + str[c+i] \right). \end{aligned}$$

Notice that the coefficient q^{k-1} is used every time. As k is fixed in each substring, I stored $-q^{k-1} \mod p$ in the variable $cmp\_kr :: iter\_const$ to improve efficiency.

Here in the project, $p = 19971019 (cmp\_kr :: mod)$, $q = 2333333 (cmp\_kr :: coef)$.

4.1.3 The extraction of fingerprints

This part is in *digest.h/cpp*.

In this part, it is no longer needed to find a pattern string. It is only necessary to get the hash of every $k-gram$ and extract some of them as fingerprints. In the winnowing algorithm, we select the minimum hash in each l-length window, so at least one of every l continuous hashes will be chosen. Therefore, each substring of a size greater than $l-k$ must influence the fingerprint.

In finding the minimum number in each window, the naive method costs $O(nl)$ time. I used a skill to reduce it, but the exact complexity has not been proved. In average (random as hash) cases it is probably $O(n)$, because step (3) only appears about once every $\gtrapprox \frac{l}{2}$ shifts.

(1)Find the minimum hash in the first window, and record the place of it.
(2)Every time the window shifts, if the recorded hash is still in range, then only compare it with the new hash. If the new one is smaller, then update it.
(3)If the recorded hash is out of range, then re-find a minimum in the window.

Because the comparing of fingerprints does not consider the order, the list of chosen hashes is sorted before outputting to reduce the running time of further comparing. It costs $O(n_0 \log n_0)$ time, in which n_0 is the number of hashes, usually much smaller than the size of the file.

In this project, $k = 32 (digest :: hash\_length)$, $l = 96 (digest :: window\_range)$.

4.1.4 The output of fingerprints

This part is in *output.h/cpp*.

In this part, the class *output* is designed to save the fingerprint list into a *.index* file in the binary form. It may be a little easier to use text form, but the binary form saves space. The list is stored in *vector<long long>* for the range of computation, but in fact every hash is an integer in $[0, p)$. Therefore, the function *binary_output* is used to write 4 bytes as a unit each time.

In the *.index* file, the first unit stores the number of hashes, and the following units stores the hashes in the ascending order.

4.2 Judge

This part is very simple, but efficiency is vital especially in multiple-file comparing. So I did not use the OOP design style.

The function *load*() can load a *.index* file and store the data in a list(*vector*<*int*>), and *count*() can get the number of collisions between two lists within $O(n_1 + n_2)$ time. This counting algorithm only works for sorted lists. Otherwise, another algorithm with higher time complexity is needed.

5 Running tests

I choose five pairs of text documents with the size up to 20KB, and the running time of generation and comparation is instantaneous. On the other hand, for a larger file, the I/O efficiency of *iostream* falls behind *stdio*, and the I/O time of the harddisk cannot be ignored, the time efficiency of this program is satisfactory. The size of the fingerprint varies from 6% to 10% of the original document. That is also acceptable.

The first case is two versions of my program for "El Dorado" AI competition (Ver.69 and Ver.152). The result is 53%. Many updates had been done between those versions, but the basic structure and many codes remains unchanged, so it is in the expectation.

The second case is two different Chinese articles written by me. The result is 0%.

The third case is two different reports on the same event. Despite they have a lot of information in common, the result is 0%. It shows the program is not "oversensitive".

The fourth case is two exactly identical articles. The result is 100% as expected.

The fifth case is two identical articles, one with paragraphs messed up and some parts removed. The result is 81%, showing the program cannot be easily "fooled".

In conclusion, this plagiarism detection does work.

6 Conclusion

This algorithm is a good and stable way to detect plagiarism, with little time and space cost. Although I have done it individually, I have tried many skills to improve the efficiency furthermore.

I have also noticed that it may perform worse if the data is purposely designed. For example, if the whole document only consists of a single letter repeated over and over again, or the hashes are exactly in the ascending order, then the fingerprints will be huge and the efficiency improvement I have made will not take effect. But "hash" itself just means randomness, and it is not likely for **a document that makes sense** to behave in this way.

Therefore, it is a good algorithm. I have managed to do it correctly, and as well as I am able to. However, I did not use OOP design style completely for efficiency, nor did I inherit as much as possible from *cmp_kr* to *digest*. To realize this, probably more knowledge about OOP or even Generic Programming is needed.

Although I have tried my best to do this project, I know that there is so much that I still have to learn.