515 Homework 1

Chenrui Xu

September 2021

1 Question 1

We input three variables. They are n,p and q. So the size of input should be $log_2 n + log_2 p + log_2 q$.

2 Question 2

So, if we use the same way we used in the lecture, the first question should how many median points are smaller than or equal to median of medians (MM)? The answer is $n/14$. How many numbers in one single group that are smaller than or equal to the median of the group? It should be 4.

Then, it is not hard for us to find out that there are $4 * (n/14)$ of numbers less than or equal to MM and $4 * (n/14)$ of numbers more than or equal to MM. If we add the low area and the high area together, the set should be the whole data set N.

$$|LOW| >= \frac{4n}{14}$$

$$|HIGH| >= \frac{4n}{14}$$

where $|LOW|$ and $|HIGH|$ denote numbers in the corresponding set. And we can get

$$|LOW|, |HIGH| >= \frac{10n}{14}$$

then, it can be reformed into

$$Max\{|LOW|, |HIGH|\} >= \frac{10n}{14}$$

When it comes to the time complexity formula of this algorithm, it should be

$$T_w(n) = T_w(\frac{n}{7}) + Max\{|LOW|, |HIGH|\} + O(n)$$

$$= T_w(\frac{n}{7}) + T_n(\frac{10n}{14}) + a * n(supposethataisagivenconstantnumber)$$

$$= T_w(\frac{n}{7}) + T_n(\frac{5n}{7}) + a * n$$

Let us guess that there exist a ζ so that

$$T_w(n) = O(n) <= \zeta * n$$

So

$$= T_w(\frac{n}{7}) + T_n(5n/7) + a * n$$

$$<= \zeta * \frac{n}{7} + \zeta * \frac{5n}{7} + a * n$$

$$= \zeta * \frac{6n}{7} + a * n$$

$$<= \zeta * n$$

As 6/7 is smaller than one, if $\zeta >> a$, the formula make sense.

3 Question 3

Step 1: Find one point (we can call it bug1). Draw a circle as bug1 is the center of it and the radius should be 2. If there are some points in the circle, find the closest one record the smallest distance as D. If not, keep D as 2 (D can be seen as a cache to record the smallest distance and the corresponding pair).

Step 2: Find another point bug2 as a center of the circle and D is the radius of the circle. If there are some points in the circle, find the closest one and replace the D with the closest distance. If not, keep D as the original one.

Step 3: Repeat step 2 for the rest of bugs. The running time should also be $O(n^2)$.

Step 4: Return to D (the smallest distance and the corresponding pair).

Some explanation:

1. There will always some pairs of points with the distance smaller than or equal to 2. Why we say that? Here is the idea: If we want to find the upper bond of the minimize distance of all kinds of situations, we should set points as separately as possible. Given the length of one side is n and n points should on the boundaries (so that points can be more separate), the equally distance between each pair of points is $\frac{n}{n-1}$. Since $\frac{n}{n-1}$ will increase when n increases, we consider the worst case, that is n=2. And $\frac{n}{n-1}$ will be 2 then.

2. Why it is linear: The bugs in the circle is limited and it should be independent by n. We can view it as a small constant number. And the only loop that depends on n is Step 3.

3. Why using replacing? The radius of the circle will be smaller and smaller, so the points we need to judge will be less too (Although if we do not use replacing, the running time is still linear). So adding this idea will be better.

4 Question 4

Decide the similarity between two C programs

So for this question, what we need to do is finding out the similarity between the two C programs.

To figure out this question, the first thing that comes out to my mind is that how to qualify the term 'similarity'? What is the definition of similarity? Secondly, what is the purpose of finding the similarity between the two C programs?

You may want to put the definition of things aside as we can search some terms via Google or read some papers on the Internet. The most frequent use of this question is in classes. Those computer science Professors and teaching assistants always spend a lot of time making judgments whether someone in his or her class cheated (copy codes from other classmates or copy codes from the Internet like GitHub, etc). There is another chance that in many companies, people will always copy codes so that they can make achievements of engineer functions and save a bunch of time at the same time. When assembly, those companies will spend a lot of their budget on purchasing hard drive to save codes while in that scenario.

From my point of perspective, I came up with some ideas.

First of all, it can be a similarity of the word text itself. The simplest way or the most violent way that people can say is that we simply input the tokenized text itself into some natural language process models (NLP) (like word2word, BERT, etc).

Or the similarity might be the function similarity. This is quite a subjective method to figure the similarity. The judgment may cut in in the direction of the input and the output of these two c programs to name it.

What is more, there is another possibility that these two code files share some kinds of same structures (like both of them used for loops or if/else judgment). That is if someone copies the code from his or her classmates

Then, it is the time to figure out what is similarity. We take two extreme sides. We will say that these two C programs are 100% same or 0% same. So all possible situations should be in this interval. We will take the quantify number $[0, 1]$ to represent similarity.

If we google online, there are a bunch number of papers talked about the code similarity. Someone just gave methods like $diff()$ in the Linux system and open source moss from Stanford University. Also the most popular method people

will hire is parse tree[1]. It can depart the whole code into several parts based on some key words and main structures.

Here is my method. The method has five mainly parts, including some prepare works, data cleaning methods, modeling part and training data generation part.

4.1 Step 1

In this step, we need to recognize some keywords. When we get one code file, we can extract some keywords like 'class','function','while','for','int', etc. There are two ways to use them. So in the following steps, we need to match some variables and modules based on the location and other information of these keywords.

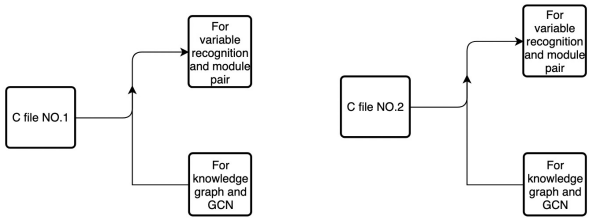

Figure 1: Step 1

On the other hand, based on these words, we will create a new structure of input data: knowledge graph. Nodes in the knowledge graph can be variables, some if/for/while judgment, some self-defined functions and some other classic functions (like 'cout', 'cin', etc).

4.2 Step 2

Based on our purpose, there is the situation that someone copies others code and only change variable names. So in this step, we will get rid of the effect of it. We can get a lot of information from system RAM and environment. We can get how many variables are stored in the RAM based on different time. And also, from Step 1 (keywords recognition), we can figure out what kinds of variables have probability to be the same one.

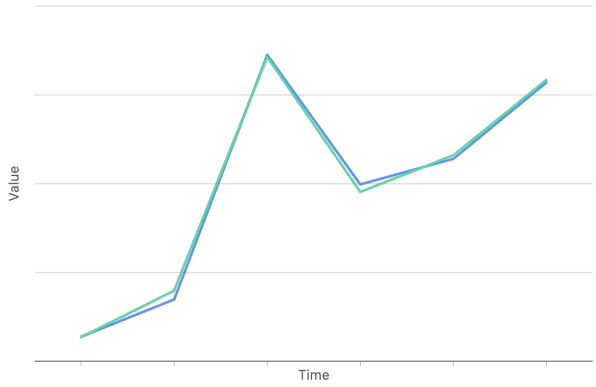

Figure 2: Step 2 Judge variables

In the graph above, we can see that two variables can match well in terms of value and time (data from system log). We may have the conclusion that these variables are actually the same one. Once we matched one pair, we can replace both of them into x instead. And the nth pair of variables will be names after x_n. If there is no variables need to change, we skip this step then.

4.3 Step 3

Based on those key words we found in Step 1, we can divide the whole file into several modules. The module can be a self-defined function, a main function, some casual lines of code (based on cited functions and variables).

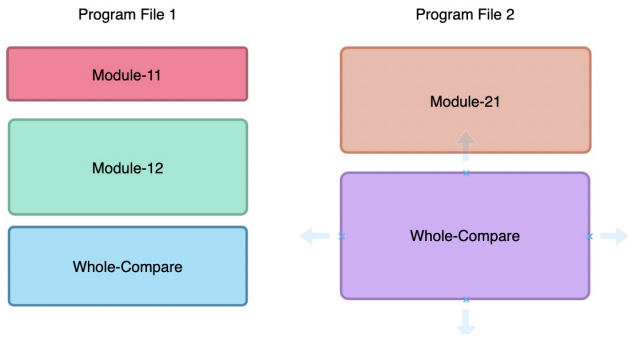

Figure 3: Step 3

Then, we match those modules into pairs. The reason why we need to divide the whole file into parts is, the accuracy to make judgment of the whole file can be very low (whatever kinds of model you want to use). Also, sometimes only

one module or one function will be copied from one to another and we need to figure it out. And the method to match them is that if module one shares some variables and functions with another, they matched. If one module does not match all modules in another file, we will put it into the whole-compare area (the two whole-compare area must be matched). Also, each module, we can get the weight of it based on how many lines it takes. Also, one module in file one can match several modules in file two.

$$w_i = \frac{number \quad of \quad module \quad lines \quad (paired)}{number \quad of \quad all \quad lines \quad (both files)}$$

w_i will be used in the next step to calculate the decision weight.

4.4 Step 4

Here is the model step. There are two main blocks we will use.

We can take a look at the picture shown below. The first part is NLP part and the second part is graph part. The corresponding weight are 70% and 30%. Both parts are matrix learning category. What we want is feature extraction. After that, we can let it connect fully connection layers and sigmoid active function to determine output one or zero. Also, we can let it connect cosine function or kinds of distance calculation methods with a threshold (we can set it to be 0.5).

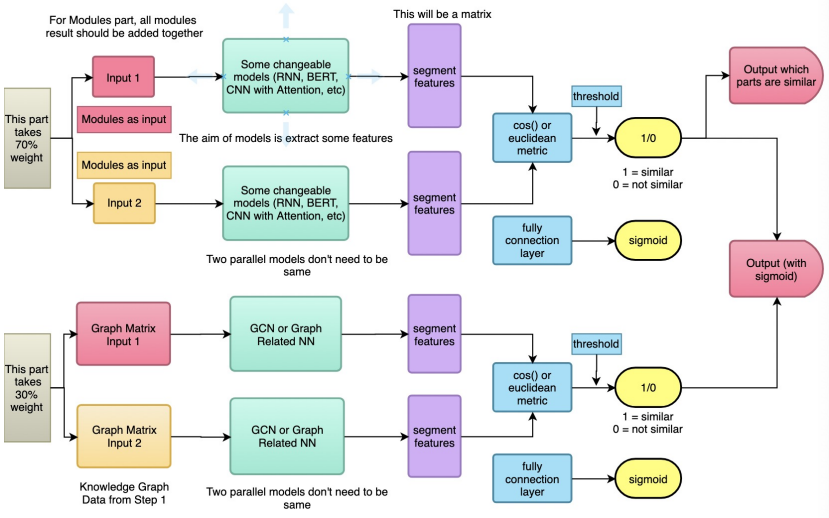

Figure 4: Step 4

While the output is one or zero. We can also get rid of the last step to get a number in the interval from zero to one. We have already calculated weighted for all paired modules. After getting outputs, we add all weighted

results together and time 0.7. Also, at the same time, notice that on the right up corner, there is another output, that is the mechanism of supervising. Once the output is equal to 1, the system will automatically record and report which two parts are similar so that people can track them from the original files.

There might be some probability that the cheater didn't use Object Oriented Programming method. Instead, when he or she was copying, he or she only copy the method instead of structure. That is, there are no functions defined, no classes. The only thing he or she got is a plain programming file. In that case, the mechanism can also detect all parts as the modules can be matched for many times. Then the system will output those matched chunks (already divided into modules).

4.5 Step 5

After building the whole models, the next step is setting some things related. The loss function can be cross-entropy as it is in the classification category. Also, as I will mention in the next step, we can also use triplet loss function with both positive and negative data. The model should be trained from part to part instead of trained as a whole one.

4.6 Step 6

Since the models are supervised learning, I have to consider how to get training data. At first, my thought was to let the output to be a number in the interval $[0, 1]$.Human beings cannot name the exact number of similarity like 0.87, 0.53, etc.What should we do? The answer is to mark labels by ourselves. The only answer they would give will be same or not same. That is one or zero in terms of mathematics. So the label we will use are ones and zeros.

We can do some sudo data or data expansion. For positive pairs, based on some anchor files, we change some structures and functions into plain code. On the other hand, we also can only copy a part of the file to other irrelevant code. For some negative pairs, we must make sure neither structures not same but engaged in different programs as well.

What is more, if we hire knowledge from the contrastive learning, semi-supervised learning or GAN, it is not hard for us to generate some data by our own.