Data Structure & Algorithm Final Project

TeamID: team53 (藍泡麵) JudgeScore: 5719245.94

B07505045 王靖婷 B09902002 蕭朵婕 B09902088 傅樂芸

n: n_mails
m: length of token/name
c: number of tokens in a mail / expression
s: number of people involved in the mails given
phash: hash function generated by perfect hash tool
pmail: array that stores numbers i*138078+j, representing j-th token is in i-th mail
existence array: array that stores 0 or 1 based on whether the token exists or not for each mail
sim: direct map of similarity of every pair of mails

Data Structures & Algorithms

- Token / Name Hashing

  The recognition of tokens is required for both Expression-Match and Find-Similar queries. And for Group-Analyse query, it requires the recognition of names. In order to compare and recognize the string efficiently, our initial thought is using a **hash table** for every possible combination of successive alpha-numeric characters within the range of the length of the longest token / name.

- Mails Analysis

  Since we were given the dataset of every mail that might appear in the query, we decided to go through the mails and obtain some essential information, which includes:
  - length of the longest token (51) / name (11)
  - maximum number of tokens in a mail (14906)
  - a list of all unique tokens (total: 138078) / names (total: 560)

- Perfect Hash

  After realizing that there are limited tokens / names and we can generate a list of all unique possibilities, we decided to use python's package “perfect-hash 0.4.2” to generate perfect hash functions for both tokens and names. By giving it the unique token / name lists we have generated as inputs, it forms a one-to-one hash function that can look-up a specific string in time complexity of O(m), where m is the length of the string, since the calculation of the hash value can be done in O(1) for each character.

- Expression Match

  For Expression-Match query, the first method that came to mind is the postfix conversion taught in class. First, we process the given expression into an array of integers, which stores the hash value of the tokens, mixing it with some operators represented by negative values. Then we convert the expression into postfix using **stack**, and get rid of the brackets at the same time. After generating the postfix int function of the expression, we let the program iterate through each mail, evaluate the expression using **stack**, and use the existence arrays to calculate [token].

- Find Similar
  1. Rabin Karp

     Mapping digits 0-9 into values 1-10, the alphabets a(A)-z(Z) into values 11-37, we used the formula: **hash_value = (((s[1]*39+s[2])*39)+)...+s[n]** to represent a hash value of a token of length n. Given the type of hash_value is unsigned long long, we found this hash function is one-to-one (no spurious hit). The multiplier is 39 since we've tried 37, 38, and found 39 makes no collision.

2. Preprocess token existence map

   In order to minimize the time of checking if a specified token exists in a given mail, we decided to pre-calculate the existence of each token in all the mails into an array "pmail". We calculated the hash value of the token existing in the specified mail using "phash" and (the mail's id) *(138078, the number of all unique tokens), and stored their sum into "pmail".

3. Preprocess similarity

   We decided to pre-generate all the similarity between every two mails, the problem we encountered was the file being too large (900MB, where the limit is 120MB). Hence, we started to brainstorm ideas to compress data:

   (1.) **Magic hash function**

   We had the idea that maybe there was a kind of hash function generator that could find patterns with the list we provide and make it into a hash function, but it turned out we couldn't find any.

   (2.) **Reduce duplication, precision and commas**

   We noticed that half of the map is duplicated since the similarity of "a" and "b" is the same as "b" and "a". And we found a hash function that the input two keys are interchangeable:

   $$f(a,b)=\max(a,b)(\max(a,b)+1)/2+\min(a,b)$$

   Moreover, if we reduce the precision (to 4 decimal places) and give up the first digit (which is rarely 1), the file size could be further decreased. However, after halving the size and reducing some precisions, the file size was still at about 200MB, which was still too large.

   (3.) **ASCII code compression (final solution)**

   Then we noticed a character can represent 95 values (0-32 are unprintable) with a single space, so we decided to convert the numbers into characters. What we did was take two decimal places of the similarity, plus 32 and convert it into a character. Since we only have 95 characters to use, we set every number > 95 to 95, meaning that we gave up the queries with threshold > 0.95.

- Group Analyse

  For implementation, we had come up with two methods:

  1. Use **graph** and **DFS** to implement, with vertices representing people, edges representing the relation of sender and receiver. Every vertex has an array, which is used to store all the vertices it points to. After processing every mail in mids[], run DFS to count the answers required.
  2. Use the concept of **disjoint set** and skills taught in class like **union by size** and **path compression**. There are three arrays in our code: people[560], parent[560], size[560], recording whether the person has been handled in this query, its set's parent, and its set's size, respectively.

  And for cleaning people[] after the query, we've tried two means. One is to clean up every grid no matter if it's used or not, another is to record and clean the used ones only.

Cost Estimations of Queries

- Pre-Query

Rabin Karp	no pre-generated data	hash a token takes O(m), n mails each has c tokens	O(m*n*c)
mail preprocess	pmail[1628877]	malloc and generate 10000 existence array using pmail	O(1), malloc takes time
similarity pre-calculated	sim[10000][10000]	no operation required	O(1), almost no time

- Expression Match

naive	hash a token takes O(m) , each n mails has c tokens.	O(m*n*c)

- Find Similar

calculate similarity	O(1) to compare each token in each mail	O(n*min(c1, c2))
direct comparison	O(1) for each mail by looking up the table	O(n)

- Group Analyse

using graph	O(s) to construct the graph, and the number of vertices and edges are some multiples of s	O(s)
using disjoint set	as taught in class, it takes O(ɑs) each person, hence, O(sɑs) in total	O(sɑs)≅O(s)

Clean up after query	clean all	go through people[560] and set the value to 0	O(1)
	clean used	set those used grids to 0	O(s)

Scheduling Strategy

We've tried various combinations of different queries, such as Expression Match and Find Similar, Find Similar and Group Analyse. We had the conclusion that only doing the Find Similar query gets a much higher score than other combinations, because its efficiency of earning scores (average rewards per query * the percentage) is the highest, and unless we can finish all the queries of Find Similar and still have time left, focusing on this query is the best strategy.

Additional Note

For Group Analyse query, it turns out the combination of the disjoint set implementation and the first method of cleaning the array gets a better score on DSA Judge.

Before we are able to compress the similarity table, we have tried storing half of the similarity table and calculating the rest part along the way, or setting up a table and filling in the similarity table every time there is a comparison. Though none of them get a better score.