Codeforces 154C

Double Profiles: Problems and Solutions

Jansen Ken Pegrasio

National University of Singapore

May 20, 2025

Overview

Problem Statement

This problem is taken from Codeforces with this following link (click here for direct access).

An undirected graph with n vertices and m edges, where $n, m \leq 10^6$ is given. The task is to find how many unordered pairs of vertices (i, j) such that i may be a *friend* to j and all k other vertices are either a *friend* to both i and j, or not a *friend* to both of them. Vertices a and b is *friend* if there is a direct edge connecting both of them.

Graph Representation: Intuition

The first thing to consider when solving this problem is the graph representation. Since we are interested in identifying which vertices are connected to a specific vertex, we will not use an edge list representation. We will only consider these two approaches:

- Adjacency matrix
- Adjacency list

Graph Representation: Adjacency Matrix

Adjacency matrix helps to store the *friend* relationship between each vertex. However, to compute it, a complexity of $O(n^2)$, where n is the number of vertices, is required. This truly is too large. We will get TLE or MLE if we use this approach.

Graph Representation: Adjacency List

Another approach is to use the adjacency list. The strategy is to construct an array (sized n) of vector. Each vector will store the vertices which are *friend* to it. The complexity to compute this is $O(n + m)$ which fits the time and memory constraint.

The Equivalence of Being a *Friend*: Introduction

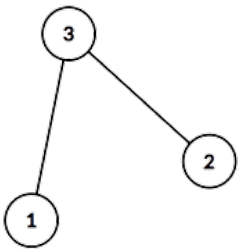
Given an undirected graph, we want to check whether i and j are *friend* or not. Then, consider these two scenarios:

- i and j are not *friend*
- i and j are *friend*

The Equivalence of Being a *Friend*: Scenario 1 (1)

If i and j are not friend, then the vectors corresponding to the index i and j in the adjacency list will be identical.

Consider the example below.



The Equivalence of Being a *Friend*: Scenario 1 (2)

The adjacency list for this graph will be in the forms of

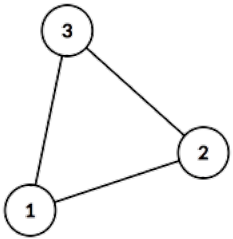
$$\mathbf{v}[1] = \{3\}$$
$$\mathbf{v}[2] = \{3\}$$
$$\mathbf{v}[3] = \{1, 2\}$$

Observe that $\mathbf{v}[1]$ and $\mathbf{v}[2]$, i.e., $\{3\}$ are identical.

The Equivalence of Being a *Friend*: Scenario 2 (1)

For this case, we need to perform a slight modification to the adjacency list. Instead of checking the identicalness of vectors in adjacency list, we will check the identicalness of the vectors appended by the current vertex.

Consider the example below.



The Equivalence of Being a *Friend*: Scenario 2 (2)

The adjacency list for this graph will be in the forms of

$$\mathbf{v}[1] = \{2, 3\}$$
$$\mathbf{v}[2] = \{1, 3\}$$
$$\mathbf{v}[3] = \{1, 2\}$$

Let say we want to check whether vertex 1 and 2 is *friend* or not. Then, instead of checking the equivalence of $\mathbf{v}[1]$ and $\mathbf{v}[2]$, we will check the identicalness of $\mathbf{v}[1] \cup \{1\} = \{1, 2, 3\}$ and $\mathbf{v}[2] \cup \{2\} = \{1, 2, 3\}$. They are indeed identical because as we can see from the graph that vertices 1 and 2 are *friend*.

Counting Lemma

Advancing the intuition, we will use the map to count the occurrence of all the sorted vector produced in both scenario 1 and 2. Observe that the vectors produced in scenario 1 and 2 will never be double counted. Therefore, for each vector $\mathbf{v}$ in the map *mp*, the number of pairs that are *friend* with common vector $\mathbf{v}$ is

$$\binom{mp[\mathbf{v}]}{2}$$

In total, the number of pairs that are *friend* is

$$\sum_{\mathbf{v} \in mp} \binom{mp[\mathbf{v}]}{2}$$

However, using a vector as a key on a map consumes a lot of memory, it is not a good idea. One way to optimize this is to perform hashing.

Hashing: Procedures

The first step in the hashing process is to assign every node with a 64-bit random number. Then, instead of saving a vector in the map, we will store the XOR of all the random numbers assigned to all nodes in the vector. By hashing the vector, we can save a lot of memory.

Hashing: Proof 1 (1)

In the first proof, we will justify that assigning every node a 64-bit random number is indeed safe, which means that the probability that two nodes have the same random number is negligible.

Let say that we have $M = 2^{64}$ possible hashing values, and we will choose k possible hashes from them. So, the probability that no chosen hash values are the same (no collision) is

$$
\begin{aligned}
P(\text{no collision}) &= \frac{\#\text{ configurations that result in no collision}}{\#\text{ configurations in total}} \\
&= \frac{{}_M P_k}{M^k} = \frac{M}{M} \cdot \frac{M-1}{M} \cdot \frac{M-2}{M} \cdots \frac{M-k+1}{M} \\
&= \left(1 - \frac{0}{M}\right) \cdot \left(1 - \frac{1}{M}\right) \cdot \left(1 - \frac{2}{M}\right) \cdots \left(1 - \frac{k-1}{M}\right) \\
&= \prod_{i=0}^{k-1} \left(1 - \frac{i}{M}\right)
\end{aligned}
$$

Hashing: Proof 1 (2)

According to the Taylor series expansion of the exponential function, we know

$$e^x \approx 1 + x$$

Therefore, using this approximation, we can approximate the probability of no collision which is

$$P(\text{no collision}) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{M}\right) \approx \prod_{i=0}^{k-1} e^{-\frac{i}{M}}$$
$$\approx e^{\sum_{i=0}^{k-1} -\frac{i}{M}} \approx e^{-\frac{(k-1)k}{2M}}$$

By this, we can calculate the probability of collision which is

$$P(\text{collision}) = 1 - e^{-\frac{(k-1)k}{2M}}$$

Hashing: Proof 1 (3)

The worst scenario happens when $k = 10^6$, the upper bound of N. When this is the case, the probability of collision can be calculated by

$$P(\text{collision}) \approx 1 - e^{-\frac{(10^6-1)(10^6)}{2 \cdot 2^{64}}} \approx 5.421 \cdot 10^{-8}$$

The probability of collision is extremely small and can be neglected.

Hashing: Proof 2

Next, we will prove that the probability of having vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ where $\mathbf{v}_1 \neq \mathbf{v}_2$ and the XOR of all elements in $\mathbf{v}_1$ and $\mathbf{v}_2$ are the same is negligible.

Let us define $X(\mathbf{v})$ where $\mathbf{v}$ is the vector that we hash as the XOR of all elements in $\mathbf{v}$.

We are given the assumption that $X(\mathbf{v}_1) = X(\mathbf{v}_2)$. By the property of XOR, this means that $X(\mathbf{v}_1)$ *XOR* $X(\mathbf{v}_2) = 0$, which implies that $X(\mathbf{v}_1 \cup \mathbf{v}_2) = 0$. Take an element from $\mathbf{v}_1 \cup \mathbf{v}_2$, call it $\mathbf{p}$. Let $mp[\mathbf{p}]$ be the random number assigned to $\mathbf{p}$. Then, we would have

$$X((\mathbf{v}_1 \cup \mathbf{v}_2) \backslash \mathbf{p}) \ XOR \ mp[\mathbf{p}] = 0$$

Observe that XOR equal to 0 is determined by $mp[\mathbf{p}]$ as $X((\mathbf{v}_1 \cup \mathbf{v}_2) \backslash \mathbf{p})$ is fixed to achieve XOR equal to 0. Then, as $mp[\mathbf{p}]$ is 64-bit, the probability that each bit value in $mp[\mathbf{p}]$ is the same as that in $X((\mathbf{v}_1 \cup \mathbf{v}_2) \backslash \mathbf{p})$ is $\frac{1}{2^{64}} \approx 5.42 \cdot 10^{-20}$, which is negligible.

Useful Reference

- To learn how to implement random number in C++, refer to this Codeforces blog (click this to navigate to it).
- Proof 1 is actually referring to a famous theory called the Birthday Paradox Approximation. A more detailed explanation can be found in this link.

The End