AAD algo analysis

1 Introduction

The whole idea behind this algorithm is to make sure that we can settle transactions between a group of a people without making it too fussy for anyone. We keep in mind two factors while thinking for solutions, they are

1. Does it respect the money cap issue while not making it hard for everyone to settle their transactions?
2. Is it scalable?

We will start looking for methods while keeping this in mind.

2 Primary approach

Consider a group of five friends named A,B,C,D and E. Each friend has some amount of money that he/she needs to get/pay to the other. Let us represent this group of friends as a directed graph with each friend as a node and the amount of money between two friends will be written on the arrow joining them.

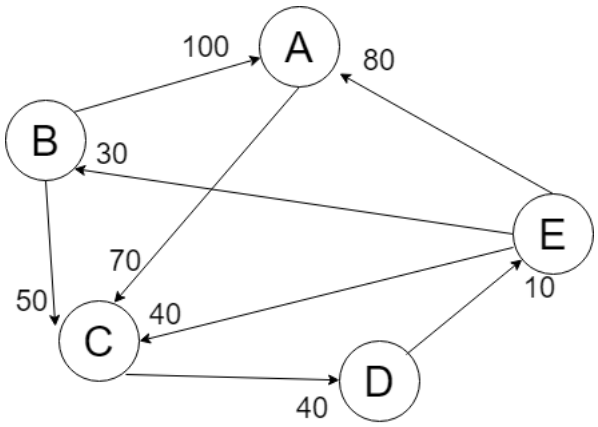

Figure 1: Representation of cash flow

Now that this is done, we calculate a value "total money" for each person in this graph (From now on, nodes refer to people).

$$\boxed{\text{Total money} = \text{cash outflow - cash inflow}}$$

For the above graph, the value for each person will be,

$$\begin{aligned} A &= -110 \\ B &= +120 \\ C &= -120 \\ D &= -30 \\ E &= +140 \end{aligned}$$

In simpler terms, having a positive value implies that the person needs to receive money overall and negative means that the person needs to pay money overall. We can quickly check that the sum of these values is indeed zero always.

3 Methods of approach

This problem can be tackled in 3 ways,

- Keeping the total cash flow on the minimum
- Keeping the total number of transactions on the minimum
- An approach which is intermediate to both the above ones

Now, let us look at each of these approaches individually.

4 Minimum cash flow approach

This approach can be broken down as follows

- We sort the array of values of each node in an ascending order and we straight away neglect any node with a value zero (since they need not pay/receive anything).
- We make a transaction between the first person in this list i.e. most negative value and the last person in this list i.e. most positive value in such a way that the person with least absolute value is made zero.
- One person's value is made zero and the other person's value is adjusted accordingly as sum of the two values.
- Now we recurse the above steps until everyone is removed from the list.

Applying this on the above example, we have

$$C = -120$$
$$A = -110$$
$$D = -30$$
$$B = +120$$
$$E = +140$$

So C and E make a transaction and C pays 120 to E. C's value is zero now and is no longer considered. E's value now becomes $140 - 120 = 20$
Recurse now,

$$A = -110$$
$$D = -30$$
$$E = +20$$
$$B = +120$$

So A and B make a transaction and A pays 110 to B. A's value is zero now and is no longer considered. B's value now becomes $120 - 110 = 10$

$$D = -30$$
$$B = +10$$
$$E = +20$$

So D and E make a transaction and D pays 20 to E. E's value is zero now and is no longer considered. D's value now becomes $20 - 30 = -10$

$$D = -10$$
$$B = +10$$

So D and B make a transaction and D pays 10 to B. D's value is zero now and is no longer considered. B's value now becomes $10 - 10 = 0$
Now, the list has no nodes and the algorithm terminates here.
Here, we can see that no matter what, the maximum number of transactions that are needed is always $n - 1$ for a group of n people. There can be cases where the initial list has people with value as zero and so are not included in the algorithm from the start. Such cases will need lesser than $n - 1$ transactions.
Drawbacks:
This algorithm in some cases makes one person make too many transactions.
Consider the following list of values

$$A = -6,000$$
$$B = 1,000$$
$$C = 2,000$$
$$D = 3,000$$

The algorithm suggests that A pays 3,000 to D, 2,000 to C and 1,000 to B. A makes 3 transactions and the rest don't make any.
Consider any list of the following form

$$A_1 = \sum_{i=2}^{n} x_i$$
$$A_2 = x_2$$
$$A_3 = x_3$$
$$\vdots$$
$$A_n = x_n$$

The algorithm always asks one person to make $n - 1$ transactions and zero transactions for the rest.

5 Minimum transaction approach

For this algorithm, when the list the made and is arranged, we need to start looking for **zero total groups** in the list of people.
A zero total group is a group of people whose values, when added give a zero. We start looking for such groups with 2 people, progress to 3 and so on until $n-2$ people. Whenever such a group is found, the minimum cash flow algorithm is applied only to that group and the people belonging to that group are removed from the list. Observe that for every group found, we are effectively reducing one transaction required for the list of n people.
Refer to the list obtained from figure 1,
If we start looking for groups of 2, the group of C and B can be removed from the list with one transaction. The rest 3 people need 2 transactions to settle. The total number of transactions needed for settling the expenses here is only 3, compared to what the older algorithm did, which was 4. This algorithm is now truly using the minimum number of transactions.
A point to notice, this algorithm now becomes $O(n!)$ in time complexity. This is a gamble to make since more the number of people, the higher the chances in general to find zero total groups. But a higher n also means more time needed for the code to run.
We've noticed that on using this addition, we might reduce number of transactions, but for a difference for 2 people for n being even as small as 10, the difference is huge. So, we won't be using this addition.

6 A few other work-arounds for the solution

This was an algorithm we thought of. In this, the number of transactions aren't fully minimized. Instead, we make groups of two from the list and remove any zero total groups as usual. After that, the approach changes. We find the pair(s) with the lowest absolute total while making sure that the two values that compromised this total are opposite in sign to each other and settle the transactions for this pair(s) using the minimum cash flow algorithm. Then, we adjust the value(s) of the other person/people, sort the whole list again and recurse this algorithm until everyone is removed from the list.
This was another algorithm we came up with,
Consider the minimum cash flow algorithm again. We need to make a few changes to it. Every time we need to make a transaction, we check for an additional condition. We check whether the value that is about to zeroed and it's immediate value together can result in a higher absolute value than the value on the other extreme end. If this condition is achieved, we just continue with the original algorithm. If the above condition isn't satisfied, we zero out the the value with the higher absolute value in such a manner that it is split across the two extreme values of the other end in the ratio of their values.
Consider the following list,

$$A = -70$$
$$B = -30$$
$$C = -20$$
$$D = -10$$
$$E = 10$$
$$F = 120$$

The first transaction would be A paying 84 to F and B paying 36 to F because the condition isn't satisfied since, $abs(-70-30) < abs(120)$. The reason why A and B pay that specific amount is, $\frac{7}{7+3} \times 120 = 84$ and $\frac{3}{7+3} \times 120 = 36$. The next transactions would be C paying 14 to A, D paying 10 to E and C paying 6 to B.
The cash flow has increased by a little, but the number of transactions are now even for all the members. Consider the following list,

$$A = -890$$
$$B = -20$$
$$C = 60$$
$$D = 850$$

The pairings and their totals would be AB = -910,AC = -830,AD = -40,BC = 40,BD = 830,CD = 910.
The lowest absolute value in these totals is 40 and pairs AD and BC have it. AD and BC both are made of values which are opposite in signs to each other. SO, we settle the transactions between AD and BC as A pays 850 to D and B pays 20 to C. The resulting list would be

$$A = -40$$
$$C = 40$$

This ends with A paying 40 to C.
We decided to mention these algorithms merely for the fact that these were some algorithms we thought could make the cut, but were discarded by us.

7 The third approach

The third approach is somewhat a "mixture" of the first two approaches. We compromise a little on the number of transactions being minimum and we also compromise a little on the cash flow to ensure that the results of the algorithm is practically feasible.

The approach

We use the minimum cash flow algorithm with one extra condition. This condition is in place to ensure that the money cap is respected for every person. Say, the limit is x. Now, we add a new condition for one if condition, which is the case when the most negative value has a more absolute value than the positive one. We check if the absolute value of their sum is lesser than the limit we set. If it is, the usual minimum cash flow algorithm occurs. Else, we reverse the transaction that is bound to happen. We do this to avoid the fact that the person who needs to receive money ends up getting too much money and has to pay back to someone else too much money. This case becomes particularly problematic when the person who needs to get money doesn't receive it immediately.

Consider the following example,

$$A = -60$$
$$B = -40$$
$$C = -10$$
$$D = 20$$
$$E = 90$$

The transactions made by the algorithm when the limit is set to 500 are as follows,

E pays 90 to A, A pays 30 to B, D pays 20 to B and B pays 10 to C.

When the limit is set to 20, the transactions change to respect the limit. The new transactions are as follows,

E pays 60 to A, E pays 30 to B, D pays 20 to B and B pays 10 to C.

This algorithm, is the best possible approach for us and hence we've decided to use this algorithm in the app.