Who Owes Who? Payment Optimization for Digital Cash Transfer

Ali Monfre, Oyu Choijamts, Samuel Kaplan, William Whitman

Overview:

Our goal was to implement a system which, given a number of payments between a group of people, would find an optimization to resolve these payments. As an additional degree of difficulty/functionality, our system aimed to also take into account friendship and trust in informing what payment solution was optimal. Stated more clearly, we allow users to input people they "trust" who are considered part of their network. Then for that particular network of trust, everyone in that group is able to pay off the others' debts in order to minimize transactions. For example, if Charlie and Ben are friends, and Charlie owes Ben money and Ben owes John money, the algorithm will tell Charlie to pay John directly.

To do this, we implemented two separate algorithms. We had two separate implementations because for different situations, as well as personal preferences, a different solution would be optimal and we wanted to account for this. The first algorithm simply aims to minimize the total number of transactions. The second algorithm aimed to also decrease the amount of payments, but more importantly looked to allow only for transactions between friends. When this was not possible the algorithm then looked to minimize the total amount of "un-trusted", or non friend (group) to friend (group) transactions. In a scenario in which we want to minimize the number of payments to untrusted individuals, this would be optimal.

These algorithms were packaged into a sleek user interface that allowed for easy use of the system, as well as easy visualization of both the original payment solution, and the optimal solution given by both of our algorithms.

Instructions:

We included the instructions in the README.txt file in the code folder.

Planning:

We accomplished what we planned to do in our specifications. Please see the initial draft specification and final specification for details. We stuck pretty close to the timeline we proposed. We concurrently worked on the algorithms and the GUI, and successfully integrated them together.

Design and Implementation:

Our first algorithm, greedy, tries to minimize the total number of transactions. For each friend group, all individuals' net worths are calculated. We then order the net worths, so we can keep track of who owes the most amount of money and who owes the least amount of money. If any perfect matches exist (i.e. someone owes the exact amount that someone else should receive), we match them because that results in one transaction - the minimum possible. Otherwise, we match the person who owes the most with the person who needs the most. This results in one transaction for one of the individuals, which is still an optimization. In this way, we are always guaranteeing one transaction for at least one of the two individuals, bringing the number of transactions down to, in the worst case, n-1.

Our second algorithm, friendly, was designed to first minimize the total number of transactions, and to then make sure that all these transactions occurred within a friend group. We wanted to only allow one transaction between friend groups. To do this, we first computed the net worth of each individual in our system. Then, given a net worth list, we were able to minimize the total number of payments within the group. To do this, we first searched for debts and credits that were exactly equal. If the debt and credit were equal, we would first want to resolve these, as it would allow for one less payment. We then went through the list and calculated the max amounted indebted and credited. We resolved these payments, adding the payment to our payment matrix, and continued to do so until everyone's net worth was 0. Once we calculated this optimal payment matrix, we went in and enforced the condition that if possible, all payments must occur between friends. To do so we identified all payments not between friends. We then performed a breadth first search on the creditors friends to see if the two unconnected parties were connected by a mutual friend. If so, the payment was re-routed through this individual. If not, the search was continued, expanding the breadth of the search up to 4 friends in common. If a common friend "channel" had not been identified by this point, it was deemed to be impractical to continue, as the marginal return on ensuring a trusted payment channel to the increased number of payments from the creation of this channel was diminished.

Once we had ensured all possible payments occurred within a trusted network, we analyzed all the remaining "un-trusted" transactions. We knew that these transactions would have to be made, but we wanted to minimize the total number of these transactions. To do so, we found the first un-trusted payment in the matrix. We then looked at all friends of the creditor of this un-trusted payment. If one of these friends also had an un-trusted payment to

a person within the same friend group as the debtor on the previous payment, we knew we could combine these payments to decrease the number of un-trusted payments. While this did come with an increase in the total number of payments, we deemed the decrease in un-trusted payments to be outweigh this. To combine these payments we resolved the additional un-trusted payment within each friend group, and then simply had the amount of total money owed between the two transaction be transferred over in only one un-trusted payment. For an easier explanation and visualization of this, you can see the code in lines 219-276 of friendly.py.

We can see the implementation of these algorithms in the “algorithms” folder of our code. Greedy_init.py and friendly_init.py take into account raw files (wow_init.in) and only disjoint friend groups. Greedy_final.py and friendly_final.py take input from the user interface (like, for example, wow.in) and take into account overlapping friend groups.

The GUI for our project is straightforward, elegant and intuitive. It is built on Django for easy integration with the python algorithm files, and can be found in the “interface” folder within the code. The platform allows for registering, logging in, adding transactions and visualizing them in a graphical representation among others. We also account for whether the transaction is borrowing or lending. The user can see the transactions being moved around to give the updated, optimized solution.

Results:

Looking at our algorithm output, we can first see that our algorithms perform very well with the given data. The number of payments are significantly decreased, and there are no double payments. We also see that each algorithm performs in a very distinct way. The greedy algorithm computes a web of payments that are, for the most part, all connected. This is to be expected given the way it optimizes the data, because unless a perfect match exists, one of the two people involved in a transaction need to undergo at least one more transaction to pay off their debt or receive what they are owed. The second algorithm, in contrast, computes a more disjointed web of payments. Looking at how we calculate this solution, this is also to be expected. Because we first optimize the payments taking into account no friend groups, and then reroute this optimization to always travel through friends, we would expect to see multiple disjointed payment “clusters”, each of which flows within a particular friend group.

Reflection:

Although we were planning to implement one algorithm that would work in all situations, we found that in fact, we have two algorithms that are optimal for different scenarios. It was a pleasant surprise that helped us understand the algorithms better and look more in-depth into the problem at hand. Because of the complexity of this challenge, if given more time we would like to explore more optimization algorithms, specifically adding elements in the algorithm to allow for it to take distance, payment type, and even preferred currency into account. We would also like to work more on the web of trust and incorporate the Harvard Pin network. In fact, we began to work on cryptography and security, and implemented a simple RSA cryptosystem (found in the algorithms folder as rsa.py) that we could have integrated into the code to make the transactions more secure.

Advice for Future Students:

The most important thing we learned was that it is important to keep thinking about the problem and trying to find more optimizations and solutions, even if you already have a working one. It has helped us broaden our horizons and push ourselves to dig deeper. Also, at times this problem seemed overwhelming, and a solution seemed to be near impossible. However, when this happens, instead of freaking out and totally changing direction, simply break the problem down into much smaller pieces and see if you can just solve those. Once you do this the final solution may seem much more obvious and simple.