Analysis

Project Outline

A project to help groups of people manage money using digitally signed transactions. Since the project is not intended to handle actual money, it also provides a way to quickly and easily settle transactions using a minimal number of steps [1]. To interact with the final product, a simple, easy to use command line interface will be provided

Features

- Cryptographically signed transactions guaranteeing security and integrity of your transactions
- Settle your group's debts in the fewest number of transactions
- Command Line Interface (CLI)
- Client / Server Model
- Database

Non-Features

- None of the user's money is ever put into the app. This is simply a tracker, you cannot settle debts through the app
- No policing of people who do not pay their debt - this is a problem for people in the group to deal with as they choose

Background to the Problem

A common problem for many young people is that of money. More specifically, keeping track of who owes who how much money in a group of friends. Arguments about how much money is owed, and whether or not people have been remunerated are commonplace. This is something I often see amongst my own group of friends. I know one person in particular (the end user) feels as though he is never payed back, and would like to see a solution to the money tracking problem.

In order to arrive at a solution, what is needed is a reliable, trustworthy way to track money. People who use the tracker will need some sort of guarantee that people cannot 'hack' the app, changing people's debts. Since I am the creator, and likely a future user of this app, my friends also need confidence that I will not be able to write off all of my debts. Hence, that is problem number 1 - **secured transactions**.

A problem inherent to a money tracker is that it is just that - a tracker. Since no money flows through the software, long chains of debt may form. This may result in people needing to make many small transactions to different people in order to pay what they owe. This seems a shame, when it is possible to have the software simplify the debts into a minimal number of transactions per person, to ensure money flows as efficiently as possible through the network. This idea was presented to me by the end user, but I anticipate it to be an integral part of the project, hence the brief mention here. This makes problem number 2 - **efficient settling of group debt**.

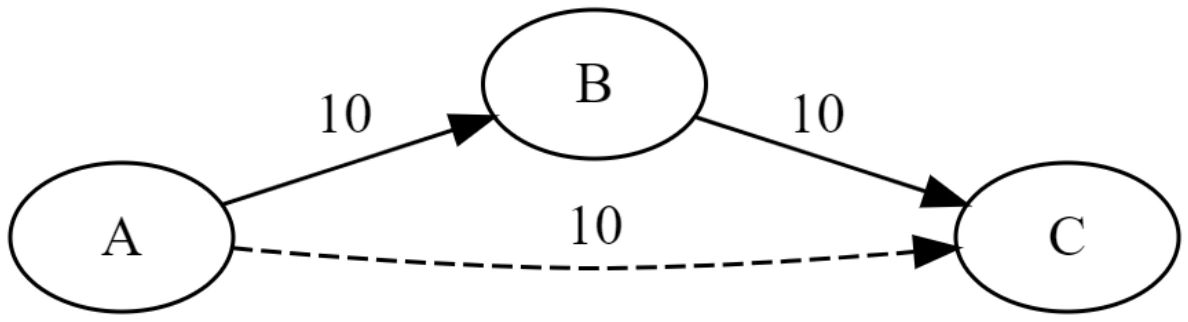
The End User

The end user that I had in mind is the friend who inspired this project. I think he fits perfectly within the target market - 13-18 years old, fairly sociable, not too technologically minded, and just wants an easy way to keep track of his transactions in his group of friends.

My interviewee was keen to point out that people may abuse this system. They may just use it to rack up debt, and then never pay anyone back. He wanted to know if there was a way that I could stop this occurring. To this I replied no, not really. They can do the same thing without the app. It is your decision whether or not to lend to them. With the app however you can see exactly how much they've taken. It does however assume that people are willing to pay up. It is up to the user to deal with the eventuality that they don't.

Another problem that he identified was that of chains of debt. He (rightly) pointed out that if you owe people who owe people, its sometimes easier to cut out the middle man and turn two discreet transactions into one smaller one. This idea naturally extends to a group of friends, who may all have varying levels of debt between them. Thus, instead of making the group do a large number of transactions with money going back and forth frequently between the same hands, I will aim to let a group settle in the easiest way possible.

A valid concern with this plan is the fact that some may end up owing people they didn't before the simplification. Consider a simple case where A owes B £10, and B owes C £10. Two transactions could be reduced to 1, if A were to pay C directly. However, in a larger group, people may not like giving money to people who they do not directly owe on the will of my program. Hence, an important constraint is that no one owes someone that they didn't owe before settling occurred (see image below.)



Even though the dashed edge would reduce transactions, it should not be added.

The end user suggested that I keep the interface as simple as possible. He maintained that he didn't want lots of unnecessary frills - just a simple, functional interface. To this, I suggested the use of a CLI. I was a little concerned that most people would not have used one before and may not know what it is. However, once I explained the concept, he seemed to come round to the idea. It's main benefit over a graphical user interface (GUI) is that it is unambiguous. It is also, arguably, easier to do things with a CLI once you know how to use it.

The final main worry that my interviewee brought up was that of guaranteed security. I had talked to him when I had the idea for this project, before I had learned about asymmetric encryption and cryptographic signatures. He wanted to know how a transaction coming from him could be verified as his, and no one else could pretend that they are, say, owed lots of money. He also didn't trust me, and said that if our group of friends started using this product, he would suspect that I would "code away my debts"

In short, the problems identified here are as follows:

- How to make sure that users trust the integrity of the transactions, making sure that users know that no one can tamper with their debts.
- How to settle debt across large graphs efficiently (here meaning few transactions per person)
- How to make a CLI that is as simple as possible

This is not an exhaustive list - it leaves out all of the technical problems I will likely face, which are discussed in the next two sections

---

Research of existing solutions

Currently on the market, there are a few products similar to that which I am proposing. Having surveyed a few options, I decided to look at Evenfy and Splitwise in more detail.

Both work on the same premise that I have outlined: an intuitive way to track who owes who in a group.

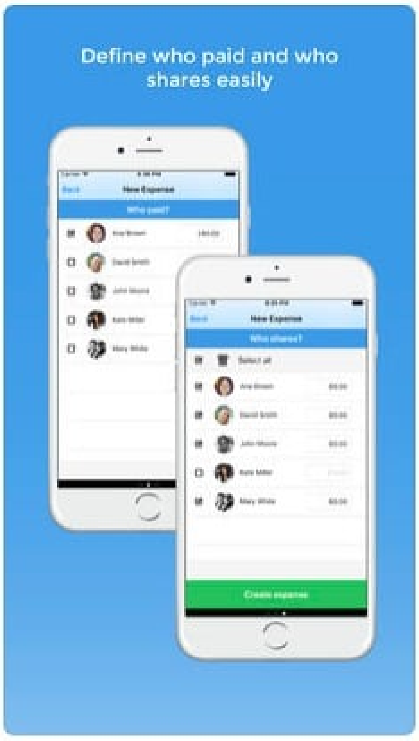
This is all accurate at time of writing, however new updates since may

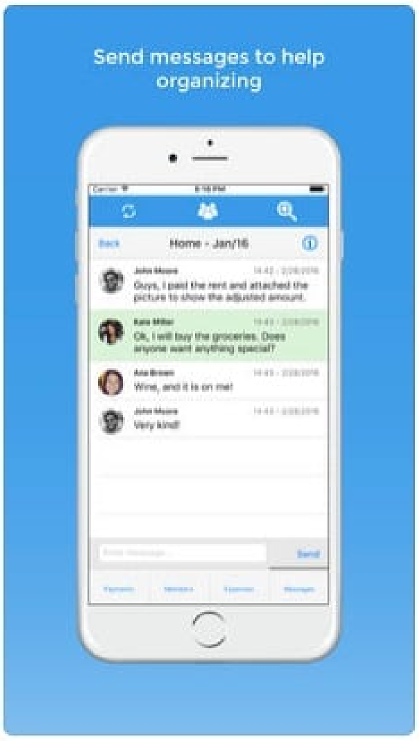
Evenfy

Evenfy is an app that does exactly what I set out to achieve. It mainly focuses on group expenses, and can be accessed from a computer.

It has an interesting feature in that it allows for temporary groups to be created in order to track short term expenses, such as over the course of an event. Evenfy tries to learn about common expenses, and suggests who pays over time. This may be useful if you rent a house with a few others, is what Evenfy say.

Evenfy will also calculate the easiest way to settle the group; that is, ensure that everyone's debt goes to 0. This is an integral part of keeping an expense tracking app usable in my opinion, and in the app's reviews, users seem to agree.



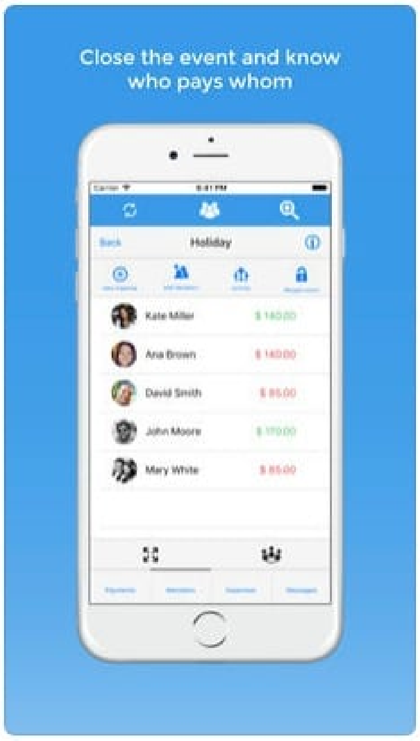


John Moore: Guys, I paid the rent and attached the picture to show the adjusted amount.

Kate Miller: Ok, I will buy the groceries. Does anyone want anything special?

Ana Brown: Wine, and it is on me!

John Moore: Very kind!



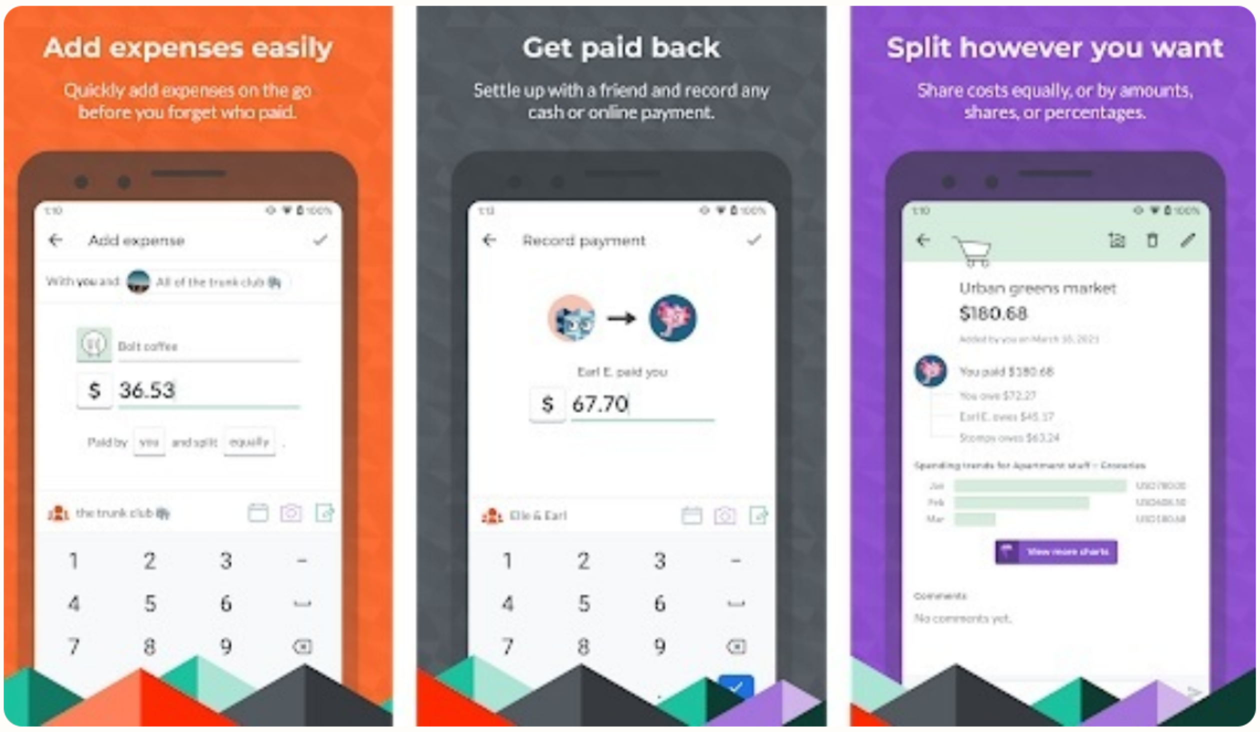
Kate Miller	$ 140.00
Ana Brown	$ 140.00
David Smith	$ 85.00
John Moore	$ 170.00
Mary White	$ 85.00

Evenfy also allows for a group to settle debts easily. This feature will be discussed more in the next product evaluation, as an identical feature appears there.

Evenfy is free to use for the first 6 months, but then requires monthly subscription of 99 cents if you want to track more than 10 expenses per month.

Splitwise

Splitwise is similar to Evenfy in many ways. It allows the easy splitting of bills (by percentage or equally), and keeps track of who owes whom within the group.


Add expenses easily
Quickly add expenses on the go before you forget who paid.
Get paid back
Settle up with a friend and record any cash or online payment.
Split however you want
Share costs equally, or by amounts, shares, or percentages.

After an account is created, you can create a group of people. Splitwise will then track all expenses in this group. Expenses can be referenced, and you can see at a glance exactly how much you owe.

In comparison with Evenfy, the overall experience and feature set is similar. Both are well put together and include features that I think are unnecessary for my target market. I do prefer Splitwise's UI slightly: it is easier on the eyes, and slightly less cluttered. Both have an excellent UX.

My only complaint is that I feel as though I have to search through a user interface for many quite simple tasks. The settings menu in particular feels like it obfuscates settings such as deleting your account, as well as updating user information. This is, however, common to many graphical user interfaces. As I do not plan on building a graphical interface, I do not think this is something that I need to be too mindful of. Instead, I will strive to make the CLI as straightforward as possible.

In terms of flaws, Splitwise requires a £2.99 per month subscription to unlock every feature (most core features are free to use). This is, in my opinion, better than Evenfy's payment model. However, the point is moot as I will not be charging for the use of this project.

The interesting part of these apps is in the debt simplification process. Since neither are open source, one cannot know for sure how the debt simplification is done. However, after much investigation, I found a few possible options.

Add another research

Givers and Receivers

This, I believe, is the less likely of the two possible approaches, because it reduces down to a decision theory problem which is NP-Complete. It also takes a few passes of the data to get there. It is not particularly efficient. The steps are as follows

1. Calculate the net flow of each node
2. Categorize nodes into 'givers' (those who overall owe money) and 'receivers' (those who are overall owed), and those who owe/are owed nothing.
3. Settle any '1-1 transactions', where one 'giver' can completely remunerate a 'receiver'.
4. Settle any transactions where a receiver is owed a perfect subset of givers money (i.e. a receiver who is owed £5 could be settled by two givers, with £3 and £2)
5. Settle any remaining transactions by splitting money from givers in a greedy fashion to receivers.

The 'NP-completeness' of the problem starts at stage 4, when lots of computation has already been done on the data.

Stage 1 reducing all the transaction from Person A → Person B to a single number. This is then done for the whole group, and a weighted digraph is built, where edges represent money owed. A residual graph, wherein an edge in the opposite direction with a weight $\times = -1$ the initial weight is added.

A breadth-first-search, where each edge traversed from the current node (*u*) has its weight recorded (and summed when all edges from *u* are explored) is required to obtain the flow of the graph. This gives a time complexity of $\mathcal{O}(|V| + |E|)$.

The second step can be done very efficiently, in $\mathcal{O}(logV)$ time if a mergesort is used to sort the nodes in descending order of money they have (assuming those owed have negative amounts of money). Then the list of nodes can be split into two subsets at the point where 0 would be inserted into the list (those with a net debt of 0 can be removed before splitting).

Step 3 could be done by walking through the sorted givers list. For each node *g* in the givers list, you would walk down the receivers list (*r* in receivers), settling any transactions where *g*[money] = r[money]. (Any nodes which are not 'filtered' in this way move to a new stage of processing). This would give a time complexity of $\mathcal{O}(G \cdot R)$, where G is the number of 'givers,' and R is the number of 'receivers'. This can be made much more efficient by using sorted lists, and remembering how far the receivers list was walked down in previous runs.

In pseudocode

```
for giver in givers:
        for receiver in receivers:
                if giver.money > receiver.money:
                        // giver has more money than any receiver in receivers
                        giver passes filter
                else if giver.money < receiver.money:
                    // receiver is owed more money than any giver in givers has
                        receiver passes filter
                else:
                        // giver and receiver have the same amount of money so a 1-1 can happen
                        settle(giver, receiver)
```

Step 4 becomes a sum of subsets problem, and is NP-Complete, i.e. there is no algorithm which can solve this problem in quicker than exponential time. It is at this stage that this approach becomes infeasible for the real world, and thus, A better heuristic must be considered.

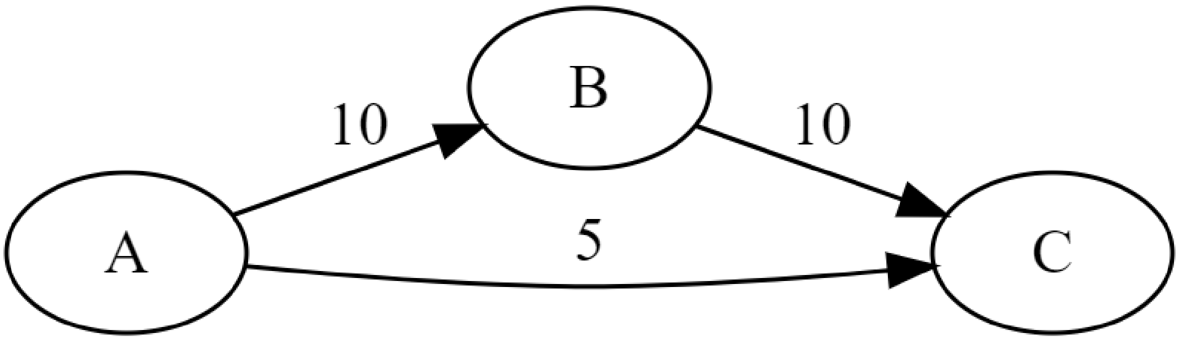
Another problem with this approach is that it contradicts one of my initial high level requirements. It does not preserve any sense of who owes whom past step 1. Thus, it cannot guarantee that no one will owe anyone that they did not previously owe.

For these two reasons, I shall elect to not use this method.

Max Flow

I think that this problem can be modelled as a problem of flow. The question 'how can we minimise the number of transactions one person has to pay to the group' can be reduced to the question 'how do I maximise the amount of money I sent to one person'. If the amount of money sent to someone is maximised such that no one ever has to pay more than they owe, then people will, on average, have fewer transactions to make after settling.

Consider the simple case of three people, Alice, Bob and Charlie. Let Alice owe Bob £10, Bob owe Charlie £10, and Alice owe Charlie £5. This can be represented as a weighted digraph



Notice that there is a chain from A -[10]→ B -[10]→ C.

We can think of this chain of money transferring as £10 of Alice eventually ends up with Charlie, even though it goes via Bob. Thus, this can be simplified by cutting out the middle man, and instead letting Alice owe Charlie £15.

I believe that this functionality can be achieved through a slightly unusual application of the Edmonds-Karp Max Flow Algorithm.

Fulkerson-Ford Max Flow

The Edmonds-Karp Max flow algorithm is really a combination of two separate algorithms: a breadth first search, and the Ford-Fulkerson max flow algorithm.

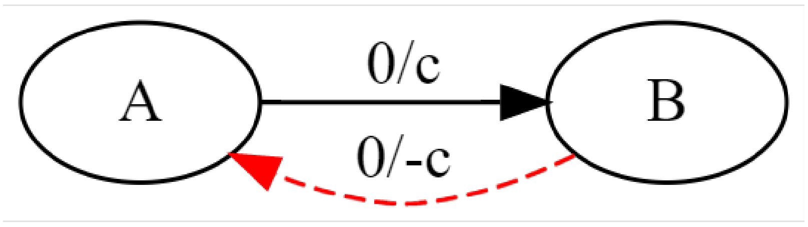
Ford-Fulkerson aims to answer the question: how much flow can one push along a network, without exceeding the capacity of any edge? The algorithm works on flow graphs.

The way in which the algorithm works is simple:
1) Find an augmenting path from source node to sink node (through the residual graph)
2) Augment flow down path
3) Repeat until no more augmenting paths exist

To define some terms:
Flow Graph
A flow graph is a form of weighted digraph, in which edges have a flow and a capacity (as opposed to a single weight). Each edge has the notion of remaining capacity (remaining capacity := capacity - flow). Each edge is initialised with flow = 0. This changes during the course of the algorithm.
The flow of an edge is never allowed to exceed its capacity. (i.e. edge will never have a negative remaining capacity)

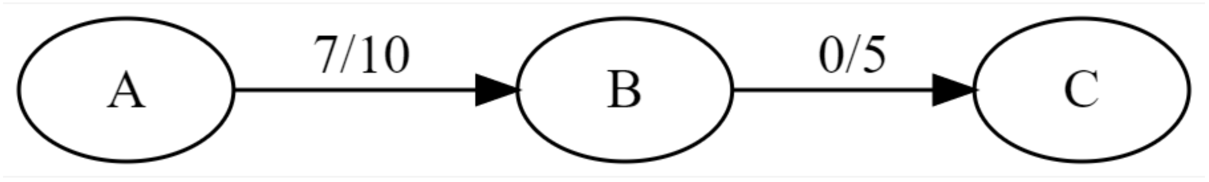
Augmenting Path
The augmenting path is a path of edges in the residual graph, where each edge has a remaining capacity > 0. The path is from two specified nodes - a source node s and a sink node t

Residual Graph and **Residual Edges**
The residual graph is the combination of the flow graph , and residual edges. For each original edge from $u \rightarrow v$, with capacity c, there exists a residual edge from $v \rightarrow u$, with capacity $-c$

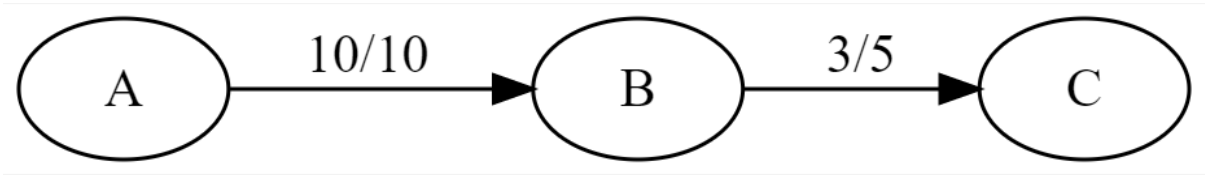


Residual edges are valid edges to consider when looking for an augmenting path, given that they have unused capacity (the above example's residual edge has an unused capacity c, as $0 - -c = c$)

Augmenting the flow
The act of pushing as much flow as possible along an augmenting path.
The amount of flow pushed down the path is equal to the bottleneck value of the path.
The bottleneck value is given by the edge with smallest amount of unused capacity



In the above case, the bottleneck value is 3. This is because A →B has 3 units of unused capacity, and B → C has 5 units of remaining capacity. Thus, the maximum amount of flow that can be pushed through this augmenting path is 3 units. After augmenting the flow, the path looks like this



An example of the algorithm working is provided in the next subsection

Once no more augmenting paths can be found, the bottleneck values of each of the augmenting paths used in the BFS are summed. This value is the max flow through the given graph from the source node to the sink node.

Complexity

Finding an augmenting path is completed in $\mathcal{O}(E)$ time (where E is the number of edges in the graph). In the worst case, 1 unit of flow is added every iteration. This makes the overall time complexity of the Fulkerson-Ford Max Flow $\mathcal{O}(E \cdot f)$, where f is the max flow of the graph.

This is not ideal, as the time complexity is heavily dependant on the flow through the graph. This is improved upon in the strongly polynomial Edmonds-Karp Max Flow algorithm

Edmonds-Karp Max Flow

Edmonds-Karp Max Flow differs from Fulkerson-Ford in the finding of augmenting paths. Fulkerson-Ford does not specify how an augmenting path should be found, whereas Edmonds-Karp finds the shortest [2] augmenting path from s to t.

This is ensured by using a Breadth First Search (BFS) to find augmenting paths.

A short augmenting path is favourable, as the longer the augmenting path is, the higher the chance of an edge with with very little unused capacity. This could lead to edges reaching capacity in more iterations, giving a considerably slower runtime. As aforementioned, the worst case is that every path has a bottleneck of 1 unit of flow. Since Edmonds-Karp uses a BFS to find augmenting paths, we are guaranteed the shortest (in terms of number of edges traversed) path from s to t.

This detail gives Edmonds-Karp a much more favourable time complexity, of $\mathcal{O}(EV^2)$. This is a product of the time complexity of a BFS, $\mathcal{O}(V^2)$ (when using an adjacency matrix to represent the graph) and the Fulkerson Ford time complexity, $\mathcal{O}(E \cdot f)$. However, flow does not appear in the time complexity of Edmonds-Karp.

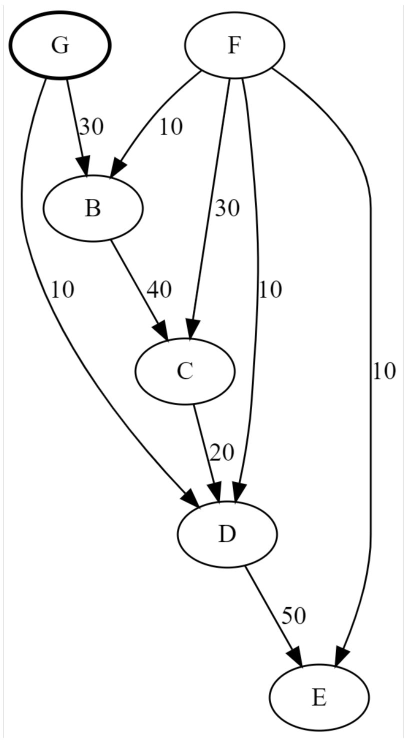
A property of BFS is that, when it finds a path from a source node s to a target node t, that path is guaranteed to be the shortest path (P) from s to t. A corollary of this is that P_{n+1} is guaranteed to be a longer path than P_n. This reduces the upper bound run time of one iteration to $\mathcal{O}(E)$ versus the original $\mathcal{O}(E \cdot f)$.

Since Edmonds-Karp's runtime is independent of flow, its input, it is classed as a strongly polynomial algorithm, making it perform much better than the Fulkerson-Ford max flow algorithm. Thus, this is the algorithm that I will be implementing to simplify debts across a group.

Settling a graph using a Max Flow algorithm

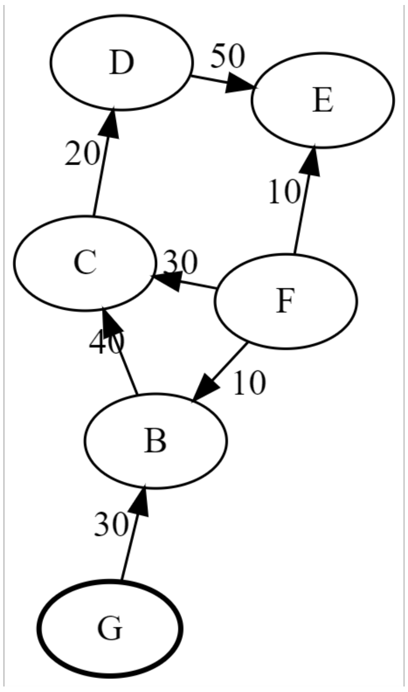
Having explored various max flow algorithms, the question now becomes how to settle an entire graph's worth of debt. This is a fairly challenging problem since max flow algorithms only work on a source node an sink node.

The solution is to walk through the graph, and run a max flow from the current node to each of their neighbours.



First, a new weighted digraph is generated (with no edges, but all the same nodes) for the cleaned edges.

Starting on node G, we would therefore run a max flow from $G \to B$. If the max flow from $G \to B > 0$, then an edge with the weight of the max flow from $G \to B$ is added to the new weighted digraph. The edge from $G \to B$ in the flow graph is deleted. This process will happen again from $G \to D$. After having explored all neighbours, the BFS continues, until every edge in the graph has been settled. In the above example, a valid settling could look like this [3]



Note: **Implications to security**

Since the server that is settling a group of transactions is creating and destroying new transactions that haven't happened in the real world, the signatures that transactions were initialised with will no longer be valid after settling.

The solution to extend security through the settling process is to have the server be able to sign the new transactions. There may be a security risk here in that the server's private key can act as a master key, validating any transaction.

However, as with all private key cryptography, private keys need to be kept secret. Thus, while this is something that needs to be kept in mind, it does not require large redesigns of the security software.

High Level Objectives for the Solution

After careful consideration of the end user, and existing systems, I can arrive at my high level and low level requirements

On a high level, my objectives are as follows:

1. [x] An RSA implementation that will allow the signing, and verification, of transactions
2. [x] A way to settle the debts of the group in as few as possible (heuristically speaking) monetary transfers
3. [ ] A server-side component of the application which can verify transactions, and store / retrieve them from a database
4. [ ] A client-side component of the application that will have a simple user interface (CLI)
5. [ ] A database that should be able to store user and transaction information

Low Level Requirements

For the purposes of testing, these are low level requirements that I would like to fulfil.

RSA Implementation (A)

1. A reliable interface to a hashing module

2. RSA Key Handling:

    1. Be able to load RSA public/private keys in PEM format from files / STDIN
    2. Be able to validate the format of these keys
    3. Be able to parse these keys extracting all necessary numbers for RSA decryption

3. Signing/Verification

    1. Have a valid RSA encryption scheme (encryption with public key)

2. Have a valid RSA decryption scheme (decryption with private key)
3. Have a valid RSA signing (sig) scheme (signing with private key)
4. Have a valid RSA signature verification (verif) scheme (verify with public key)

4. Object Signing

    1. Algorithm to convert an object to a hash in a reproducible way, minimising the chance of hash collisions
    2. Ability to sign a class of object with RSA sig scheme
    3. Ability to verify a signed object with RSA verif scheme, raising an error if signature is invalid
    4. The server is able to use a master key to sign settled transactions

Debt Simplification (B)

1. A reliable digraph structure, with operations to `transactions.graph.GenericDigraph`

    1. Get the nodes in the graph `nodes()`
    2. Check if a node is in a graph (`is_node(v: Vertex)`)
    3. Check if an edge exists between two nodes
    4. Nodes can be added
    5. Nodes can be removed
    6. Edges can be added
    7. Edges can be removed
    8. Neighbours of a node should be easily accessed (neighbours for the purposes of a breadth first search)

2. A reliable weighted graph structure `transactions.graph.WeightedDigraph`

    1. All of the operations listed in B.1.1
    2. Adding an edge should have different functionality: edge should be able to be added with a weight

3. A reliable flow graph structure

    1. All of the operations listed in B.1.1
    2. Adding an edge should have different functionality: edge should be able to be added with a capacity, and edges should have a notion of flow and unused capacity
    3. Be able to return neighbours of nodes in the residual graph (i.e. edges, including residual edges, that have unused capacity)
    4. A way to get the bottleneck value of a path, given a path of nodes

4. A reliable recursive BFS that works on

        1. Digraphs
        2. Weighted Digraphs
        3. Flow Graphs
        4. BFS should also be able to be used to operate on edges in a graph

talk about combination, passing function into recursive bfs
5) Implementation of Edmonds-Karp
2) Way to find shortest augmenting path between two nodes
3) Way to find bottleneck value of a path
4) Finding max flow along a flow graph from source node to sink node

6. Simplifying an entire graph using Edmonds Karp, using the method laid out in [Settling a graph using a Max Flow algorithm](#).

7. Be able to convert a list of valid transactions into a flow graph

8. Be able to convert a digraph into a list of transactions, signed by the server

Client / Server Structure (C)

1. The server should be accessible to the client via a REST API
2. The server should be able to pull a group's transactions from a database, run the settling
3. The client should be able to request
    1. See their own user information
        1) Total debt across all groups
        2) Open transactions
        3) Closed transactions
        DECIDE / SEE TIME
    2. Open transactions / closed transactions
    3. Mark a transaction as settled
    4. Make / invite people / leave groups
    5. Settle a group
    6. Create a transaction
    7. Sign an open transaction
    8. Mark a transaction as settled

Command Line Interface (D)

1. Everything listed in C.3
2. Should be able to make an account, and delete account (given that debt = $\pm$ 0)

Database Architecture (E)

1. User information
    1. User ID

    2. Contact info
    3. Associated Groups
    4. Public Key
2. Transaction Information
    1. Transaction ID
    2. Payee
    3. Recipient
    4. Transaction reference
    5. Amount (£)
    6. Payee's signature
    7. Recipient's signature
    8. Whether or not transaction has been settled
3. Group information
    1. Group name
    2. People in the group
    3. Transactions in the group

---

1. based on a heuristic model ↩
2. shortest here referring to fewest edges traversed (not accounting for edge weight) ↩
3. Due to the heuristic nature of the model there will likely be multiple valid settled graphs ↩