INTRODUCTION TO DATA STRUCTURE &

ALGORITHM – FALL 2022

CS512 – FINAL PROJECT

DEBT SIMPLIFICATION USING DINIC'S MAXIMUM-FLOW ALGORITHM

SUBMITTED BY:

GROUP – 31

DHRUVI LALIT JAIN

DJ470

SIDDHARTH DEVENDRA GUPTA

SDG141

KANCHU KIRAN

KK1219

SHRADDHA SANJEEV PATTANSHETTI

SP2304

TABLE OF CONTENT

1. Abstract

Problem Statement

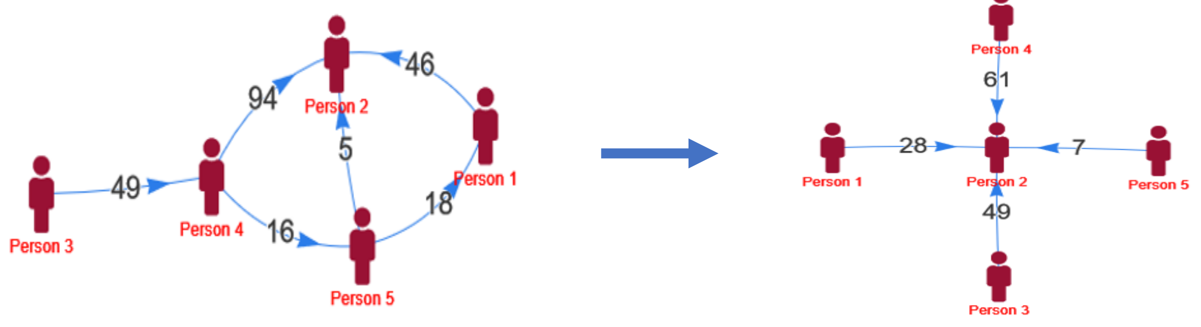
A group of friends lend each other money throughout the year. They carefully record each transaction and at the end of the year wish to settle their debts.

- How should they transfer money so as to settle all debts?
- How difficult is it to find an appropriate settling scheme?
- How efficient is that scheme?
- Try to minimize the number of transfers and the total amount transferred.

Some payments get mixed up in groups whenever we go out, and someone needs to catch up on the payments. The dues keep on piling up.

Simplification of debts is the approach of restructuring the debt within a group of people. It reduces the number of payments without actually changing the total amount owed.

This is a common problem for groups of friends or roommates, and the solution is implemented in various apps and websites, such as Split wise, Tricount, and others.



2. Introduction

In our project, each person will be represented as a node of a directed graph, and each edge of this graph will represent the transaction amount. We will simplify this graph to get the least number of transactions.

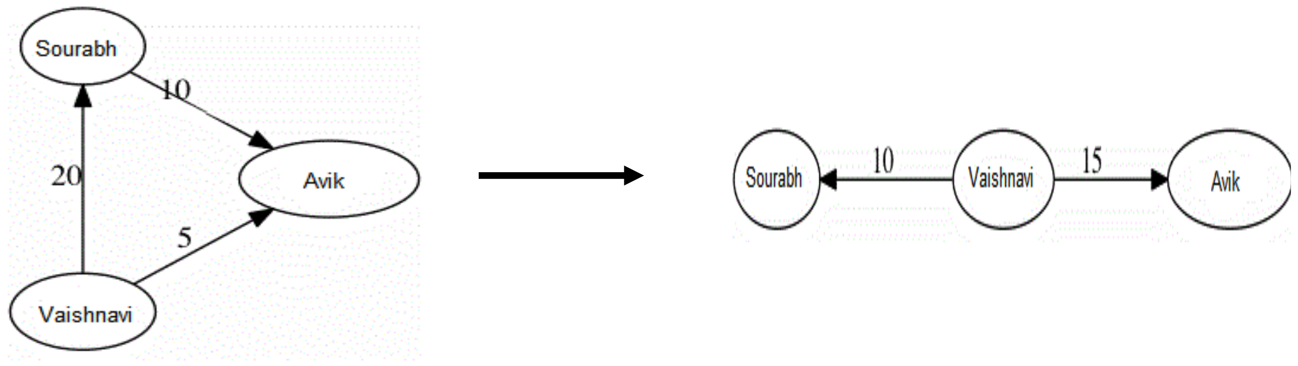
To minimize the total payments made for resolving debts, the Debtor must transfer money only to the Recipient, and the Receiver must receive money only from the giver. Furthermore, it can be noted that the total money owed by a person always equals the total money received by a person.

We aim to reduce the no of transactions; in terms of the graph, we should minimize no of edges. This can be achieved by maximizing the edge weight between two users, possibly minimizing the number of edges.

Consider an example given below,

Saurabh and Vaishnavi owe Avik 10 dollars and 5 dollars each. In addition to this, Vaishnavi owes 20 dollars to Sourabh. The number of transactions, in this case, is 3, which can be reduced to 2.

Since Vaishnavi is beholden to both Sourabh and Avik, she must pay Sourabh 10 instead of 20 dollars and Avik 15 dollars on behalf of Sourabh. This way, complications are mitigated with the help of the simplification debt feature.



3. Design

There are three essential rules we need to follow while designing the algorithm:

- Everyone owes the same net amount at the end
- No one owes a person that they didn't owe before
- No one owes more money in total than they did before the simplification.

Overview of Algorithm

- Feed the debts in the form of a directed graph (let's represent it by G) to the algorithm. • Select one of the non-visited edges, say (u, v) from the directed graph G.
- Now, with u as the source and v as the sink, run a maxflow algorithm - Dinic's maxflow algorithm, since it's one of the optimal implementations, to determine the maximum flow of money possible from u to v.
- Also, compute the Residual Graph (let's represent it by G'), which indicates the additional possible flow of debts in the input graph after removing the discharge of debts between u and v.
- If the maximum flow between u and v (let's represent it by max-flow) is greater than zero, then add an edge (u, v) with weight as max-flow to the Residual Graph.
- Now go back to Step 1 and feed it the Residual Graph G'.
- Once all the edges are visited, the Residual Graph G' obtained in the final iteration will have the minimum number of edges (i.e., transactions).

4. Algorithm

The algorithm used to implement the debt simplification Project is **Dinic's Algorithm**. It is a Maximum Flow Algorithm. Its run time is independent of the flow graph's capacity value, which could be very large. It has a runtime of O (V^2 E). It tremendously increases the performance of bipartite graphs giving a time complexity of O((√VE)).

We perform BFS on the source and obtain a level graph. An edge is a part of the level graph if it follows the path to the sink. This means it must advance from one level to another after omitting back and side edges.

<u>Algorithm work as follows:</u>

- Perform BFS from the source vertex.
- Construct a level graph as you perform BFS and mark all the levels of the flow graph.
- Check if the sink is reached while keeping the levels; if more flow is impossible, return the max flow.

- Omit back and side edges and create level graphs. Keep performing BFS from source to sink until we cannot find any more paths between the source and sink since the edges have been saturated. Add the values of edges leading into the sink or the minimum value to calculate the maximum flow.

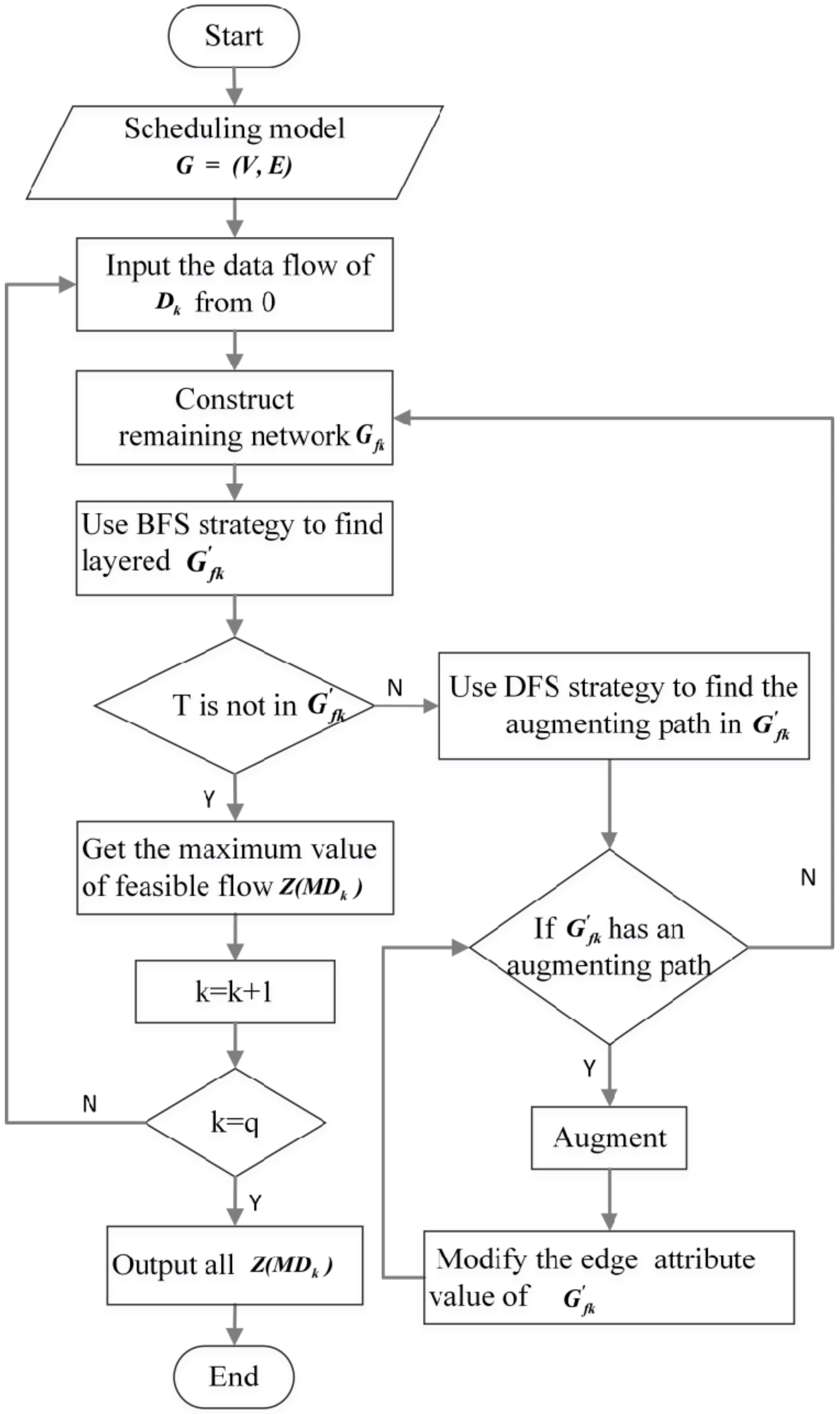
- Again, build the level graph with the edges containing the remaining capacities and perform steps 1, 2, and 3 until any more flow cannot be pushed through the network leading to the termination of the algorithm.

- A problem with Dinic's algorithm is that it might encounter many dead ends while performing BFS. A dead end occurs when we cannot reach a sink from a source through the path we follow, so we backtrack and try a different way to get the sink.

- We prune the dead ends when encountered so that they are located only once and do not see the same dead end repeatedly. We use this approach to simplify debts which is an addition to the original Dinic's algorithm.

- This means if person A owes money to person B and person B owes money to person C. It will simplify the debts and ask person C to pay money to person A if person C is unwilling to pay money to person A (dead-end). It will ask person B to pay money to person A, who originally was indebted to person A.

Above can be explained with the below flowchart



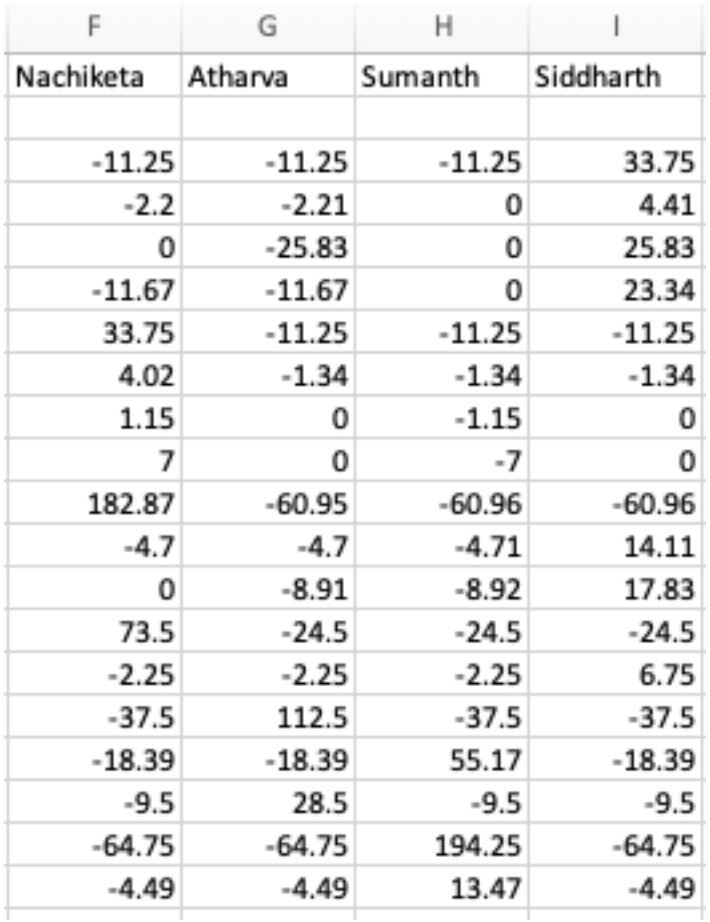
Start
Scheduling model $G = (V, E)$
Input the data flow of D_k from 0
Construct remaining network G_{fk}
Use BFS strategy to find layered G'_{fk}
T is not in G'_{fk}
N
Use DFS strategy to find the augmenting path in G'_{fk}
Y
Get the maximum value of feasible flow $Z(MD_k)$
k=k+1
N
k=q
Y
Output all $Z(MD_k)$
End
N
If G'_{fk} has an augmenting path
Y
Augment
Modify the edge attribute value of G'_{fk}

5. Implementation

We took real time data by importing csv file from splitwise app, which contained information regarding transactions between individuals.

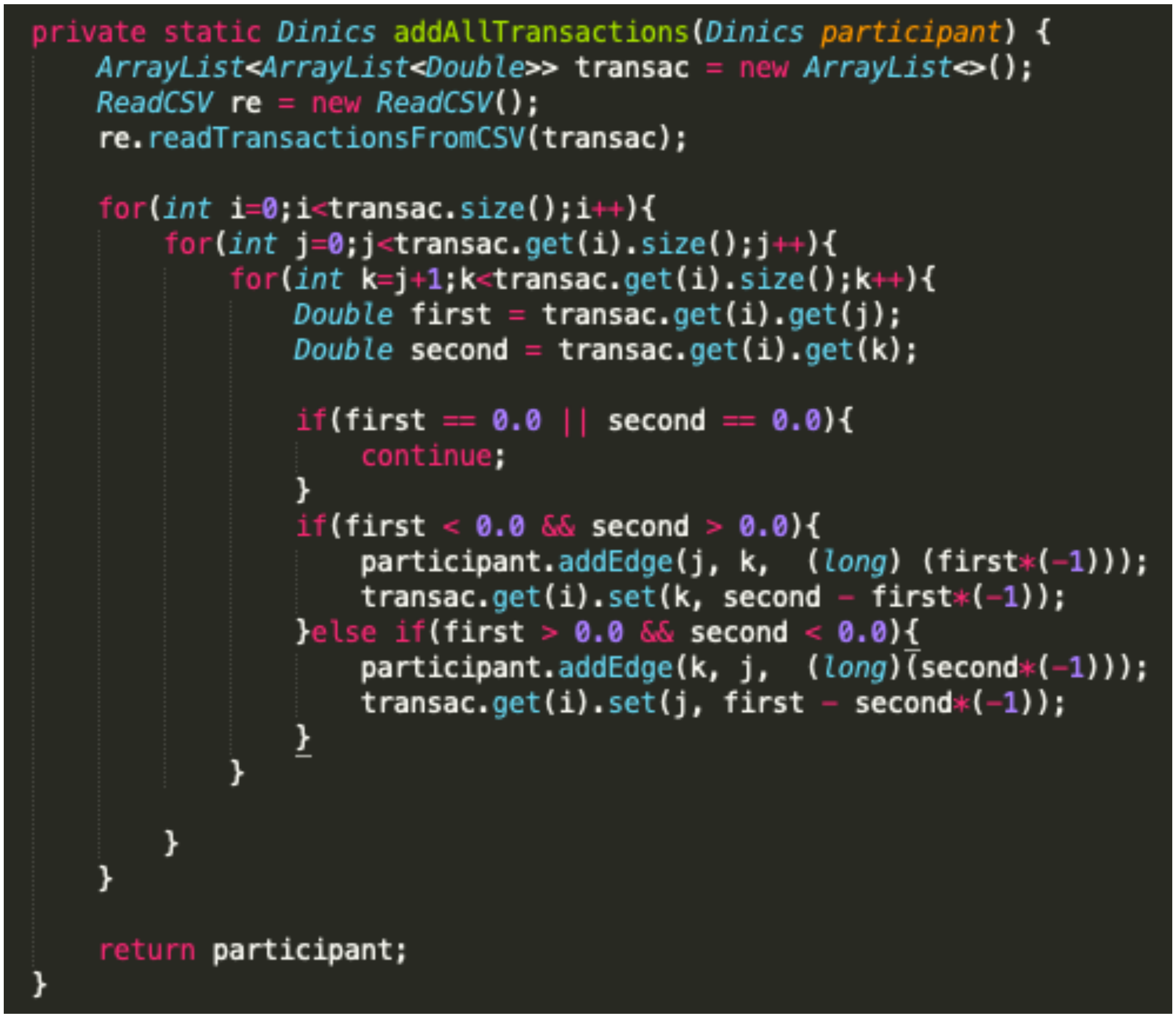
5.1. Stepwise Procedure to minimize transactions:

- **Step1:** The file is parsed a 3-tuple is created which contains information regarding (debtor, creditor, amount). A directed edge is created from debtor to creditor and a weight is assigned to it which is equal to the amount owed by debtor to creditor and a graph G is generated.



F	G	H	I
Nachiketa	Atharva	Sumanth	Siddharth
-11.25	-11.25	-11.25	33.75
-2.2	-2.21	0	4.41
0	-25.83	0	25.83
-11.67	-11.67	0	23.34
33.75	-11.25	-11.25	-11.25
4.02	-1.34	-1.34	-1.34
1.15	0	-1.15	0
7	0	-7	0
182.87	-60.95	-60.96	-60.96
-4.7	-4.7	-4.71	14.11
0	-8.91	-8.92	17.83
73.5	-24.5	-24.5	-24.5
-2.25	-2.25	-2.25	6.75
-37.5	112.5	-37.5	-37.5
-18.39	-18.39	55.17	-18.39
-9.5	28.5	-9.5	-9.5
-64.75	-64.75	194.25	-64.75
-4.49	-4.49	13.47	-4.49

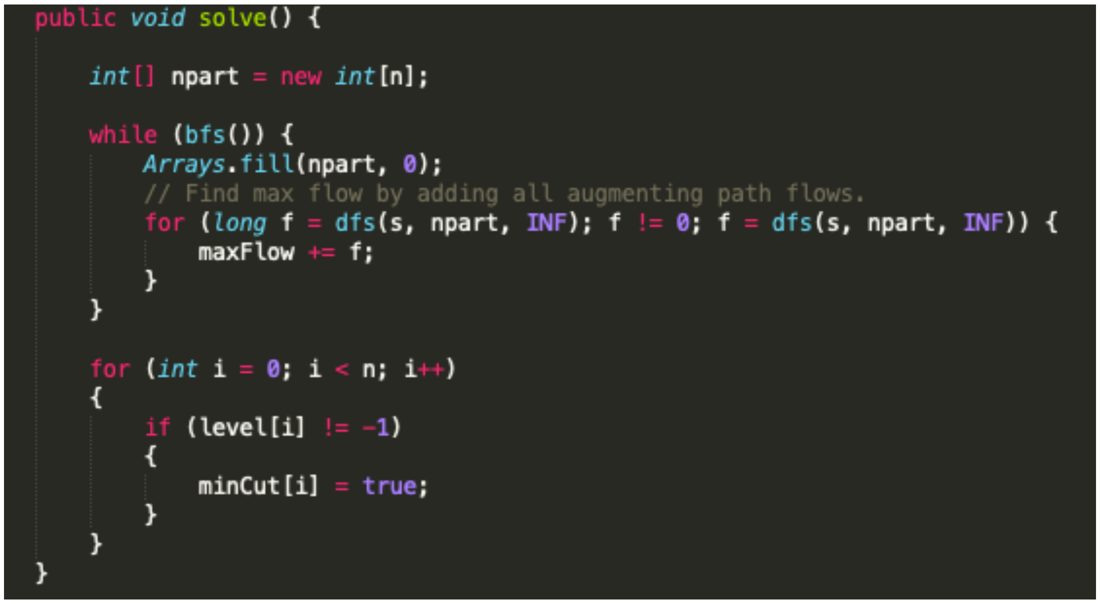
- **Step 2:** Initialize residual graph G as generated graph from the data parsed from csv file.



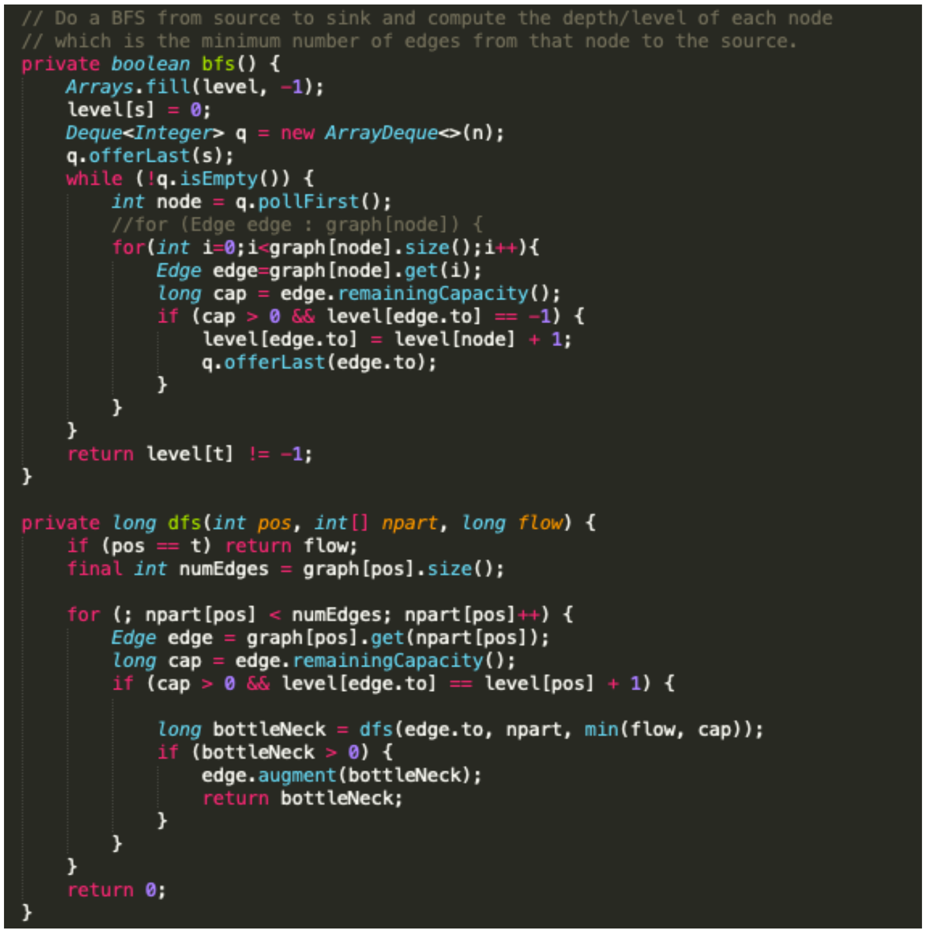
```java
private static Dinics addAllTransactions(Dinics participant) {
    ArrayList<ArrayList<Double>> transac = new ArrayList<>();
    ReadCSV re = new ReadCSV();
    re.readTransactionsFromCSV(transac);

    for(int i=0;i<transac.size();i++){
        for(int j=0;j<transac.get(i).size();j++){
            for(int k=j+1;k<transac.get(i).size();k++){
                Double first = transac.get(i).get(j);
                Double second = transac.get(i).get(k);

                if(first == 0.0 || second == 0.0){
                    continue;
                }
                if(first < 0.0 && second > 0.0){
                    participant.addEdge(j, k,  (long) (first*(-1)));
                    transac.get(i).set(k, second - first*(-1));
                }else if(first > 0.0 && second < 0.0){
                    participant.addEdge(k, j,  (long)(second*(-1)));
                    transac.get(i).set(j, first - second*(-1));
                }
            }

        }
    }

    return participant;
}
```

- **Step 3:** Call Dinics for all pair of vertices. Do BFS of G to construct a level graph (or assign levels to vertices) and also check if more flow is possible.
    - **Step 4:** If more flow is not possible, then return
    - **Step 5:** Send multiple flows in G using level graph until blocking flow is reached.

Dinic network solver code, which calls BFS from source to sink and compute depth/level of each node:

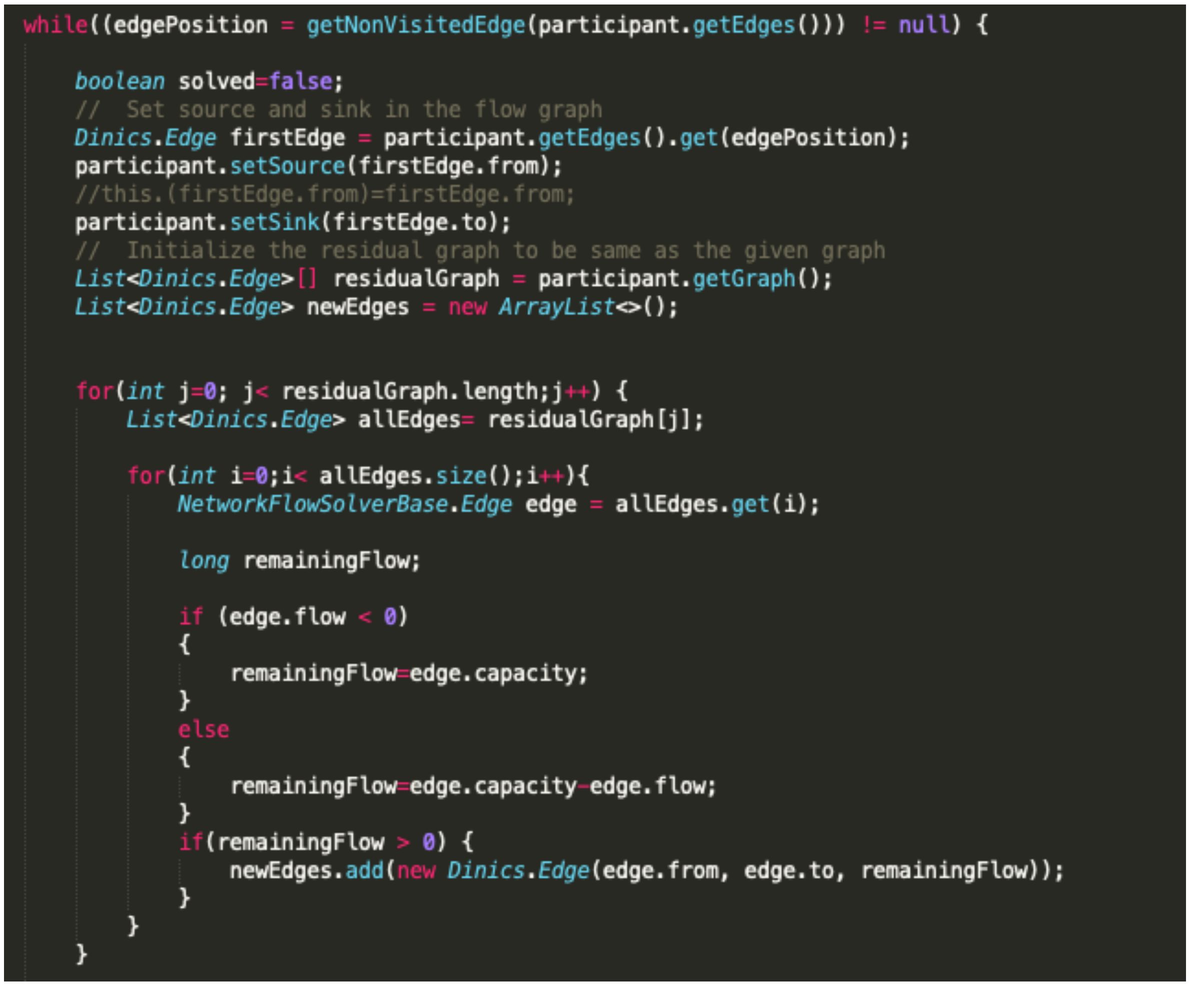


```
public void solve() {

    int[] npart = new int[n];

    while (bfs()) {
        Arrays.fill(npart, 0);
        // Find max flow by adding all augmenting path flows.
        for (long f = dfs(s, npart, INF); f != 0; f = dfs(s, npart, INF)) {
            maxFlow += f;
        }
    }

    for (int i = 0; i < n; i++)
    {
        if (level[i] != -1)
        {
            minCut[i] = true;
        }
    }
}
```



```
// Do a BFS from source to sink and compute the depth/level of each node
// which is the minimum number of edges from that node to the source.
private boolean bfs() {
    Arrays.fill(level, -1);
    level[s] = 0;
    Deque<Integer> q = new ArrayDeque<>(n);
    q.offerLast(s);
    while (!q.isEmpty()) {
        int node = q.pollFirst();
        //for (Edge edge : graph[node]) {
        for(int i=0;i<graph[node].size();i++){
            Edge edge=graph[node].get(i);
            long cap = edge.remainingCapacity();
            if (cap > 0 && level[edge.to] == -1) {
                level[edge.to] = level[node] + 1;
                q.offerLast(edge.to);
            }
        }
    }
    return level[t] != -1;
}

private long dfs(int pos, int[] npart, long flow) {
    if (pos == t) return flow;
    final int numEdges = graph[pos].size();

    for (; npart[pos] < numEdges; npart[pos]++) {
        Edge edge = graph[pos].get(npart[pos]);
        long cap = edge.remainingCapacity();
        if (cap > 0 && level[edge.to] == level[pos] + 1) {

            long bottleNeck = dfs(edge.to, npart, min(flow, cap));
            if (bottleNeck > 0) {
                edge.augment(bottleNeck);
                return bottleNeck;
            }
        }
    }
    return 0;
}
```

- Here using level graph means, in every flow, levels of path nodes should be 0, 1, 2… (in order) from s to t.
- A flow is Blocking Flow if no more flow can be sent using level graph, i.e., no more s-t path exists such that path vertices have current levels 0, 1, 2… in order.



```java
while((edgePosition = getNonVisitedEdge(participant.getEdges())) != null) {

    boolean solved=false;
    //  Set source and sink in the flow graph
    Dinics.Edge firstEdge = participant.getEdges().get(edgePosition);
    participant.setSource(firstEdge.from);
    //this.(firstEdge.from)=firstEdge.from;
    participant.setSink(firstEdge.to);
    //  Initialize the residual graph to be same as the given graph
    List<Dinics.Edge>[] residualGraph = participant.getGraph();
    List<Dinics.Edge> newEdges = new ArrayList<>();


    for(int j=0; j< residualGraph.length;j++) {
        List<Dinics.Edge> allEdges= residualGraph[j];

        for(int i=0;i< allEdges.size();i++){
            NetworkFlowSolverBase.Edge edge = allEdges.get(i);

            long remainingFlow;

            if (edge.flow < 0)
            {
                remainingFlow=edge.capacity;
            }
            else
            {
                remainingFlow=edge.capacity-edge.flow;
            }
            if(remainingFlow > 0) {
                newEdges.add(new Dinics.Edge(edge.from, edge.to, remainingFlow));
            }
        }
    }
```

5.2. Time Complexity: O(EV^2).

- Doing a BFS to construct level graph takes O(E) time.
- Sending multiple more flows until a blocking flow is reached takes O(VE) time.
- The outer loop runs at-most O(V) time.
- In each iteration, we construct new level graph and find blocking flow. It can be proved that the number of levels increase at least by one in every iteration (Refer the below reference video for the proof). So, the outer loop runs at most O(V) times.
- Therefore, overall time complexity is O(EV^2).

6. Result and Analysis

6.1. Input: comma delimited csv file

File is read and parsed and (debtor, creditor, amount) is created and stored for graph edge creation. Below is a sample file read and tuple values:

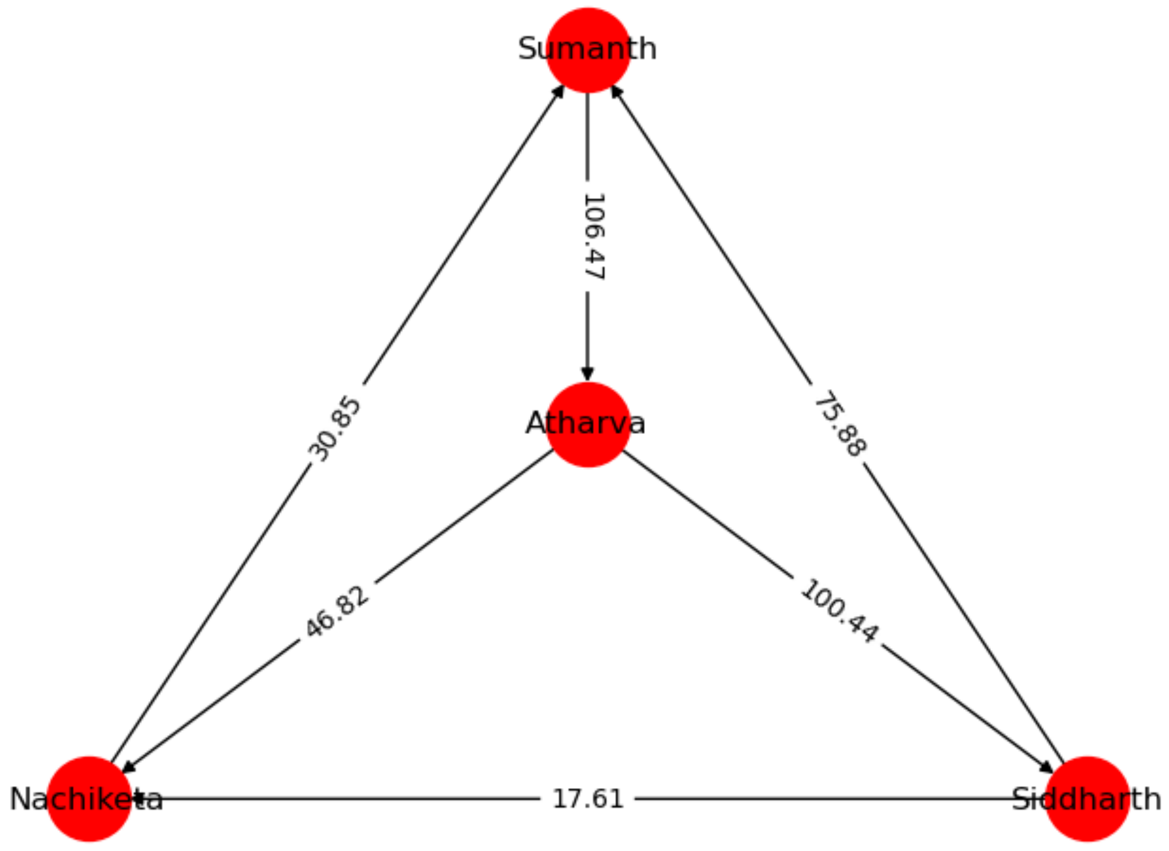
```
('Nachiketa Prasad', 'Siddharth Gupta', 11.25)
('Atharva', 'Siddharth Gupta', 11.25)
('Sumanth Tangirala', 'Siddharth Gupta', 11.25)
('Nachiketa Prasad', 'Siddharth Gupta', 2.2)
('Atharva', 'Siddharth Gupta', 2.21)
('Atharva', 'Siddharth Gupta', 25.83)
('Nachiketa Prasad', 'Siddharth Gupta', 11.67)
('Atharva', 'Siddharth Gupta', 11.67)
('Atharva', 'Nachiketa Prasad', 11.25)
('Sumanth Tangirala', 'Nachiketa Prasad', 11.25)
('Siddharth Gupta', 'Nachiketa Prasad', 11.25)
('Atharva', 'Nachiketa Prasad', 1.34)
('Sumanth Tangirala', 'Nachiketa Prasad', 1.34)
('Siddharth Gupta', 'Nachiketa Prasad', 1.34)
('Sumanth Tangirala', 'Nachiketa Prasad', 1.15)
('Sumanth Tangirala', 'Nachiketa Prasad', 7.0)
('Atharva', 'Nachiketa Prasad', 60.95)
('Sumanth Tangirala', 'Nachiketa Prasad', 60.96)
('Siddharth Gupta', 'Nachiketa Prasad', 60.96)
('Nachiketa Prasad', 'Siddharth Gupta', 4.7)
('Atharva', 'Siddharth Gupta', 4.7)
('Sumanth Tangirala', 'Siddharth Gupta', 4.71)
('Atharva', 'Siddharth Gupta', 8.91)
('Sumanth Tangirala', 'Siddharth Gupta', 8.92)
('Atharva', 'Nachiketa Prasad', 24.5)
('Sumanth Tangirala', 'Nachiketa Prasad', 24.5)
('Siddharth Gupta', 'Nachiketa Prasad', 24.5)
('Nachiketa Prasad', 'Siddharth Gupta', 2.25)
('Atharva', 'Siddharth Gupta', 2.25)
('Sumanth Tangirala', 'Siddharth Gupta', 2.25)
('Nachiketa Prasad', 'Atharva', 37.5)
('Sumanth Tangirala', 'Atharva', 37.5)
('Siddharth Gupta', 'Atharva', 37.5)
('Nachiketa Prasad', 'Sumanth Tangirala', 18.39)
('Atharva', 'Sumanth Tangirala', 18.39)
('Siddharth Gupta', 'Sumanth Tangirala', 18.39)
('Nachiketa Prasad', 'Atharva', 9.5)
('Sumanth Tangirala', 'Atharva', 9.5)
('Siddharth Gupta', 'Atharva', 9.5)
('Nachiketa Prasad', 'Sumanth Tangirala', 64.75)
('Atharva', 'Sumanth Tangirala', 64.75)
('Siddharth Gupta', 'Sumanth Tangirala', 64.75)
('Nachiketa Prasad', 'Sumanth Tangirala', 4.49)
('Atharva', 'Sumanth Tangirala', 4.49)
('Siddharth Gupta', 'Sumanth Tangirala', 4.49)
```

6.2. Graph creation

Graph is created with one directed edge for each transactions mentioned above. If there are multiple transactions from same creditor to same debtor, the weight is just updated to by adding the new amount.

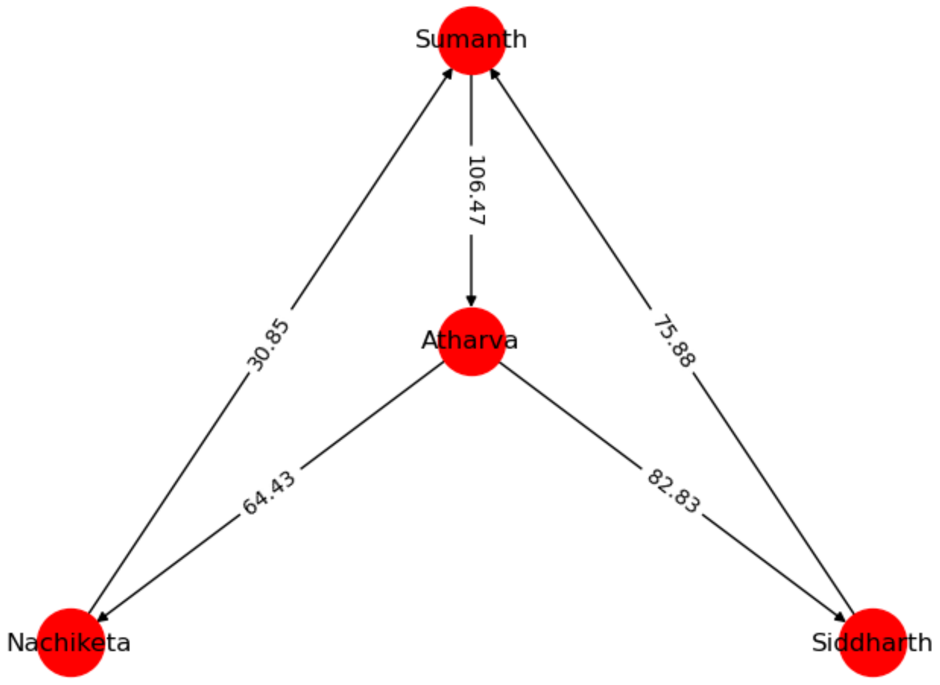
We can see that vertices in the below graph denotes the people involved in the transactions and there is one vertex created for every person in the group.

A directed edge is created for each effective transition from the above transaction tuples and weight is updated accordingly.

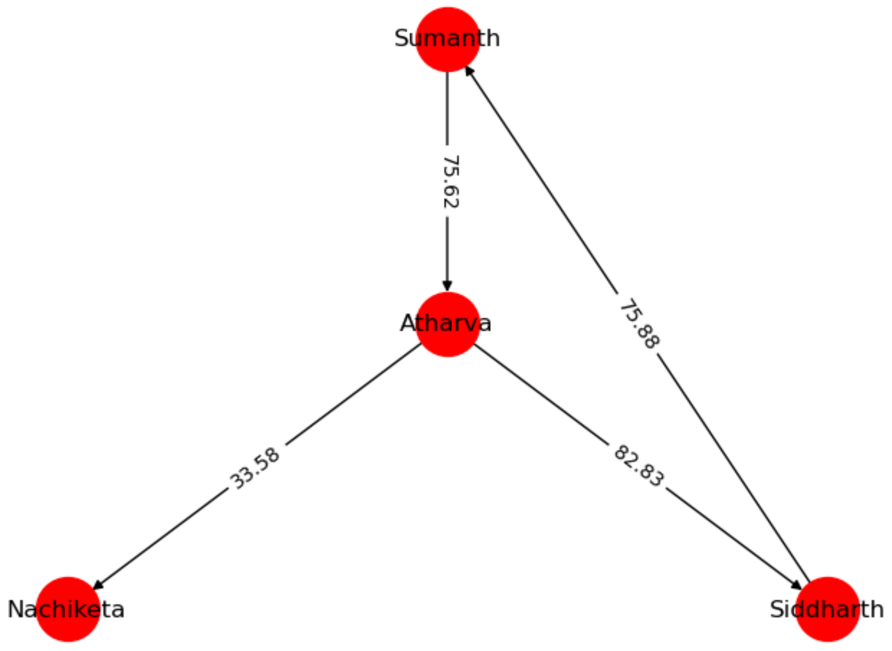


6.3 Transaction simplification using Dinic Algorithm

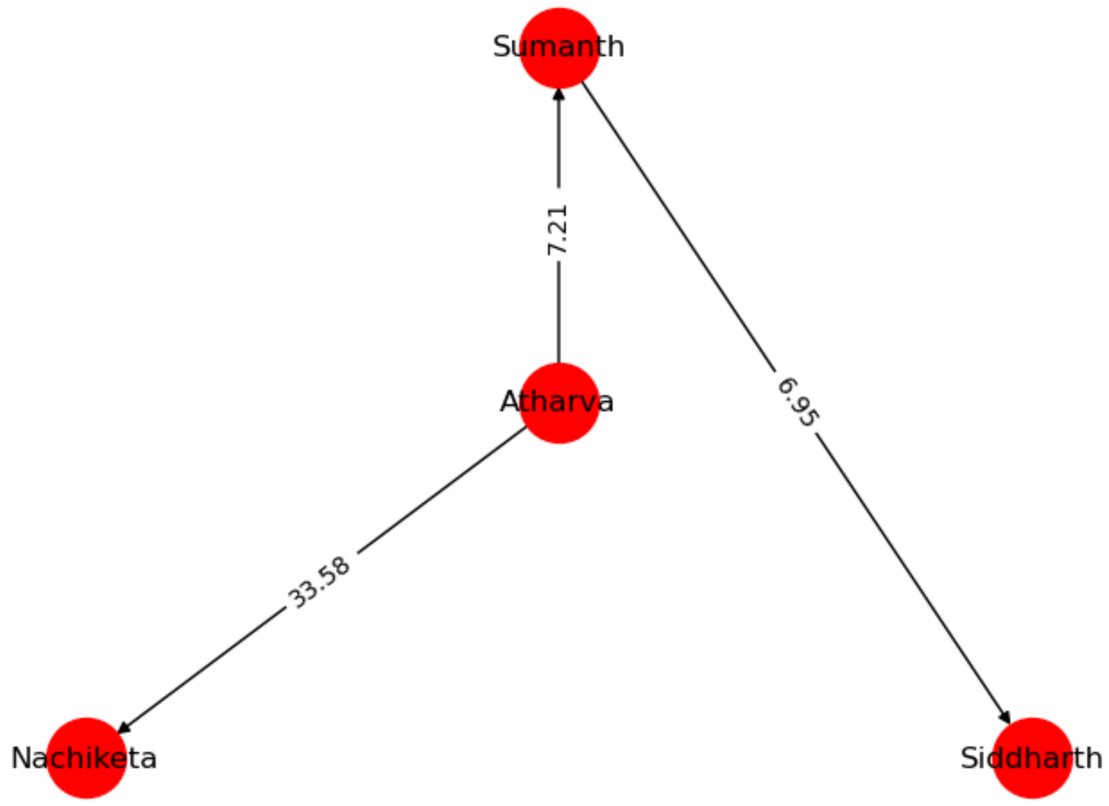
- After adjusting ['Nachiketa', 'Siddharth', 'Atharva'] by -17.61:



- After adjusting ['Nachiketa', 'Sumanth', 'Atharva'] by 30.85



- After adjusting ['Atharva', 'Siddharth', 'Sumanth'] by 82.83:



6.4 Final output

- Atharva owes Nachiketa Prasad $33.58
- Atharva owes Sumanth Tangirala $7.21
- Sumanth Tangirala owes Siddharth Gupta $6.95

7. Conclusion

People foot the bill for different expenses and need to get paid back later. Debt Simplification does all the bookkeeping for you, which helps you keep track of your debits and credits. Furthermore, it assists people in narrowing the complex task of splitting the bill among several people.

Many algorithms can be used to accomplish the task of implementing the application. A few are Karzanov, Galil, Tarjan, Bertsekas, Edmonds and Karp, Dinics, and many more. Of these, the Dinics algorithm is one of the best options in terms of Time complexity which is $O(n^2m)$.

8. References

- Debt Simplification Feature by Mithun Mohan K, Debt Simplification Techniques, Mithun Mohan K, 2019
- A Fast and Simple Algorithm for the Maximum Flow Problem (researchgate.net), James B. Orlin, Ravindra K. Ahuja, October 1989
- Settling Multiple Debts Efficiently: Problems Tom Verhoef, June 2004
- NP-Complete Splitting, May 2016
- https://terbium.io/2020/09/debt-simplification