CASH FLOW MINIMIZER

A PROJECT REPORT

Submitted by

AKSHAT MITTAL [RA2211003010790]
RITVIK RAJVANSHI [RA2211003010792]
YUGAM SHAH [RA2211003010796]

for the course 21CSC201J Data Structures and Algorithms

Under the Guidance of

Dr. Sindhuja M

Assistant Professor, Department of Computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

NOVEMBER 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that the 21CSC201J Data Structures and Algorithms course project report titled **"CASH FLOW MINIMIZER"** is the bonafide work done by **AKSHAT MITTAL [RA2211003010790], RITVIK RAJVANSHI [RA2211003010792] & YUGUAM SHAH [RA2211003010796]** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE
Faculty In-Charge
Dr. Sindhuja M,
Assistant Professor,
Department of Computing Technologies,
SRMIST.

HEAD OF THE DEPARTMENT
Dr. M. Pushpalatha,
Professor and Head,
Department of Computing Technologies,
SRMIST.

TABLE OF CONTENTS

CONTRIBUTION TABLE

NAME	CONTRIBUTION
AKSHAT	• Code Implementation • PPT – Problem statement and related • Problem definition and explanation by finding example. • Surfing for required material
RITVIK	• Code Implementation • PPT – Data Structure Explanation • Data Structure Explanation and connecting code • Surfing for required material
YUGAM	• Code Implementation • PPT – Output and Demo Simulation calculation • Complexity Analysis and Future Scope • Surfing for required material

PROBLEM DEFINITION

The “Cash Flow Minimizer” project addresses the optimization of cash flow within a group of individuals who have financial transactions and debts among themselves. The primary objective is to determine the smallest number of transactions required to settle all debts within the group efficiently.

The "Cash Flow Minimizer" project aims to address the challenge of efficiently settling outstanding debts or financial obligations within a group of individuals or entities. In this problem, a group of N participants is involved, each having financial transactions with others within the group. The goal is to minimize the total number of cash flow transactions needed to clear these financial relationships. To do so, the project seeks to design and implement an algorithm that, given the financial relationships among the participants, identifies the most efficient way to settle debts or credits, ultimately reducing the overall financial complexity and ensuring a fair distribution of funds. This problem can be approached using data structures and algorithms, primarily through graph theory and optimization techniques, making it an intriguing challenge for algorithmic optimization.

PROBLEM EXPLANATION

Imagine a scenario where individuals within a group owe money to each other. This interconnected web of debts forms a complex graph. Each person represents a vertex, and each financial transaction is an edge connecting two individuals. The task is to settle these debts using the least number of transactions.

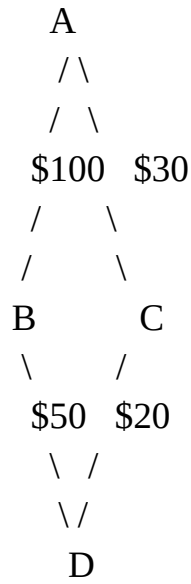
The "Cash Flow Minimizer" problem revolves around the optimization of cash flow transactions within a group of individuals or entities with interrelated financial obligations. The central challenge is to streamline the settlement of debts or credits among the participants in a way that minimizes the number of transactions needed. This problem is not only of practical significance for efficient financial management but also an intriguing computational challenge, where data structures and algorithmic techniques can be leveraged to find an optimal solution. By reducing the financial complexity and ensuring fairness in the distribution of funds, the Cash Flow Minimizer project aims to develop a practical and efficient approach to handling intricate financial relationships within a group.

Example:

Suppose there are four individuals: A, B, C, and D, and they have the following financial relationships:

- A owes $100 to B.
- B owes $50 to C.
- C owes $30 to A.
- D owes $20 to A.

We can represent these relationships as a graph:



In this diagram, the arrows represent the direction of the financial transactions, and the numbers indicate the amounts involved. The goal of the Cash Flow Minimizer project is to find the most efficient way to settle these debts, reducing the number of transactions.

In this example, a possible optimal solution could be:

1. A pay $30 to C, and the debt between A and C is settled.
2. A pay $20 to D, and the debt between A and D is settled.
3. A pay $50 to B, and the debt between A and B is settled.

This results in a minimal number of transactions, simplifying the financial relationships within the group. The Cash Flow Minimizer project aims to develop algorithms to automate this process, making it more efficient for larger and more complex financial networks.

DATA STRUCTURE USED

Using a graph data structure is a common and efficient approach for solving the Cash Flow Minimizer problem. The financial relationships among individuals or entities can be naturally represented as a directed graph. Let's elaborate on how to use a graph data structure for this problem:

1. **Nodes (Vertices):** Each individual or entity in the group is represented as a node in the graph. These nodes contain information about the participants and their financial positions.

2. **Edges:** Edges in the graph represent financial transactions. Specifically, a directed edge from node A to node B indicates that A owes money to B. The weight of the edge represents the amount of the debt or credit. A positive weight represents money owed, while a negative weight represents money lent.

3. **Graph Construction:** To build the graph, you take the list of financial transactions between participants and create nodes for each participant and directed edges to represent the financial relationships. For example, in Python, you can use a dictionary to represent the graph, where the keys are participants, and the values are lists of tuples (target, amount) indicating the target of the transaction and the amount.

4. **Graph Algorithms:** Once you've constructed the graph, you can apply various graph algorithms to find an optimal solution. Common algorithms and techniques include:

    - Minimum Spanning Tree (MST): You can find the minimum spanning tree of the graph (e.g., using Kruskal's algorithm). The edges in the MST represent a minimal set of transactions to settle all debts.

    - Flow Network Algorithms: Techniques like the Ford-Fulkerson algorithm can be used to find maximum flow and minimum cut, which help identify efficient debt settlement paths.

- Depth-First Search (DFS) or Breadth-First Search (BFS): These algorithms can help traverse the graph and identify interconnected debt relationships, simplifying the settlement process.

5. **Transaction Generation:** After applying the appropriate algorithm, you can generate a list of transactions that need to occur to settle the debts. This list will contain the debtor, creditor, and the amount to be transferred for each transaction.

By using a graph data structure and graph algorithms, you can efficiently analyse and optimize cash flow within the group, reducing the number of transactions required to settle financial obligations while ensuring fairness and simplicity in the financial relationships.

ALGORITHM OF THE PROBLEM

The "Cash Flow Minimizer" problem can be solved using various algorithms and approaches. One common approach is to find the minimum number of transactions required to settle debts efficiently. Here's a high-level algorithm for solving this problem:

Algorithm: Cash Flow Minimizer

1. **Create a Directed Graph:** Construct a directed graph where nodes represent individuals or entities, and directed edges represent the financial transactions, with edge weights indicating the amount owed or lent.

2. **Calculate Net Balances:** For each node (individual or entity) in the graph, calculate the net balance by summing up the total amount owed and the total amount owed to them. This creates a simplified representation of each participant's overall financial position.

3. **Identify Debtor-Creditor Pairs:** Find pairs of individuals/entities where one owes money to the other (i.e., positive-negative debt relationships). Create a list of debtor-creditor pairs, including the amount to be settled.

4. **Optimization Algorithm:**
   a. Initialize an empty list to store the transactions.
   b. While there are debtor-creditor pairs to be settled:
   i. Select a debtor-creditor pair with the largest absolute debt.
   ii. Determine the amount to transfer (minimum of the debt of the debtor and the credit of the creditor).
   iii. Create a transaction record with the debtor, creditor, and the amount to transfer.
   iv. Update the balances of the debtor and creditor by deducting the transferred amount.
   v. If the debtor's balance becomes zero, remove them from the list of debtors. If the creditor's balance becomes zero, remove them from the list of creditors.
   vi. Add the transaction to the list of transactions.
   c. Repeat step 4b until there are no more debtor-creditor pairs.

5. **Output:** Return the list of transactions as the minimal set of transactions needed to settle all debts efficiently.

This algorithm efficiently minimizes the number of transactions required to clear the financial relationships within the group, ensuring fairness and simplicity in the cash flow. Depending on the complexity and size of the problem, you can implement various graph algorithms and data structures to optimize the process further.

EXPLANATION OF ALGORITHM USING EXAMPLE

Suppose we have a group of four individuals (A, B, C, and D) with the following financial relationships:

- A owes $100 to B.
- B owes $50 to C.
- C owes $30 to A.
- D owes $20 to A.

We want to minimize the number of transactions to settle these debts efficiently.

Step 1: Create a Directed Graph

We represent these financial relationships as a directed graph:

Step 2: Calculate Net Balances

We calculate the net balances for each individual/entity:

- A: $(100 - 30 - 20) = $50 (A owes B and D).
- B: $50 (B owes C).
- C: $30 (C owes A).
- D: $20 (D owes A).

Step 3: Identify Debtor-Creditor Pairs

We identify the debtor-creditor pairs with positive-negative debt relationships:

- A owes $100 to B.
- B owes $50 to C.
- C owes $30 to A.
- D owes $20 to A.

Step 4: Optimization Algorithm

Now, we optimize the transactions:

a. Initialize an empty list of transactions.
b. While there are debtor-creditor pairs to be settled:

i. Select the debtor-creditor pair with the largest absolute debt, which is A owing $100 to B.

ii. Determine the amount to transfer. In this case, the minimum of A's debt ($100) and B's credit ($50) is $50.

iii. Create a transaction: A pays $50 to B.

iv. Update the balances:
- A's balance: $50 (original balance) - $50 (transferred) = $0.
- B's balance: $50 (original balance) - $50 (transferred) = $0.

v. Since both A and B have zero balances, remove them from the debtor and creditor lists.

vi. Add the transaction (A pays $50 to B) to the list of transactions.

Now, we have:

- C: $30 (C owes A).
- D: $20 (D owes A).

Continue with the next largest debt, which is C owing $30 to A:

i. Select the debtor-creditor pair: C owing $30 to A.
ii. Determine the amount to transfer: $30.
iii. Create a transaction: C pays $30 to A.
iv. Update the balances:
    - C's balance: $30 (original balance) - $30 (transferred) = $0.
    - A's balance: $0 (A's balance already settled in the previous transaction).
v. Remove C and A from the debtor and creditor lists.
vi. Add the transaction (C pays $30 to A) to the list of transactions.

Finally, we have D owing $20 to A:

i. Select the debtor-creditor pair: D owing $20 to A.
ii. Determine the amount to transfer: $20.
iii. Create a transaction: D pays $20 to A.
iv. Update the balances:
    - D's balance: $20 (original balance) - $20 (transferred) = $0.
    - A's balance: $0 (A's balance already settled in the previous transactions).
v. Remove D and A from the debtor and creditor lists.
vi. Add the transaction (D pays $20 to A) to the list of transactions.

Step 5: Output

The list of transactions that minimizes the cash flow is as follows:

1. A pay $50 to B.
2. C pays $30 to A.
3. D pays $20 to A.

With these transactions, all debts are efficiently settled, and the financial relationships within the group are simplified, with no outstanding balances.

CODE

```
#include <bits/stdc++.h>
using namespace std;

class bank{
    public:
    string name;
    int netAmount;
    set<string> types;
};

int getMinIndex(bank listOfNetAmounts[],int numBanks){
    int min=INT_MAX, minIndex=-1;
    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) continue;

        if(listOfNetAmounts[i].netAmount < min){
            minIndex = i;
            min = listOfNetAmounts[i].netAmount;
        }
    }
    return minIndex;
}

int getSimpleMaxIndex(bank listOfNetAmounts[],int numBanks){
    int max=INT_MIN, maxIndex=-1;
    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) continue;

        if(listOfNetAmounts[i].netAmount > max){
            maxIndex = i;
            max = listOfNetAmounts[i].netAmount;
        }
    }
    return maxIndex;
}
```

```
pair<int,string> getMaxIndex(bank listOfNetAmounts[],int numBanks,int
minIndex,bank input[],int maxNumTypes){
   int max=INT_MIN;
   int maxIndex=-1;
   string matchingType;

   for(int i=0;i<numBanks;i++){
      if(listOfNetAmounts[i].netAmount == 0) continue;

      if(listOfNetAmounts[i].netAmount < 0) continue;

      //TODO
      //see complexity of intersection

      vector<string> v(maxNumTypes);
      vector<string>::iterator
ls=set_intersection(listOfNetAmounts[minIndex].types.begin(),listOfNetAmoun
ts[minIndex].types.end(),
listOfNetAmounts[i].types.begin(),listOfNetAmounts[i].types.end(), v.begin());

      if((ls-v.begin())!=0 && max<listOfNetAmounts[i].netAmount ){
         max=listOfNetAmounts[i].netAmount;
         maxIndex=i;
         matchingType = *(v.begin());
      }
   }

   //if there is NO such max which has a common type with any remaining
banks then maxIndex has -1
   // also return the common payment type
   return make_pair(maxIndex,matchingType);
}

void printAns(vector<vector<pair<int,string>>> ansGraph, int numBanks,bank
input[]){
```

```
    cout<<"\nThe transactions for minimum cash flow are as follows : \n\n";
    for(int i=0;i<numBanks;i++){
        for(int j=0;j<numBanks;j++){

            if(i==j) continue;

            if(ansGraph[i][j].first != 0 && ansGraph[j][i].first != 0){

                if(ansGraph[i][j].first == ansGraph[j][i].first){
                    ansGraph[i][j].first=0;
                    ansGraph[j][i].first=0;
                }
                else if(ansGraph[i][j].first > ansGraph[j][i].first){
                    ansGraph[i][j].first -= ansGraph[j][i].first;
                    ansGraph[j][i].first =0;

                    cout<<input[i].name<<" pays Rs" << ansGraph[i][j].first<< "to
"<<input[j].name<<" via "<<ansGraph[i][j].second<<endl;
                }
                else{
                    ansGraph[j][i].first -= ansGraph[i][j].first;
                    ansGraph[i][j].first = 0;

                    cout<<input[j].name<<" pays Rs "<< ansGraph[j][i].first<<" to
"<<input[i].name<<" via "<<ansGraph[j][i].second<<endl;

                }
            }
            else if(ansGraph[i][j].first != 0){
                cout<<input[i].name<<" pays Rs "<<ansGraph[i][j].first<<" to
"<<input[j].name<<" via "<<ansGraph[i][j].second<<endl;

            }
            else if(ansGraph[j][i].first != 0){
                cout<<input[j].name<<" pays Rs "<<ansGraph[j][i].first<<" to
"<<input[i].name<<" via "<<ansGraph[j][i].second<<endl;
```

```
            }

            ansGraph[i][j].first = 0;
            ansGraph[j][i].first = 0;
        }
    }
    cout<<"\n";
}

void minimizeCashFlow(int numBanks,bank
input[],unordered_map<string,int>& indexOf,int
numTransactions,vector<vector<int>>& graph,int maxNumTypes){

    //Find net amount of each bank has
    bank listOfNetAmounts[numBanks];

    for(int b=0;b<numBanks;b++){
        listOfNetAmounts[b].name = input[b].name;
        listOfNetAmounts[b].types = input[b].types;

        int amount = 0;
        //incoming edges
        //column travers
        for(int i=0;i<numBanks;i++){
            amount += (graph[i][b]);
        }

        //outgoing edges
        //row traverse
        for(int j=0;j<numBanks;j++){
            amount += ((-1) * graph[b][j]);
        }

        listOfNetAmounts[b].netAmount = amount;
    }
```

```
    vector<vector<pair<int,string>>>
ansGraph(numBanks,vector<pair<int,string>>(numBanks,{0,""}));//adjacency
matrix



    //find min and max net amount
    int numZeroNetAmounts=0;

    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) numZeroNetAmounts++;
    }
    while(numZeroNetAmounts!=numBanks){

        int minIndex=getMinIndex(listOfNetAmounts, numBanks);
        pair<int,string> maxAns = getMaxIndex(listOfNetAmounts, numBanks,
minIndex,input,maxNumTypes);

        int maxIndex = maxAns.first;

        if(maxIndex == -1){

            (ansGraph[minIndex][0].first) +=
abs(listOfNetAmounts[minIndex].netAmount);
            (ansGraph[minIndex][0].second) = *(input[minIndex].types.begin());

            int simpleMaxIndex = getSimpleMaxIndex(listOfNetAmounts,
numBanks);
            (ansGraph[0][simpleMaxIndex].first) +=
abs(listOfNetAmounts[minIndex].netAmount);
            (ansGraph[0][simpleMaxIndex].second) =
*(input[simpleMaxIndex].types.begin());

            listOfNetAmounts[simpleMaxIndex].netAmount +=
listOfNetAmounts[minIndex].netAmount;
            listOfNetAmounts[minIndex].netAmount = 0;
```

```
            if(listOfNetAmounts[minIndex].netAmount == 0)
numZeroNetAmounts++;

            if(listOfNetAmounts[simpleMaxIndex].netAmount == 0)
numZeroNetAmounts++;

        }
        else{
            int transactionAmount =
min(abs(listOfNetAmounts[minIndex].netAmount),
listOfNetAmounts[maxIndex].netAmount);

            (ansGraph[minIndex][maxIndex].first) += (transactionAmount);
            (ansGraph[minIndex][maxIndex].second) = maxAns.second;

            listOfNetAmounts[minIndex].netAmount += transactionAmount;
            listOfNetAmounts[maxIndex].netAmount -= transactionAmount;

            if(listOfNetAmounts[minIndex].netAmount == 0)
numZeroNetAmounts++;

            if(listOfNetAmounts[maxIndex].netAmount == 0)
numZeroNetAmounts++;
        }

    }

    printAns(ansGraph,numBanks,input);
    // cout<<"HI\n";
}

//correct
int main()
{
    cout<<"\n\t\t\t\t******** WELCOME TO CASH FLOW MINIMIZER
SYSTEM ********\n\n";
```

```
    cout<<"This system minimizes the number of transactions among multiple banks in the different\n""corners of the world that use different modes of payment.\n""There is one world bank (with all payment modes) to act as an intermediary between\n""banks that have no common mode of payment. \n\n";
    cout<<"Enter the number of banks participating in the transactions.\n";
    int numBanks;cin>>numBanks;

    bank input[numBanks];
    unordered_map<string,int> indexOf;//stores index of a bank

    cout<<"Enter the details of the banks and transactions as stated:\n";
    cout<<"Bank name ,number of payment modes it has and the payment modes.\n";
    cout<<"Bank name and payment modes should not contain spaces\n";

    int maxNumTypes;
    for(int i=0; i<numBanks;i++){
        if(i==0){
            cout<<"World Bank : ";
        }
        else{
            cout<<"Bank "<<i<<" : ";
        }
        cin>>input[i].name;
        indexOf[input[i].name] = i;
        int numTypes;
        cin>>numTypes;

        if(i==0) maxNumTypes = numTypes;

        string type;
        while(numTypes--){
            cin>>type;

            input[i].types.insert(type);
        }
```

```
    }

    cout<<"Enter number of transactions.\n";
    int numTransactions;
    cin>>numTransactions;

    vector<vector<int>>
graph(numBanks,vector<int>(numBanks,0));//adjacency matrix

    cout<<"Enter the details of each transaction as stated:";
    cout<<"Debtor Bank , creditor Bank and amount\n";
    cout<<"The transactions can be in any order\n";
    for(int i=0;i<numTransactions;i++){
        cout<<(i)<<" th transaction : ";
        string s1,s2;
        int amount;
        cin >> s1>>s2>>amount;

        graph[indexOf[s1]][indexOf[s2]] = amount;
    }

    //settle

minimizeCashFlow(numBanks,input,indexOf,numTransactions,graph,maxNum
Types);
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;

class bank{
    public:
    string name;
    int netAmount;
    set<string> types;
};

int getMinIndex(bank listOfNetAmounts[],int numBanks){
    int min=INT_MAX, minIndex=-1;
```

```
    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) continue;

        if(listOfNetAmounts[i].netAmount < min){
            minIndex = i;
            min = listOfNetAmounts[i].netAmount;
        }
    }
    return minIndex;
}

int getSimpleMaxIndex(bank listOfNetAmounts[],int numBanks){
    int max=INT_MIN, maxIndex=-1;
    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) continue;

        if(listOfNetAmounts[i].netAmount > max){
            maxIndex = i;
            max = listOfNetAmounts[i].netAmount;
        }
    }
    return maxIndex;
}

pair<int,string> getMaxIndex(bank listOfNetAmounts[],int numBanks,int
minIndex,bank input[],int maxNumTypes){
    int max=INT_MIN;
    int maxIndex=-1;
    string matchingType;

    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) continue;

        if(listOfNetAmounts[i].netAmount < 0) continue;

        //TODO
        //see complexity of intersection

        vector<string> v(maxNumTypes);
        vector<string>::iterator
ls=set_intersection(listOfNetAmounts[minIndex].types.begin(),listOfNetAmounts[
minIndex].types.end(),
listOfNetAmounts[i].types.begin(),listOfNetAmounts[i].types.end(), v.begin());

        if((ls-v.begin())!=0 && max<listOfNetAmounts[i].netAmount ){
            max=listOfNetAmounts[i].netAmount;
            maxIndex=i;
            matchingType = *(v.begin());
```

```
        }
    }

    //if there is NO such max which has a common type with any remaining banks
then maxIndex has -1
    // also return the common payment type
    return make_pair(maxIndex,matchingType);
}

void printAns(vector<vector<pair<int,string>>> ansGraph, int numBanks,bank
input[]){

    cout<<"\nThe transactions for minimum cash flow are as follows : \n\n";
    for(int i=0;i<numBanks;i++){
        for(int j=0;j<numBanks;j++){

            if(i==j) continue;

            if(ansGraph[i][j].first != 0 && ansGraph[j][i].first != 0){

                if(ansGraph[i][j].first == ansGraph[j][i].first){
                    ansGraph[i][j].first=0;
                    ansGraph[j][i].first=0;
                }
                else if(ansGraph[i][j].first > ansGraph[j][i].first){
                    ansGraph[i][j].first -= ansGraph[j][i].first;
                    ansGraph[j][i].first =0;

                    cout<<input[i].name<<" pays Rs" << ansGraph[i][j].first<<
"to "<<input[j].name<<" via "<<ansGraph[i][j].second<<endl;
                }
                else{
                    ansGraph[j][i].first -= ansGraph[i][j].first;
                    ansGraph[i][j].first = 0;

                    cout<<input[j].name<<" pays Rs "<< ansGraph[j][i].first<<"
to "<<input[i].name<<" via "<<ansGraph[j][i].second<<endl;

                }
            }
            else if(ansGraph[i][j].first != 0){
                cout<<input[i].name<<" pays Rs "<<ansGraph[i][j].first<<" to
"<<input[j].name<<" via "<<ansGraph[i][j].second<<endl;

            }
            else if(ansGraph[j][i].first != 0){
                cout<<input[j].name<<" pays Rs "<<ansGraph[j][i].first<<" to
"<<input[i].name<<" via "<<ansGraph[j][i].second<<endl;
```

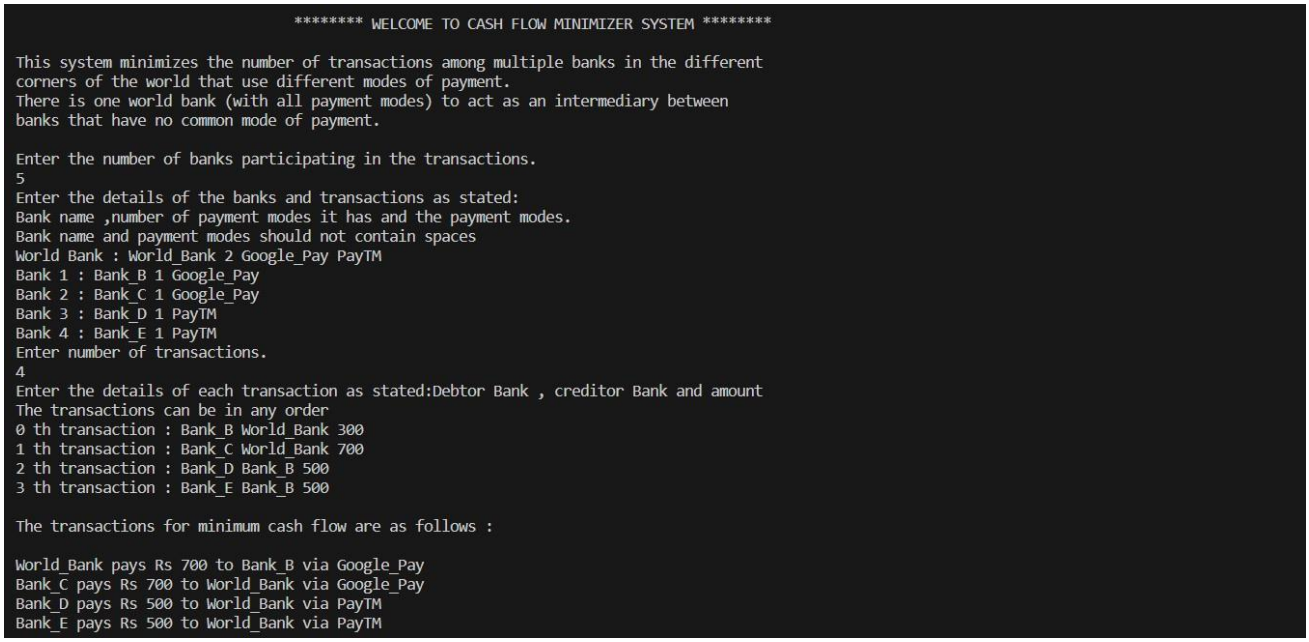
```
            }

            ansGraph[i][j].first = 0;
            ansGraph[j][i].first = 0;
        }
    }
    cout<<"\n";
}

void minimizeCashFlow(int numBanks,bank input[],unordered_map<string,int>&
indexOf,int numTransactions,vector<vector<int>>& graph,int maxNumTypes){

    //Find net amount of each bank has
    bank listOfNetAmounts[numBanks];

    for(int b=0;b<numBanks;b++){
        listOfNetAmounts[b].name = input[b].name;
        listOfNetAmounts[b].types = input[b].types;

        int amount = 0;
        //incoming edges
        //column travers
        for(int i=0;i<numBanks;i++){
            amount += (graph[i][b]);
        }

        //outgoing edges
        //row traverse
        for(int j=0;j<numBanks;j++){
            amount += ((-1) * graph[b][j]);
        }

        listOfNetAmounts[b].netAmount = amount;
    }

    vector<vector<pair<int,string>>>
ansGraph(numBanks,vector<pair<int,string>>(numBanks,{0,""}));//adjacency
matrix


    //find min and max net amount
    int numZeroNetAmounts=0;

    for(int i=0;i<numBanks;i++){
        if(listOfNetAmounts[i].netAmount == 0) numZeroNetAmounts++;
    }
    while(numZeroNetAmounts!=numBanks){
```

```
        int minIndex=getMinIndex(listOfNetAmounts, numBanks);
        pair<int,string> maxAns = getMaxIndex(listOfNetAmounts, numBanks,
minIndex,input,maxNumTypes);

        int maxIndex = maxAns.first;

        if(maxIndex == -1){

            (ansGraph[minIndex][0].first) +=
abs(listOfNetAmounts[minIndex].netAmount);
            (ansGraph[minIndex][0].second) = *(input[minIndex].types.begin());

            int simpleMaxIndex = getSimpleMaxIndex(listOfNetAmounts,
numBanks);
            (ansGraph[0][simpleMaxIndex].first) +=
abs(listOfNetAmounts[minIndex].netAmount);
            (ansGraph[0][simpleMaxIndex].second) =
*(input[simpleMaxIndex].types.begin());

            listOfNetAmounts[simpleMaxIndex].netAmount +=
listOfNetAmounts[minIndex].netAmount;
            listOfNetAmounts[minIndex].netAmount = 0;

            if(listOfNetAmounts[minIndex].netAmount == 0) numZeroNetAmounts++;

            if(listOfNetAmounts[simpleMaxIndex].netAmount == 0)
numZeroNetAmounts++;

        }
        else{
            int transactionAmount =
min(abs(listOfNetAmounts[minIndex].netAmount),
listOfNetAmounts[maxIndex].netAmount);

            (ansGraph[minIndex][maxIndex].first) += (transactionAmount);
            (ansGraph[minIndex][maxIndex].second) = maxAns.second;

            listOfNetAmounts[minIndex].netAmount += transactionAmount;
            listOfNetAmounts[maxIndex].netAmount -= transactionAmount;

            if(listOfNetAmounts[minIndex].netAmount == 0) numZeroNetAmounts++;

            if(listOfNetAmounts[maxIndex].netAmount == 0) numZeroNetAmounts++;
        }

    }
```

```
    printAns(ansGraph,numBanks,input);
    // cout<<"HI\n";
}

//correct
int main()
{
    cout<<"\n\t\t\t\t******** WELCOME TO CASH FLOW MINIMIZER SYSTEM
********\n\n";
    cout<<"This system minimizes the number of transactions among multiple
banks in the different\n""corners of the world that use different modes of
payment.\n""There is one world bank (with all payment modes) to act as an
intermediary between\n""banks that have no common mode of payment. \n\n";
    cout<<"Enter the number of banks participating in the transactions.\n";
    int numBanks;cin>>numBanks;

    bank input[numBanks];
    unordered_map<string,int> indexOf;//stores index of a bank

    cout<<"Enter the details of the banks and transactions as stated:\n";
    cout<<"Bank name ,number of payment modes it has and the payment
modes.\n";
    cout<<"Bank name and payment modes should not contain spaces\n";

    int maxNumTypes;
    for(int i=0; i<numBanks;i++){
        if(i==0){
            cout<<"World Bank : ";
        }
        else{
            cout<<"Bank "<<i<<" : ";
        }
        cin>>input[i].name;
        indexOf[input[i].name] = i;
        int numTypes;
        cin>>numTypes;

        if(i==0) maxNumTypes = numTypes;

        string type;
        while(numTypes--){
            cin>>type;

            input[i].types.insert(type);
        }

    }
```

```
    cout<<"Enter number of transactions.\n";
    int numTransactions;
    cin>>numTransactions;

    vector<vector<int>> graph(numBanks,vector<int>(numBanks,0));//adjacency
matrix

    cout<<"Enter the details of each transaction as stated:";
    cout<<"Debtor Bank , creditor Bank and amount\n";
    cout<<"The transactions can be in any order\n";
    for(int i=0;i<numTransactions;i++){
        cout<<(i)<<" th transaction : ";
        string s1,s2;
        int amount;
        cin >> s1>>s2>>amount;

        graph[indexOf[s1]][indexOf[s2]] = amount;
    }

    //settle
    minimizeCashFlow(numBanks,input,indexOf,numTransactions,graph,maxNumTypes)
;
    return 0;
}
```

OUTPUT SCREENSHOT



COMPLEXITY ANALYSIS

The complexity analysis of the "Cash Flow Minimizer" project involves evaluating the computational efficiency of the algorithm in terms of time and space complexities. Let's perform a complexity analysis for the algorithm described earlier:

Time Complexity:

1. **Creating the Directed Graph:** Constructing the directed graph requires examining each financial transaction. This step has a time complexity of O(E), where E is the number of financial transactions.

2. **Calculating Net Balances:** Calculating the net balances for each participant involves iterating through the graph's edges and nodes once. This step has a time complexity of O (V + E), where V is the number of participants (vertices) and E is the number of financial transactions (edges).

3. **Identifying Debtor-Creditor Pairs:** To identify debtor-creditor pairs, you need to analyse the balances of each participant. This step has a time complexity of O(V).

4. **Optimization Algorithm (Transaction Generation):**

    - The optimization loop iterates through the debtor-creditor pairs. In the worst case, each pair requires one transaction, so this loop contributes O(V) operations.

    - Inside the loop, you select the debtor-creditor pair, determine the amount to transfer, and create a transaction. These operations are constant time, O (1).

    - Updating balances and removing participants can be done in O (1) time for each transaction.

Overall, the time complexity of the optimization algorithm is O(V).

1. **Total Time Complexity:** The most time-consuming part of the algorithm is constructing the graph (O(E)), followed by calculating net balances (O (V + E)). The optimization algorithm (transaction generation) adds O(V) operations in the worst case. Therefore, the overall time complexity is O (E + V).

Space Complexity:

1. **Graph Representation:** The space required to represent the graph is determined by the number of participants (V) and financial transactions (E). The space complexity for the graph is O (V + E).

2. **Additional Data Structures:** Data structures used to store net balances, debtor-creditor pairs, and transactions require extra space, but these typically have a linear or constant space complexity, O(V).

3. **Total Space Complexity:** The overall space complexity of the algorithm is O (V + E).

In summary, the complexity analysis of the "Cash Flow Minimizer" project indicates that the algorithm is efficient for reasonably sized financial networks, with a time complexity of O (E + V) and a space complexity of O (V + E). However, the algorithm may experience performance limitations for very large and complex financial networks due to its dependence on the number of transactions and participants.

CONCLUSION

In conclusion, the "Cash Flow Minimizer" project tackles the important challenge of efficiently settling financial transactions within a group of individuals or entities. By constructing a directed graph to represent the financial relationships and employing an optimization algorithm, this project aims to minimize the number of transactions needed to simplify cash flow while ensuring fairness in the settlement process.

The algorithm presented provides an efficient solution for moderate-sized financial networks, with a time complexity of O (E + V) and a space complexity of O (V + E). It successfully simplifies complex financial relationships by generating the minimal set of transactions required to settle all debts. However, for extremely large and intricate financial networks, additional optimizations may be necessary to handle the computational demands effectively.

The "Cash Flow Minimizer" project demonstrates the practicality of using data structures and algorithms to automate and optimize financial transaction settlement, offering a valuable tool for financial management and reducing the complexity associated with debts and credits. This project not only streamlines financial operations but also provides a foundation for further research and development in financial optimization and algorithmic solutions.

FUTURE SCOPE

The "Cash Flow Minimizer" project offers a promising avenue for future development and application in various domains, particularly in the field of financial optimization. Here are some potential future scopes and directions for this project:

1. **Scalability:** Enhancing the project's scalability to handle larger and more complex financial networks. This may involve optimizing the algorithm and data structures to reduce computational requirements for very large datasets.

2. **Real-world Applications:** Integrating the project into financial software and platforms, such as personal finance management apps, corporate financial systems, or peer-to-peer lending platforms, to automate and optimize debt settlements.

3. **Blockchain and Cryptocurrencies:** Exploring how this project can be applied to blockchain technology and cryptocurrencies to optimize transactions and simplify financial interactions in decentralized systems.

4. **Machine Learning Integration:** Incorporating machine learning techniques to predict and identify future financial transactions, enabling proactive financial management and further reducing transaction complexity.

5. **Privacy and Security:** Addressing privacy and security concerns when handling sensitive financial data and ensuring that the project complies with data protection regulations.

The "Cash Flow Minimizer" project has the potential to revolutionize how we manage and optimize financial transactions, making it an exciting and dynamic area for future development and innovation. As financial networks continue to grow in complexity, the need for efficient and automated solutions will become increasingly essential, making this project highly relevant and adaptable for future use cases.

REFERENCES

1. **Books:**
    - **a.** "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
    - **b.** "Algorithm Design" by Jon Kleinberg and Éva Tardos.
    - **c.** "Data Structures and Algorithms" by Michael T. Goodrich and Roberto Tamassia.

2. **Academic Journals:**

    Look for research papers in academic journals related to financial optimization, algorithms, and graph theory. Some journals to consider include:

    - **a.** Journal of the ACM (JACM)
    - **b.** Operations Research
    - **c.** Journal of Financial Economics

3. **Online Courses and Tutorials:**

    There are numerous online platforms that offer courses and tutorials on data structures, algorithms, and financial optimization, such as Coursera, edX, and Khan Academy.

4. **GitHub and Open-Source Projects:**

    Explore open-source projects related to financial optimization and algorithms on platforms like GitHub. These projects often have documentation and code that can be helpful.