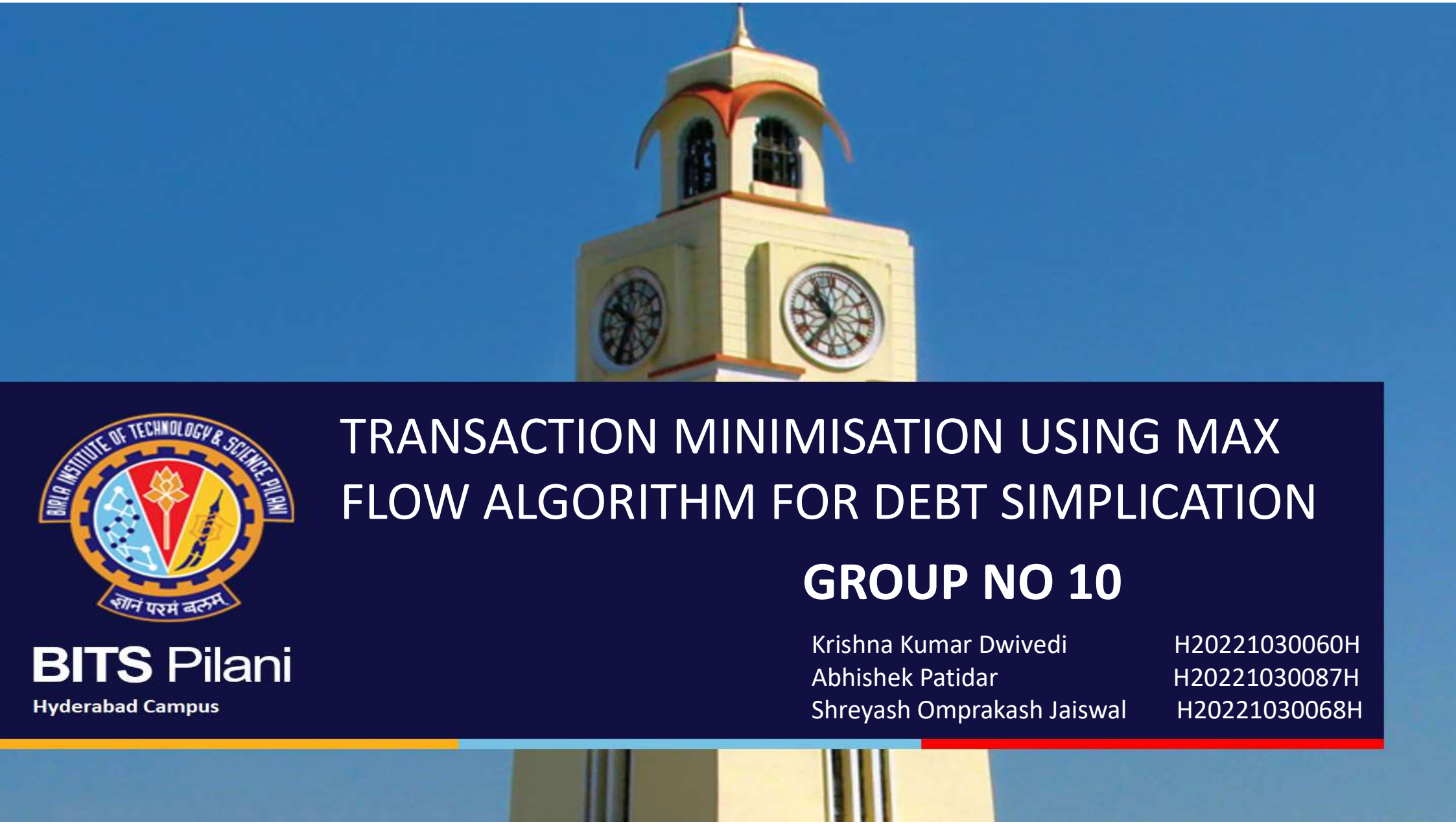


TRANSACTION MINIMISATION USING MAX FLOW ALGORITHM FOR DEBT SIMPLICATION

GROUP NO 10

BITS Pilani
Hyderabad Campus

Krishna Kumar Dwivedi	H20221030060H
Abhishek Patidar	H20221030087H
Shreyash Omprakash Jaiswal	H20221030068H

Abstract

In Today's world keeping track of all transactions and maintaining net balance is difficult in group payment transaction.

Main object of this project is to restructure debt within group such that total cashflow within group that is debt repayment is simplified while reducing number of transaction steps.

The algorithm visualizes transaction as directed edge between nodes which are visualized as member of group and it optimizes the problem using Max-flow algorithm of graph.

Max-flow algorithm involves finding a feasible flow through a flow of network which obtains maximum possible flow rate.

Project is implemented in Java /C++ programming language.

Contents

- Problem statement.
- Solving this statement using basic algebra goes to exponential time.
- How problem statement can be solved with Graph based Max-flow.
- Optimizing traditional maxflow
- Implementation progress

Problem Statement

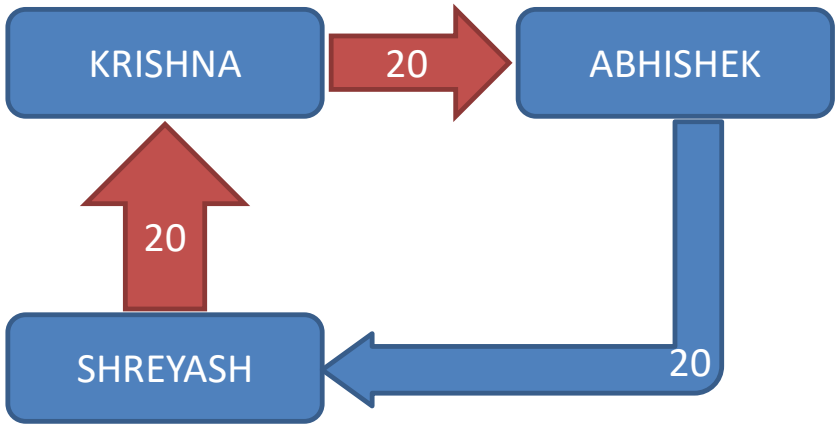
innovate achieve lead

- What is Debt simplification ?

  Restructuring debt within groups of people

  Objective : Minimizing the total number of Transaction while maintaining net balance at zero.

- Explanation through example.



Abhishek owes Krishna ₹ 20
Krishna owes Shreyash ₹ 20

Clearing this debt requires normally two transactions

1. Abhishek will pay Krishna ₹ 20
2. Krishna will pay Shreyash ₹ 20

Using Debt simplification we can reduce no of transactions to one

1. Abhishek will pay Shreyash ₹ 20

This ensures that people are paid back more quickly and efficiently

Example 2

Net Change in Cash Flow

Net-change in cash = Σ (cash_Inflow) – Σ(cash_outflow)

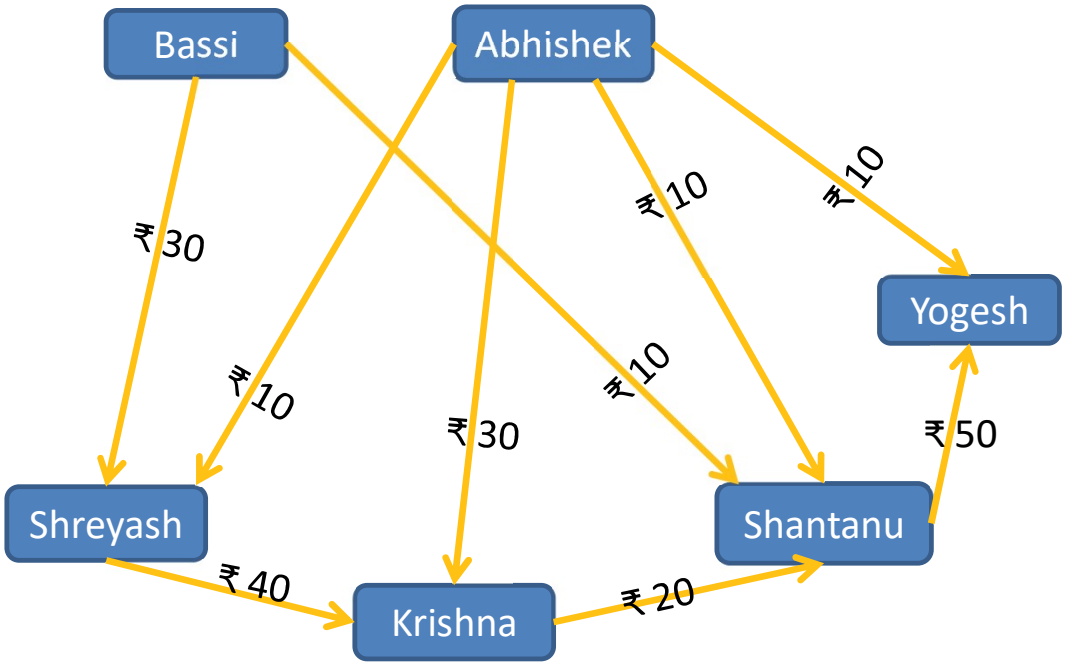
For example

Arpit

Net-change in cash(Krishna) =
Σ(cash_Inflow[Krishna]) – Σ(cash_outflow[Krishna])

= { (₹40 + ₹30) – (₹20) } = ₹50

Thus, the overall Net Change in Cash is ₹0 for Arpit, ₹0 for Shreyash, ₹50 for Krishna, -₹10 for Shantanu, ₹60 for Yogesh, -₹60 for Abhishek and -₹40 for Bassi.



Representing debt as directed graph

Sender And Receiver

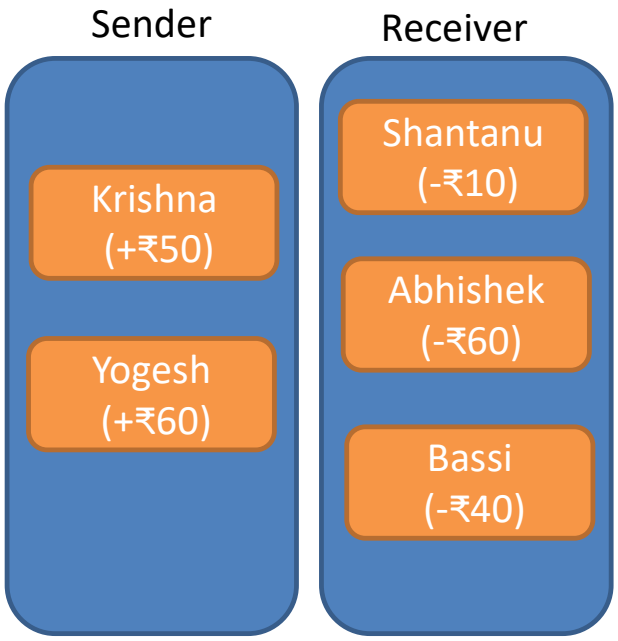
Lets categories them into two types

- Senders
  those who have extra cash, which is indicated by a positive value of ***Net Change in Cash***
- Receivers
  those who need extra cash, which is indicated by a negative value of ***Net Change in Cash***

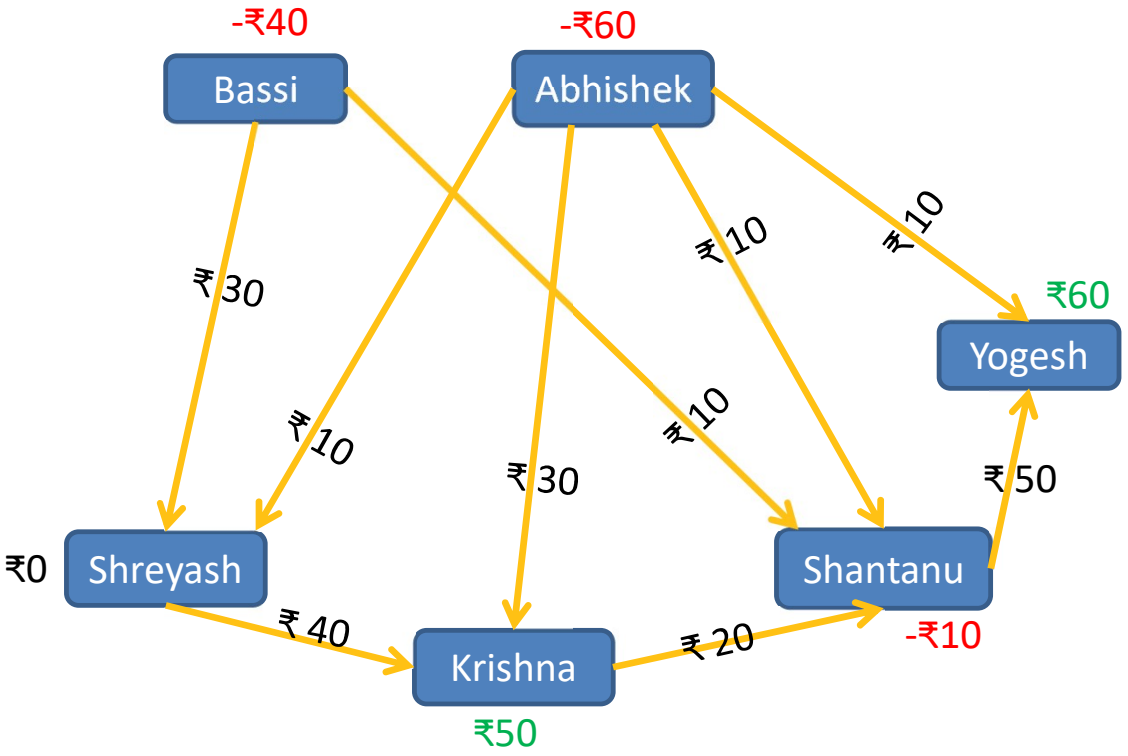
In the above example,

Krishna and Yogesh are *senders*,
Shantanu, Abhishek and Bassi are *Receivers*.
Shreyash and Arpit have their debts already sorted out and hence they are neither *Senders* nor *Receivers*.

Sender And Receiver



₹0
Arpit



Trying the solution using algebra

Since our objective is to <u>minimize total number payments.</u>

Transaction should be made between :-

1) Sender -> Receiver (Sender must only send money to *Receivers*)
2) Sender <- Receiver (*Receivers* must only receive money from Sender)
3) Obliviously Total Money owed by Sender will always equal to the total money to be received by Receiver.

To minimize the transactions we must ensure that

Sender sends money to least possible number of Receiver. Why?
Ans -> Sender in the end will make more number of transactions which will increase the total number of overall transactions made.

Cont.

For any given debts, we will check for case such that each Sender make only one transaction (to some Receiver) in order to settle debt.
(because then the total number of transactions will be equal to the number of Senders and hence will be the optimal solution)

So, in the case optimal scenario, each Sender will transfer money to exactly one Receiver.
This means that every Receiver should either receive the entire money from a Sender
or not accept any amount from them altogether.

For example :
Amount to be paid = G1 , G2 , G3 , G4
Amount to be received = R1 , R2
Any given Receiver, he/she can either accept the entire amount G1 from the first Sender or not accept any amount altogether. And so on…
This means that we are looking for a *Subset* of Sender that exactly adds up to a given *Receiver's* amount and we need to do this for all *Receivers*.

Cont.

Until now we took a case for optimal case

There might be another case such that the best possible solution requires to make subset of Sender in order to make more than one transaction.
Here to check all possible ways of splitting amount from Sender.

This indicates that the debt simplification problem is at least as hard as the Sum of Subsets Problem and hence it is NP-Complete.

It will require an exponential number of steps to minimize the total number of payments.

How problem statement can be solved with Graph based Max-flow?

In order to reduce the time complexity for debt simplification algorithm from exponential to polynomial time 3 golden rules need to be followed :-

1. Obliviously, Everyone owes the same net amount at the end.
2. No one should owe a person that they didn't owe before.
3. No one owe more money in total than they did before the simplification. (simple math)

Thus following these rules the problem can be simplified to varying the amount being transferred on existing transactions without introducing newer ones.

Mathematically
Given a Directed Graph representing Debts, change (if needed) the weights on the existing edges without introducing newer ones.

Cont.

Given a Directed Graph representing Debts, change (if needed) the weights on the existing edges without introducing newer ones.

This can be solved if we divide the problem into two parts :-

1. Will an existing edge be part of the graph after simplifying
2. If it is present in the graph after simplifying then what will be the weight (i.e. amount) of it?
   maximize the debt(i.e. weight) on the edge, so as to get rid of debts flowing along other paths between the same pair of vertices

Algorithm

1.Create a Directed Graph(G) with debts as weight.

2.Select one of the non-visited edges, lets say (u, v) from the directed graph G.

3.u as source and v as sink, run a maxflow algorithm to determine the maximum flow of money possible from u to v.

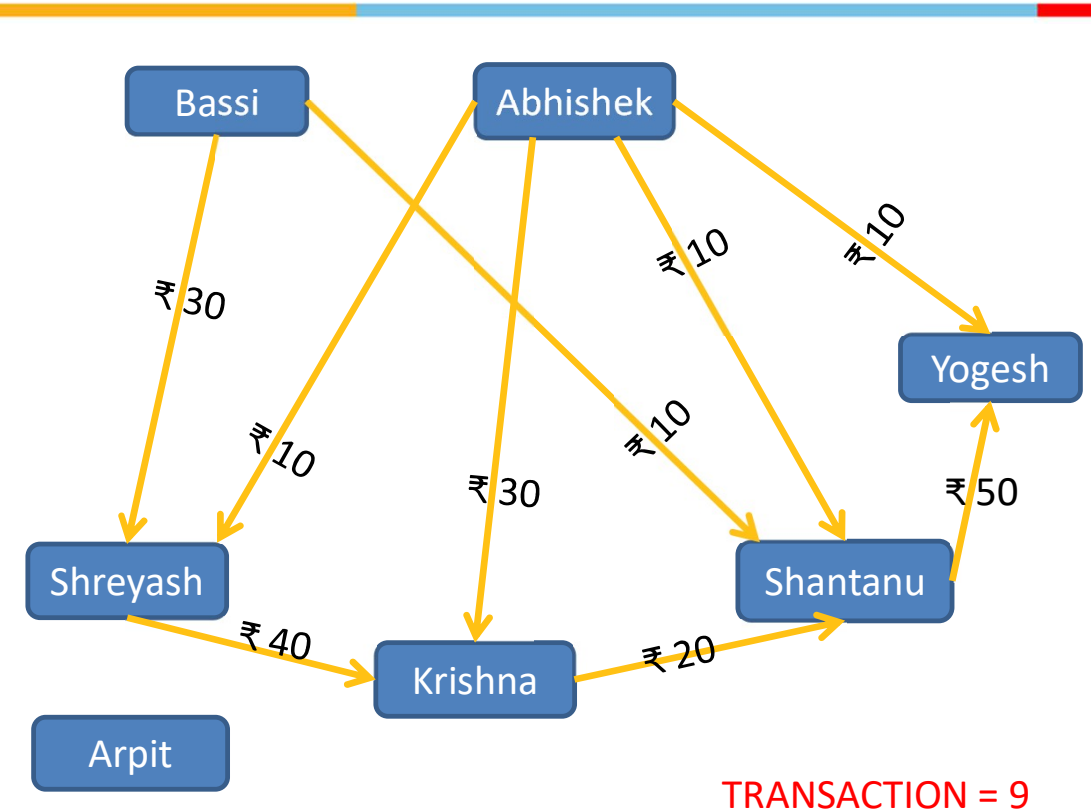
4.Compute the Residual Graph (G') which indicates the additional possible flow of debts in the input graph after removing the flow of debts between u and v.

5.If maximum flow between u and v is greater than zero, then add an edge(u, v) with weight as max-flow to the Residual Graph.

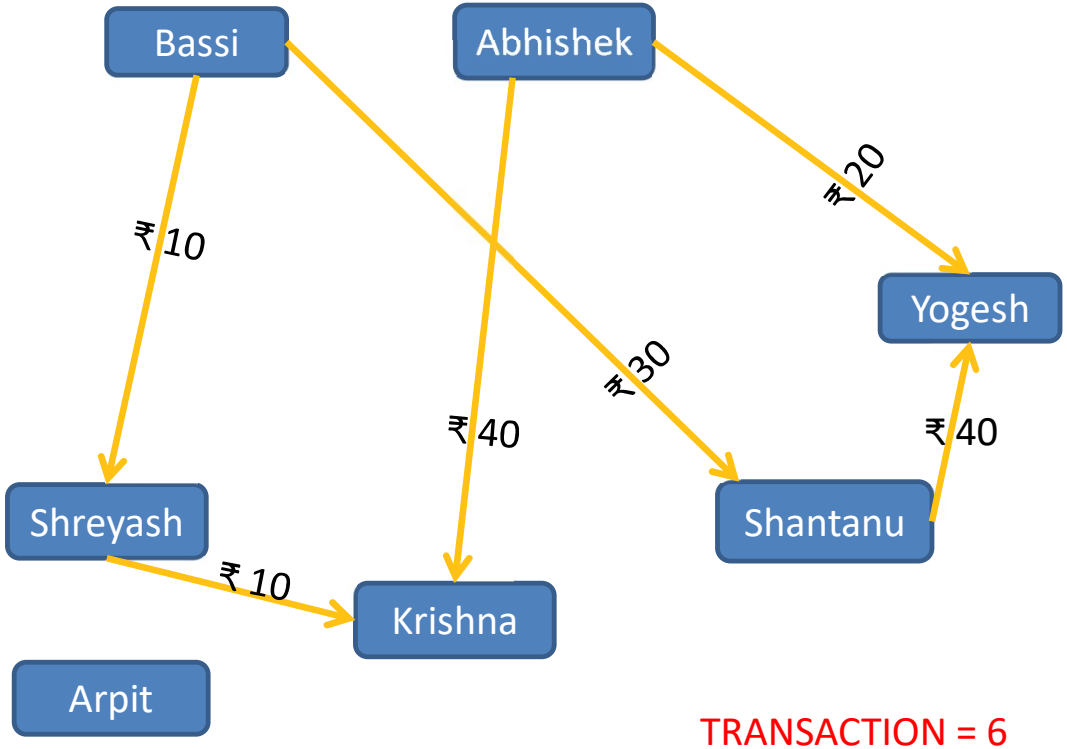
6.Now go back to Step 1 and take Residual Graph G' as Input. (Loop until all the edges are covered)

Output achieved :- Once all the edges are visited, the Residual Graph G' obtained in the final iteration will be the one that has the minimum number of edges(i.e. transactions).

Cont.



TRANSACTION = 9



TRANSACTION = 6

Time complexity Analysis

Time complexity of $O(V^2E^2)$, where V = #Vertices & E = #Edges.

$O(V^2E)$ is the complexity of the Dinics implementation of the Maximum flow Problem and the extra O(E) is because of the algorithm iterates over all the edges in the graph.

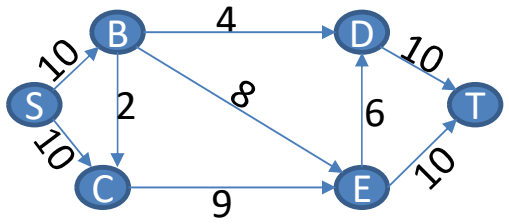
Time complexity of $O(V^2E^2) = O(E) * O(V^2E)$.

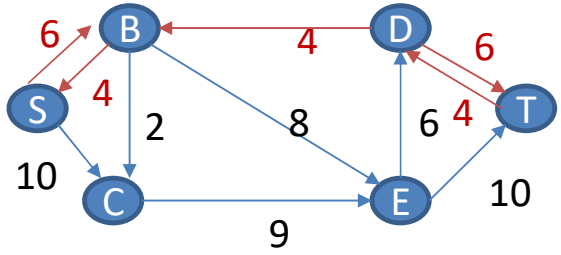
So therefore the overall complexity depends on the maxflow algorithm used hence our paper will try to optimize general max-flow algorithms

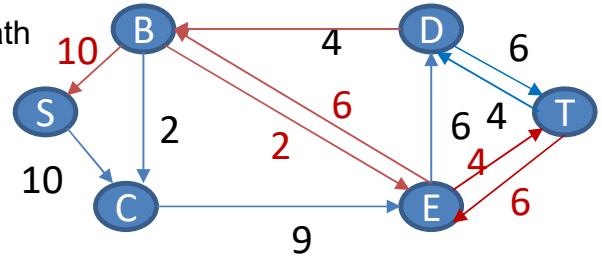
DINICS MAX-FLOW ALGORITHM

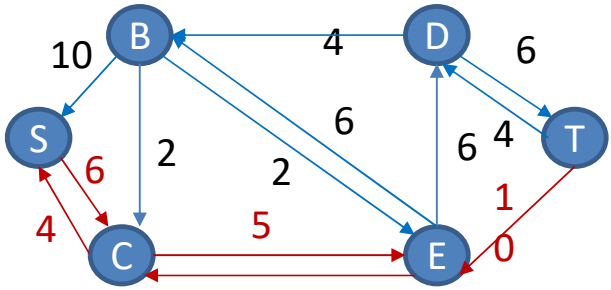
innovate achieve lead

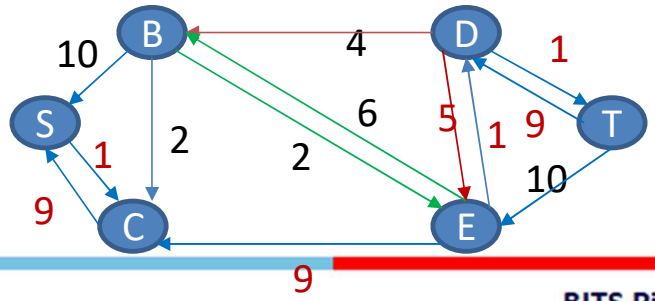
Max_flow =
For each edge (u,v) in G:
flow(u,v) = w(u,v)
While there is augmenting level path p from S -> T in RG:
assign level to vertices
while more flow is flow is possible from S -> T :
min_cap = minimum residual capacity in path
max_flow = max_flow + min_cap
for each edge (u,v) in path:
flow(u,v) = flow(u,v) – min_cap
flow(v,u) = flow(v,u) + min_cap
Return max_flow











DEFINITION

innovate achieve lead

Let G V Ac = (V,A,C) be a capacity network, with Vs , Vt being the source and the sink respectively .
V is the set of vertices;
A is the set of arcs,
c-- a vector Cij : , (i,j ∈ A) of capacities for flows on the arcs.
Each arc in the network is an ordered pair(Vi,Vj) , denote by a = (Vi,Vj).
i, j are called the tail and head of arc a.
a is called the out-arc of i v and the in-arc of j v .

(2)Degree of vertex:

In-degree $d_D^-(v)$ = the number of in-arc to v,

Out-degree $d_D^+(v)$ = the number of out-arc to v.

(3) u is a chain in a network, the positive direction is $v_s \to v_t$,then:

forward arc :The direction of the arc is the same to the positive direction,

reverse arc :The direction of the arc is the opposite to the positive direction.

Step of improved algorithm

innovate achieve lead

1. $f(v_i, v_j) \leftarrow 0$, for all edges (v_i, v_j),

2. Divide the given network G into several areas, calculating the capacity of the forward arc in every area, mark in the bottom of the lines. Arrange all forward arcs in the minimum capacity area, according to the in-degree of ai's tail from small to large denote by A= {a_1, a_2 ...a_n}. (If the in-degree of some vertex is same, the bigger capacity arc is before the others).

3. While there is a path u from v_s to v_t:

3.1 *Path Search*

In turn find all augmenting paths from v_s to v_t which contain a_i (i=1, 2...n),

Choose augmenting path u with fewest edges and contain a_i, and

If there are two: (1).when the vertex is in the left of ai, preferred choice the vertex with smaller out-degree.

(2).when the vertex is in the right of ai, preferred choice the vertex with smaller in-degree.

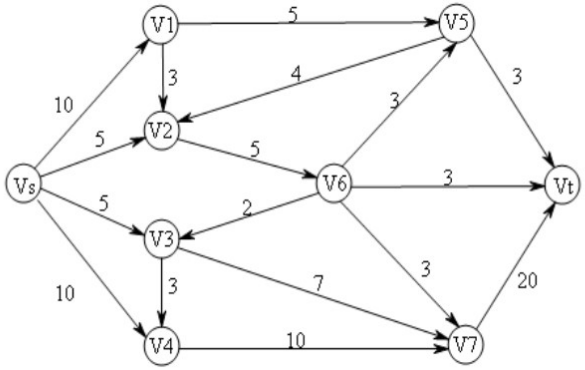
3.2 *Capacity Update:*

For each edge $(v_i, v_j) \in u$,

$$f(v_i, v_j) \leftarrow f(v_i, v_j) + \delta, \quad \delta = \min_{(v_i, v_j) \in u} (c_{ij} - f_{ij}).$$

Ensure that every time augment at least make a arc saturated, and draw "||" in saturated arc.

Let D= (V, A, c), The capacity is denoted in Figure 1.



Cont.

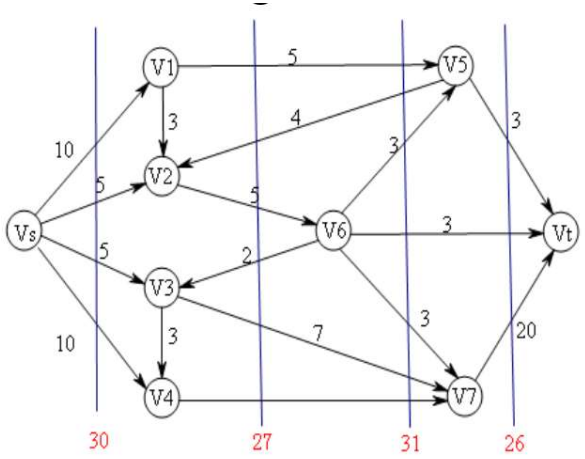


Figure 2

1. Shown in Figure 2. Divide area, and minC=26, $A=\{a_1,a_2,a_3\}=\{v_6v_t,v_5v_t,v_7v_t\}$.

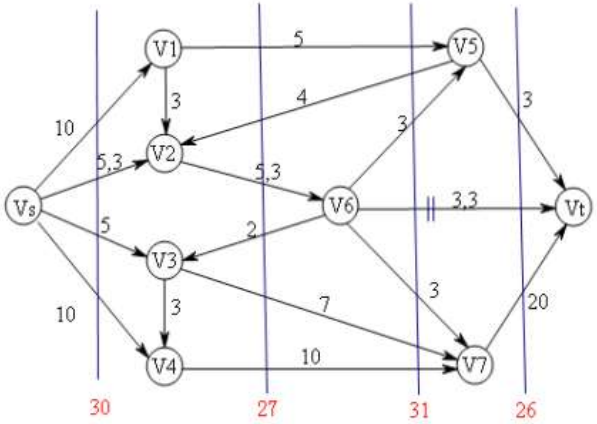
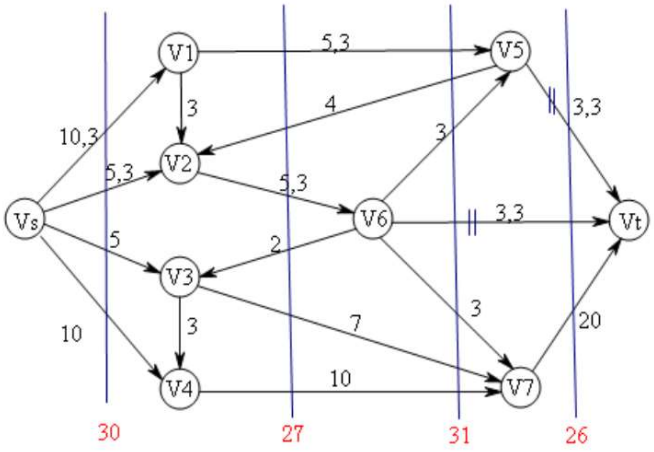


Figure 3

2. Find all augmenting paths from v_s to v_t which contain a_1, get u1: $v_s-v_2-v_6-v_t$, $\delta_1=3$, shown in Figure 3.

Cont.



3. Find all augmenting paths from v_s to v_t which contain a_2, get u_2: $v_s - v_1 - v_5 - v_t$, $\delta_2 = 3$,shown in Figure 4.

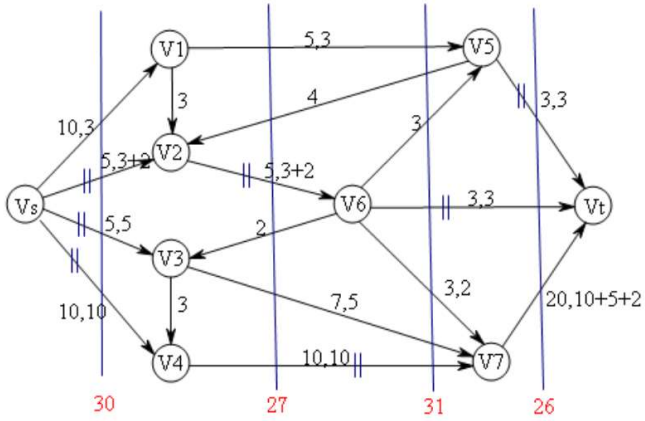


Figure 5

4. By the same reason, get u_3: $v_s - v_4 - v_7 - v_t$, $\delta_3 = 10$;u_4: $v_s - v_3 - v_7 - v_t$, $\delta_4 = 5$ u_5: $v_s - v_2 - v_6 - v_7 - v_t$, $\delta_5 = 2$, shown in Figure 5.

As you can see, in Figure 5 can't find the augmented chain from v_s to v_t, so termination algorithm, get the maximum flow f_{max} =3+3+17=23.

CONCLUSION

Note: Using this algorithm for solving the maximum flow only need to draw a diagram to be completed the entire operation process. Such as in Figure 5 can be marked with the whole operation process.

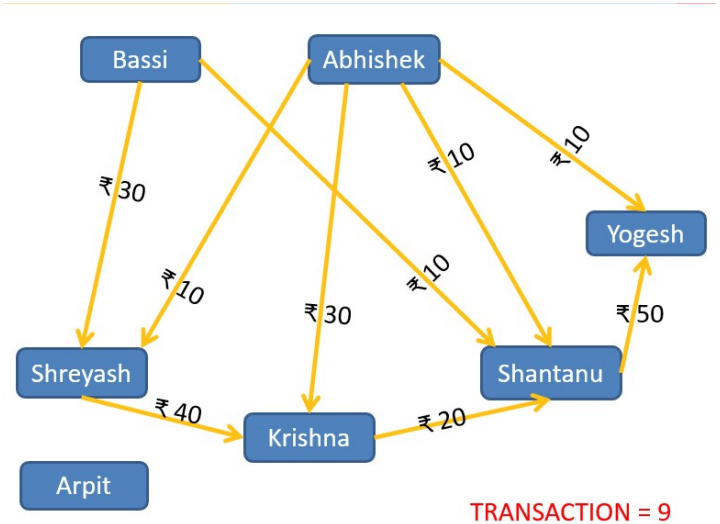
This algorithm only need to find five augmented chains to get the maximum flow, but Ford-Fulkerson algorithm needs find twelve augmented chains to complete augmented process to get the maximum flow is 23.

Implementation

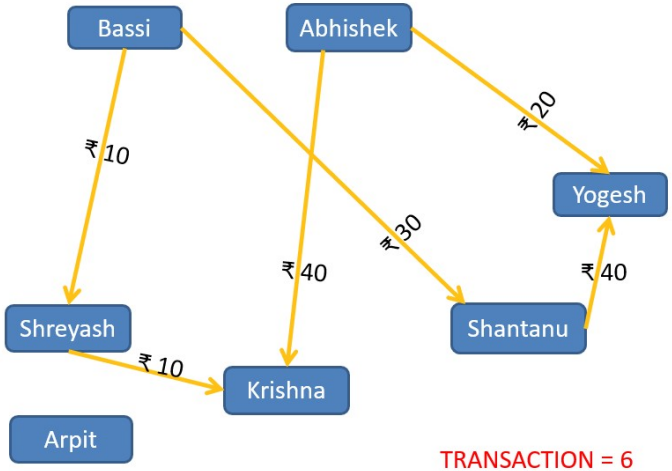
```
Transaction Graph
  Shreyash ----    40 ---->    Krishna
   Krishna ----    20 ---->   Shantanu
  Shantanu ----    50 ---->     Yogesh
  Abhishek ----    10 ---->   Shreyash
  Abhishek ----    30 ---->    Krishna
  Abhishek ----    10 ---->   Shantanu
  Abhishek ----    10 ---->     Yogesh
     Bassi ----    30 ---->   Shreyash
     Bassi ----    10 ---->   Shantanu

AFTER TRANSACTION MINIMISATION
_______________________________

  Shantanu ----    40 ---->     Yogesh
  Abhishek ----    20 ---->     Yogesh
  Abhishek ----    40 ---->    Krishna
     Bassi ----    30 ---->   Shantanu
     Bassi ----    10 ---->   Shreyash
  Shreyash ----    10 ---->    Krishna
```



TRANSACTION = 9



TRANSACTION = 6