Maximum Flow-based approaches to Debt Simplification

Luca Griso

April 2025

Indice

1 Invitation to the problem

It is common, especially among friends, for people to lend and borrow money from each other - for example, when splitting dinner bills or covering shared expenses among roommates. Rather than settling debts immediately, it is often more convenient to record them and address everything at a later time. Over time, many debts may accumulate, but settling each one individually could lead to many unnecessary transactions. Moreover, when debts are crisscrossed among group members, it can become complicated to figure out who owes what to whom.

The goal of this project is to develop an efficient algorithm that minimizes the number of transactions needed to settle all debts. I was inspired by **Splitwise's simplify debts** feature, which, like similar apps, does exactly this: it does not alter the total amount anyone owes but makes it easier to pay people back by minimizing the total number of payments. Another possible goal could be minimizing the total amount of money transferred, but I will only touch on this aspect marginally.

To be a bit more precise, denote the N friends with numbers from 0 to $N-1$, and suppose that $a_{i,j}$ represents how much friend i owes to friend j $(i \neq j)$ at the end of the considered period of time.

Despite this everyday problem looks pretty innocent, devising a suitable algorithm for an arbitrary number of people and debts entails a certain degree of effort, as we are going to see. Let me first consider some simple cases from [6]:

- $\mathbf{N = 2}$: in this case the optimal solution is intuitive:

$$\begin{cases} \text{Friend 0 pays } a_{0,1} - a_{1,0} \text{ to friend 1,} & \text{if } a_{0,1} > a_{1,0}, \\ \text{No transfers are needed,} & \text{if } a_{0,1} = a_{1,0}, \\ \text{Friend 1 pays } a_{1,0} - a_{0,1} \text{ to friend 0,} & \text{if } a_{0,1} < a_{1,0}. \end{cases}$$

- $\mathbf{N = 3}$: first let's represent the situation using this debt matrix:

$$\begin{pmatrix} \cdot & a_{0,1} & a_{0,2} \\ a_{1,0} & \cdot & a_{1,2} \\ a_{2,0} & a_{2,1} & \cdot \end{pmatrix}$$

  These debts can obviously be settled by the trivial solution requiring $N(N-1)$ transfers (each person pays back everyone else). Noticing how the elements $a_{i,j}, a_{j,i}$ refer to one couple of people, we can exploit again the solution for $N = 2$. These two mutual debts can then be solved by a single transfer involving the amount $|a_{i,j} - a_{j,i}|$, so the whole case requires 3 transfers. More generally this situation is worked out by $\frac{1}{2}N(N-1)$ transactions, a better result, unfortunately not optimal in general.

Only simple/particular cases present a straightforward solution. Indeed the $\mathbf{N > 3}$ cases are already more complicated (see [6]) and a more systematic approach quickly becomes necessary. Fortunately debt simplification issues lend themselves well to being solved by various algorithms, as illustrated by the reference literature.

1.1 Transferable debt approaches

Let me start by anticipating a few concepts from the mathematical model we are going to better discuss later.

- Notice how our goal is devising an algorithm able to restructure the debts minimizing the total number of payments to be made, but maintaining the correct debt relationship between people. Let me define a set of transactions as **correct** if it does so: everybody must pay or receive as much money as they owed or were owed initially.
- One may realize how this problem is not so much about the exact set of debts, as about how much a person owes/is owed overall. As a consequence, for each person in the group, let me define the net amount, a quantity we are going to refer to in the following, as

  $$\textbf{net amount in cash} = \text{sum of inflow cash } - \text{sum of outflow cash}$$

  Notice how a positive balance indicates that a person is supposed to receive more than they have to pay back, and vice versa.

An important assumption that divides the solutions within the literature is if the debt is **transferable** or not. It is so if, say A owes \$5 to B and B owes \$5 to C, and we can convert this into a single \$5 debt from A to C. Let's have a look at a few solutions within those exploiting this fact:

- **Central collector** by [6]

  Since there is no debt going in or out of the group as a whole, the sum of all balances must be 0. Denote one of the members as the *collector*. Let everybody with a negative net amount pay the collector for their total debt and the collector will pay back everybody with a positive balance for their total credit amount. Clearly this settles correctly everyone but the collector itself. From above we know that the collector's balance must be the opposite of the sum of everybody else's balance. Denoting each person's balance as b_p :

  $$b_{\text{collector}} = - \sum_{\substack{p \in \text{group} \\ p \neq \text{collector}}} b_p$$

  But the rhs is exactly the net amount of money that the collector is receiving, proving that this set of transactions is correct.

  We might hope for a stronger reduction in the number of transfers, and having the collector involved in so many transactions may not be ideal in a concrete application (imagine a fee for each transfer to be made).

- **Greedy algorithm** from [3]

  This solution proceeds by steps: at each, settle out of the debt balances of one person and recur for the remaining $N-1$ people. Defining the MaxCreditor and the MaxDebitor (relative to the given step) as the person with the highest and smallest net amount, the idea is that at each iteration the evened out-person is either the MaxCreditor or the MaxDebitor.

  For each step:
  define the amount to be credited MaxCreditor be maxC and the one to be debited from MaxDebitor be maxD. Let x be the minimum of the two amounts. Debit x from MaxDebitor and credit this amount to MaxCreditor. Then

$$\begin{cases} \text{Remove MaxCreditor from the set of people,} & \text{if } x = maxC, \\ \text{Remove MaxDebitor from the set of people,} & \text{if } x = maxD, \\ \text{Remove both of them from the set of people,} & \text{if } maxC = maxD, \end{cases}$$

  And recur over the remaining people.

1.2 Non Transferable debt approaches

It can be shown how finding the optimal solution for the debt simplification task is actually a NP-Complete problem (refer to [1]) : this means that it will be required an exponential number of steps to minimize the total number of payments. Actually Splitwise does not run an exponential algorithm to solve this problem in real time. Indeed, [5] mentions 3 rules the feature obeys:

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

The second is just the correctness of the restructuring, while the last one suggests narrowing down to non-transferable approaches, as it is probably by doing so that they are able to obtain a more feasible (and performing) solution.

2 Mathematical model

A particularly suited way of modelling relationships that exist between pair of objects is a **labelled directed graph**, meaning a pair (V, E) where V is a set of vertices and E is a set of edges. Representing each person as a vertex, any given sum of money, say owed by David to Ema, is then a directed edge $(David, Ema)$ between the two. This edge is labelled with a weight representing the debt, and a number representing the flow. Moreover, encoding the N people as numbers 0 to $N-1$ (as done in the code), each vertex i is labelled with the net amount b_i.

We are now able to visualize this set of 9 transactions between 6 people:

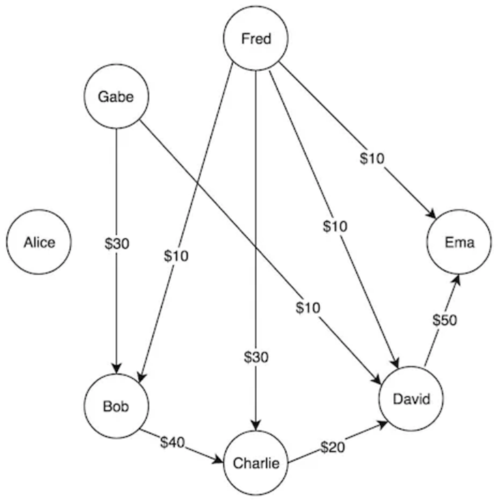


Figura 1: Representing Debts in the form of a Directed Graph

Here is the code for the two fundamental classes, defined by the attributes cited above.

```python
import networkx as nx

class Edge:
  def __init__(self, v, rev, cap, type):
      self.v = v
      self.rev = rev    #see usage in sendFlow function below
      self.cap = cap
      self.flow = 0
      self.type = type    #distinguish forward from backward edges


class Graph:
  def __init__(self, V):
      self.V = V
      self.adj = [[] for _ in range(V)]   #adjacency list

```

```python
        self.edges = [] #vertex pairs (u,v) representing the edges
        self.edge_index = {}  #dictionary to access (u, v) -> index

        self.posit = list(range(V))
        #let's now map each vertex into its index
        self.posit_map = {v: i for i, v in enumerate(self.posit)}

        #self.level[i] stores the level index of vertex i
        self.level = [-1] * V
        self.modif = [] #tracks the modifications applied to edges
        self.nonzeroedges = [] #ones with a non-zero capacity

    def addEdge(self, u, v, cap):

        #explanation of the following value for edge.rev: see sendFlow function
        #Forward edge : 0 flow and cap capacity
        a = Edge(v, len(self.adj[v]), cap, 1)
        #Backward edge : 0 flow and 0 capacity
        b = Edge(u, len(self.adj[u]), 0, 0)

        self.adj[u].append(a)
        self.adj[v].append(b)
        self.edges.append((u, v))
        self.edge_index[(u, v)] = len(self.edges) - 1 # store the index of the
    ↪ edge (u,v) in the edges list
        self.modif.append(0)

        #use self.dicts[u] to store, for the key v, the position
        #where v has been saved in the self.adj[u] list
        self.dicts[u][v] = len(self.adj[u]) - 1
        self.dicts[v][u] = len(self.adj[v]) - 1

    def draw(self):
        g = nx.DiGraph()
        g.add_edges_from(set(self.edges))
        nx.draw(g, with_labels = True, node_size=600, node_color='#00b4d9')

    #we are interested in visualizing only non zero edges in order
      #to get a visual idea of the algorithm's effectiveness
    def nonzeroedgDraw(self):
        g = nx.DiGraph()
        g.add_edges_from(set(self.nonzeroedges))
        nx.draw(g, with_labels = True, node_size=600, node_color='#00b4d9')
```

Listing 1: Edge and Graph classes definition

The task can thus be rephrased as the following graph problem: we have to find the set with fewest edges, labelled by positive numbers, such that all balances are evened out. Actually, due to the constraint of the above subsection, this boils down to changing (if needed) the weights on the existing edges without introducing newer ones.

In section 3 we are going to see how this can be achieved by addressing separately the two questions:

1. Will an existing edge be part of the graph after simplifying debts?
2. If so, what will be its weight?

In order to measure the algorithm performance, the literature typically refers only to the maximum number of transactions that a methodology needs to solve the task in the worst-case scenario. I would like to provide a quantitative index based on a result by [6], considering that such a measure seems to lack in the literature.

Proposition 1 (Lower bound on the number of edges) *Denoting the net amount of person i as b_i, a lower bound on the number of edges is:*

$$|E|_{min} = \max\left(\#i : b_i > 0,\ \#j : b_j < 0\right)$$

This is due to the following: for sure each vertex with a positive balance must have at least one incoming edge, and each negative, one outgoing one. Thus the minimum number of edges is the maximum of the two cardinalities of such sets. A reasonable **performance index** in $[0, 1]$ is then:

$$\frac{|E|_{in} - |E|_{fin}}{|E|_{in} - |E|_{min}}$$

where $|E|_{in}, |E|_{fin}$ denote the number of edges before and after the simplification.
It measures how much the obtained result differs from the optimal one, meaning the one reaching the lower bound.

3 Algorithm presentation and "Toy" example

3.1 The maximum flow problem and Dinic's algorithm

Let's start by answering the second of the two questions we divided the problem into.

The weight of an edge, after simplifying debts, will be the **maximized debt, obtained by getting rid of debts flowing along other paths between the same pair of vertices**. By doing so, we are trying to "summarize" the overall debt relationships into the single edge.

For example, consider Figura 1 again. In particular, focus on the edge $(Fred, Ema)$.

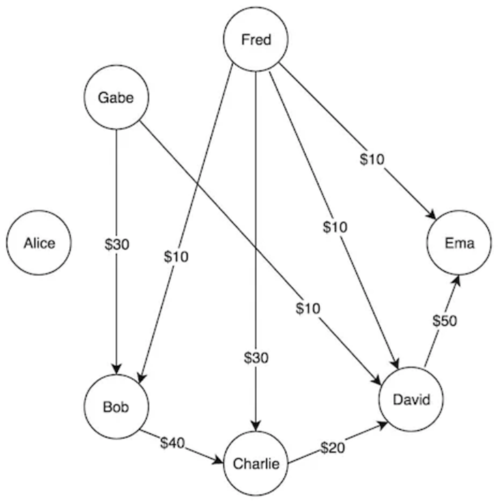


Figura 2: $(Fred, Ema)$ debt maximization

Here, if Fred transfers $20 to Ema, then Fred will not have to pay David anything and David will only have to pay $40 to Ema, thereby reducing the total transactions to be made from 3 to 2.

The edge's debt maximization problem refers to what is known in the literature as the **maximum flow problem**, that involves finding the maximum amount of flow that can be sent, from a **source s** to a **sink t**, through a network of pipes subject to capacity constraints. The capacity represents the limiting amount of flow that can be sent through a pipe. This is a very flexible tool that can model different real - world situations such as transportation systems or internet data routing (how to best route data packages through networks). But as we are going to see it is also surprisingly appliable to not so intuitive cases, like the debt simplification one.

Let's dive in the reference literature: [2]

1. One of the first algorithms designed to solve this problem has been the **Ford-Fulkerson algorithm**, introduced in 1956 by Ford and Fulkerson. It introduced the foundational concepts for many similar algorithms later developed, among which Dinic, the one I am going to implement.
   It repeatedly finds augmenting paths from the source to the sink - that is, paths along which additional flow can be pushed - and continues doing so until no such paths remain. It is efficient for small capacities, and at most for medium sized graphs.

2. A suited solution for more dense graphs of moderate size is **Edmonds-Karp algorithm**, proposed in 1972. While the previous allowed for arbitrary selection of augmenting paths, this algorithms uses BFS to find the shortest augmenting paths in terms of number of edges. This guarantees to terminate in **polynomial time**, even with real valued or large capacities.

3. **Dinic's algorithm**: this is a really efficient and revolutionary solution, introduced by Dinitz in 1970, that pushed the field forward. Besides combining together different graph traversals techniques, it has introduced many new concepts like the level graph and the blocking flow. Moreover it particularly boosts performance on bipartite graphs, reducing further the time complexity and making it feasible to deal with graphs of massive scale.

But how is the maximum flow problem related to our debt simplification task?
Let's try to understand it first for just **one edge**, the $(Fred, Ema)$ of above.
Illustrations in the following two figures.

[Left] The debt associated with each edge plays the role of capacity in the context of maximum flow. The flow attribute of all edges is initially set to zero, and what actually represents the flow an edge is still able to receive is: *capacity - flow.*

[Right] Notice how any amount of flow that is sent through a path from s to t has to respect the capacity constraint of each of the edges composing the path. Thus the maximum amount of flow that can be pushed through the path is the minimum of that path edges' capacities (*bottleneck* value from here).
Whenever such amount, say *f*, of flow is pushed from s to t, the flow of each edge along the path is increased by *f*, thus reducing the remaining flow.

[Below] After the algorithm for the maximized edge terminates, the capacity of each "modified" edge is decreased by an amount equal to the final flow of that edge. Here, by "modified" edge I mean each edge —other than the one being maximized— upon which some flow has been sent. The maximized edge's capacity is increased by *f*.
Notice how, by moving this f-flow from the path edges to the maximized one, we are actually **shifting the debt towards the primary edge**. Indeed, in

each augmented path there is now at least one edge that has zero capacity, so **one transaction less**, while the others have a decreased debt.

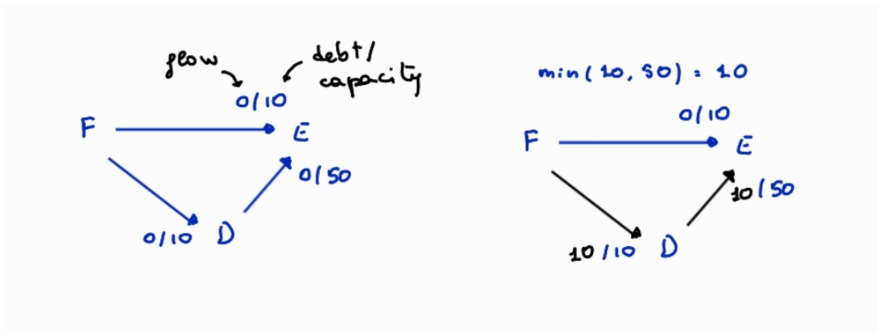


Figura 3: $(Fred, Ema)$ maximization: connection to pushed flow

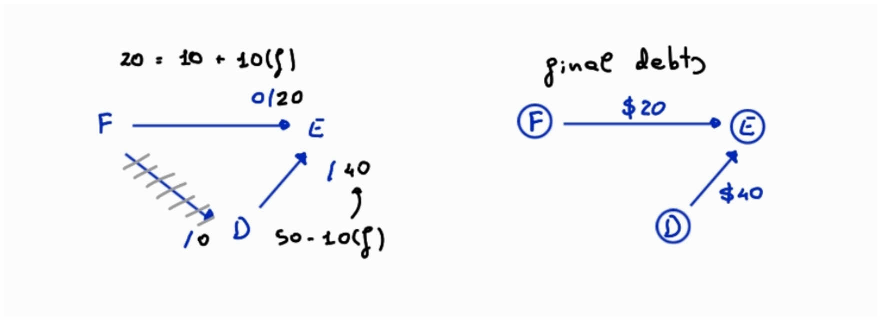


Figura 4: $(Fred, Ema)$ maximization: connection to pushed flow

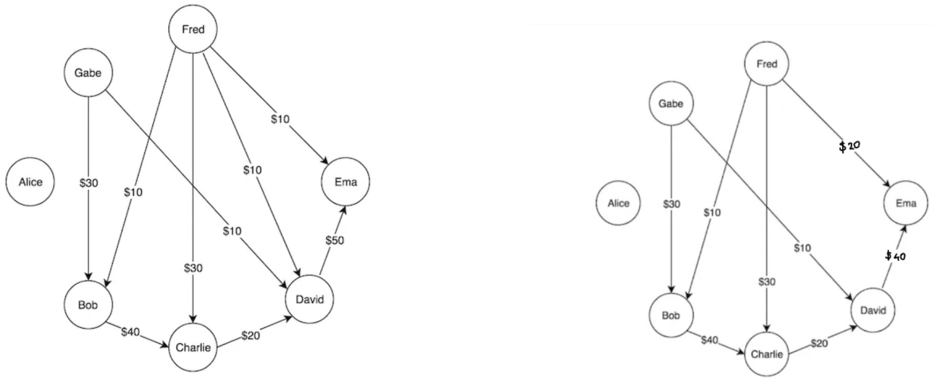


Figura 5: Results. $(Fred, Ema)$ maximized: \$20, $(David, Ema)$: \$40

We will see how Dinic's algorithm (the sendFlow function in particular) works in a similar way, just on the whole graph.

We aim at pushing as much flow as possible through the network, hoping to saturate many edges and thus reduce their total number. Observe how we are modifying the debt relationships only on existing edges, as we aimed to do.

Algorithm 1 Dinic's Algorithm for Maximum Flow **on a single edge**

```
Require: A flow network with source $s$ and sink $t$
Ensure: The maximum flow from $s$ to $t$
  repeat
      Step 1) Construct a level graph by performing a BFS from $s$ to label all the
              levels of the current flow graph
      if sink $t$ is not reached during BFS then
          Stop and return the current flow as the maximum flow
      end if
      Step 2) Using only valid edges in the level graph, do multiple DFSs from
              $s$ to $t$ until a blocking flow is reached. For each augmenting path
              on which some flow has been sent, add this flow to the current flow.
  until Step 1 condition applies
```

A few comments (for a better understanding see the "toy" example):

1. Regarding the level graph, refer to the explanation below. In this context, **valid edges** refer to those that actually lead us nearer to the sink.
2. The term **blocking flow** refers to the situation when we cannot find any more paths from the source to the sink because too many edges in the level graph have been saturated.

Being the above pseudocode quite high level, in order to present Dinic's algorithm I am going to refer to the following code (all additional methods of the already presented Graph class). Let's start by the BFS one (refer to the Notebook for the queue implementation):

```python
#BFS defines the level graph and finds if more flow can be sent from s to t
def BFS(self, s, t):
    self.level = [-1] * self.V
    self.level[self.posit_map[s]] = 0

    #using a queue implemented as a circular array
    q = CircularQueue(self.V)
    q.enqueue(s)
    while not q.is_empty():
      u = q.dequeue()
      for e in self.adj[u]:
          #only directed edges from u
          if e.type == 1 and self.level[self.posit_map[e.v]] < 0 and e.flow <
  ↪ e.cap:
```

```
14                  #Level of current vertex is level of parent + 1
15                  self.level[self.posit_map[e.v]] = self.level[self.posit_map[u]]
    ↪ + 1
16                  q.enqueue(e.v)
17
18          #If the sink can not be reached, we signal it returning False
19          return self.level[self.posit_map[t]] >= 0
```

Listing 2: BFS method implementation

The algorithm that pushes flow towards the sink must rely on some form of guidance to determine which edges, if selected next, actually lead closer to the sink ([7]). The way Dinic's determines which make progress towards t and which do not is by building a **level graph**. The indices of the level graph are the output of this method, that is nothing but a modified BFS from the source (notice line 13, since the graph is directed we only move to outgoing edges, thanks to the attribute type).

Once the vertices indices are defined, the level graph will be only composed of edges that go from a node at level L to another at level L+1. Notice how this requirement **prunes edges** that go either backwards or ”sideways”, precisely those that do not contribute to progressing toward the sink. The following illustrates this concept to the initial situation of Figura 1. This will serve as a ”toy” example, which, for better clarity, will be illustrated alongside the code.

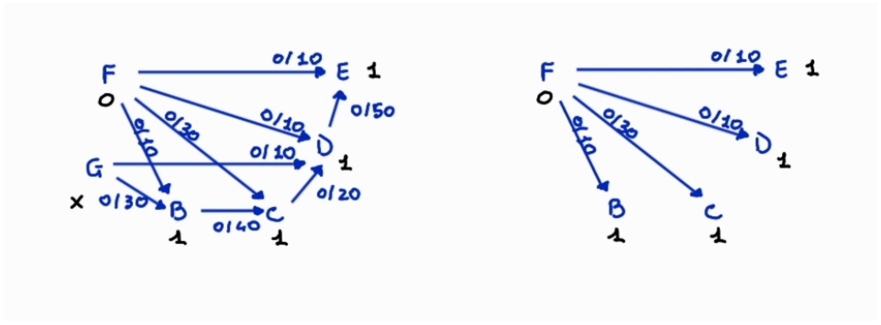


Figura 6: BFS method and level graph construction

Let's proceed with the core of the algorithm (all additional methods of the already presented Graph class). Notice that until the end of the subsection we are dealing with the maximization of a **single edge**.

```
def DinicMaxflow(self, s, t):
    if s == t:
        return 0
    total = 0

    # Augument the flow while there is path from source to sink
    while self.BFS(s, t):
        start = [0] * self.V
        while True:
```

```python
                flow = self.sendFlow(s, float('inf'), t, start)
                #float value : see curr_flow in sendFlow funct
                if flow <= 0:
                    break
                total += flow #Add path flow to overall max flow

        return total  #return maximum flow
```

Listing 3: DinicMaxflow method implementation

Until BFS indicates there is still additional flow possible, the **DinicMaxFlow function** calls the sendFlow function. Instead when BFS returns that the level graph is no longer able to reach the sink, this method returns the maximum flow as sum of all the bottleneck flow values sent before.

```python
  def sendFlow(self, u, flow, t, start):
  #u: current vertex, flow: current flow sent by parent function call
      if u == t:
          return flow

      # Traverse all outgoing edges from u, one -by -one
      while start[self.posit_map[u]] < len(self.adj[u]):
      #start[i] stores how many of the edges (from the vertex i)
      #have already been visited

          e = self.adj[u][start[self.posit_map[u]]]
          if self.level[self.posit_map[e.v]] == self.level[self.posit_map[u]] +
   ↪ 1 and e.flow < e.cap:

              # find minimum flow from u to t
              curr_flow = min(flow, e.cap - e.flow)
              temp_flow = self.sendFlow(e.v, curr_flow, t, start)

              if temp_flow and temp_flow > 0: #if flow is possible

                  #add flow to current edge
                  e.flow += temp_flow

                  #subtract flow from reverse edge
                  self.adj[e.v][e.rev].flow -= temp_flow
                  #In addEdge, I stored in the forward edge what was the length
                  #the list of the endpoint vertex when this edge has been
                  #created, so to being able to retrieve it now

                  #signal that the edge's flow has changed, useful in the last
   ↪ while loop
                  idx = self.edge_index.get((u, e.v))
                  if idx is not None:
                      self.modif[idx] += 1

                  return temp_flow
          start[self.posit_map[u]] += 1

      # Returns maximum flow in the graph
```

Listing 4: sendFlow method implementation

- Consider a certain level graph iteration (the one created by the BFS function call). The **sendFlow**(from **s** to t) method starts by looking for a path starting from the **first outgoing** edge from s, on which sending flow towards the sink. This is done by performing a modified DFS on the previously defined level graph (modified as it only consider non "saturated" edges, cf line 12). If such a path is found, recall that is the minimum within the composing edges' capacities to determine the amount of flow that can be sent.

  In doing so it behaves as the example in Figura 3: if such a path is found, on each of its edges the flow is augmented and the reversed edge (e.rev) keeps score of how much the corresponding debt will have to be decreased(cf. line 24). Moreover the pushed flow is returned and saved in the variable "total", that at the end of Dinic's algorithm will represent the maximized debt on the edge.

  Consider the toy example constructed in Figura 6 as an example, imagine to maximize the flow on the (F, E) edge. Given the level graph constructed there, the sink is reached in just one step and the corresponding flow is sent (in this case the pushed flow equals the edge's capacity, then the debt is eliminated and therefore not represented here).

- Then Dinic' s algorithm iterates the same procedure on a path starting from the **second outgoing** edge, and so on until termination. The recursive nature of the sendFLow function allow to test as paths all the combinations of the level graph edges.

In the example, no other path for reaching the sink is available and thus the algorithm proceeds by reconstructing the level graph (on the right). This in general differs from the previous, as some edges have been saturated in the process. This new level graph is able to signal the other path towards E, the one passing first by D, that is indeed the one through which flow is pushed in the next sendFlow function call.

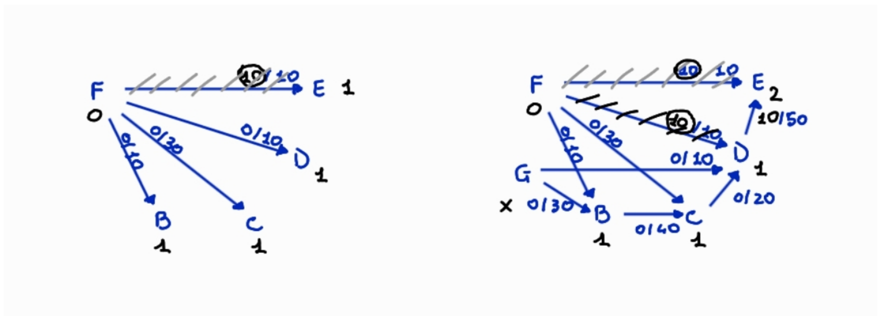


Figura 7: Toy example, (E, F) edge maximization

Notice in the left panel below how the edge (D, E) is still able to receive some flow. However, since the (F, D) one is now saturated, the algorithm must check whether there exists a path from F to D in the original graph. Fortunately this is the case and the level graph on the right indicates the path correctly.

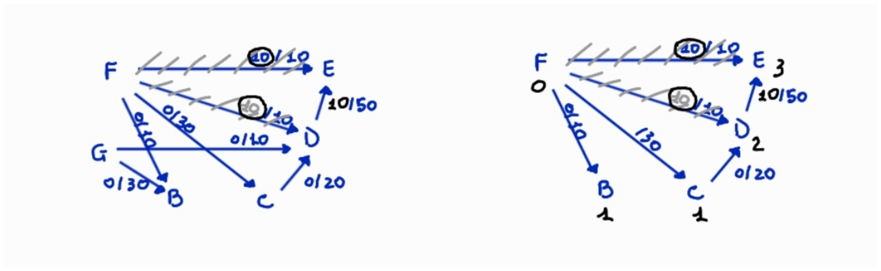


Figura 8: Toy example, (E, F) edge maximization

The next push of flow is in the following left panel. Consider instead the right panel: this represents the next level graph resulting from the updated flow values. The blocking flow has been reached, as there is not enough not saturated edges that can bring us from F to E: in this case the level graph construction would not be able to reach, starting from the source, the sink and therefore the algorithm terminates.

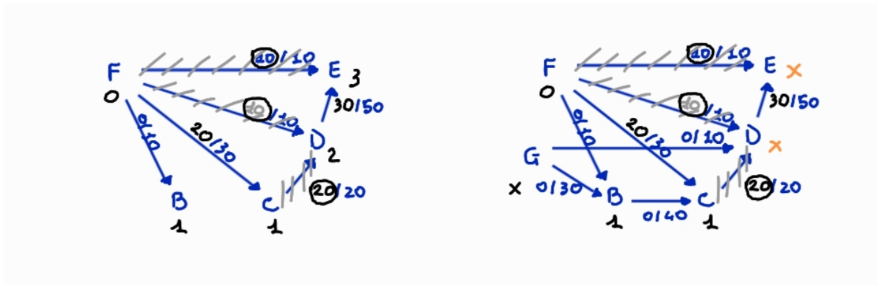


Figura 9: Toy example, (E, F) edge maximization

The variable counting the total flow being pushed, which has been updated each time a feasible path was found, now has value: $10 + 10 + 20 = 40$. This represents, as we said before, the maximized debt we now have to set the (F, E) debt to. Remember that we also have to decrease each modified edge of a quantity equals the final flow of that edge, obtaining the following restructured debt:

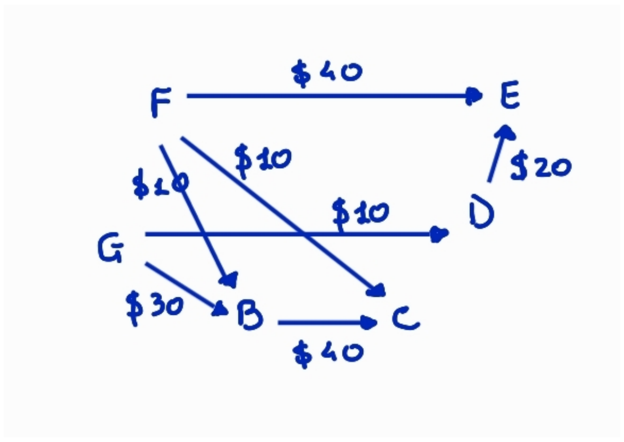


Figura 10: Toy example: (E, F) edge maximized

Notice that the max flow problem may not always provide neither a unique nor a globally optimal solution, especially if the graph is very large [3]. Nevertheless Dinic's implementation is particularly fast (refer to section 6).

In this subsection we have thus been able to determine the amount of debt (at least a temporary one) of a given edge.

3.2 Debt simplification on the whole graph

Now we are going to deal with the first question of the problem, determining if a given edge is going to be part of the graph after simplifying debts. Recall how, in the previous subsection, we have been able to eliminate some edges from the graph by "summarizing" the overall debt situation into the edge whose debt we were maximizing.

In order to answer the first question we are just going to **repeat on each edge** the maximization task of before. Notice how I previously referred to the weight we found maximizing an edge, as a temporary one: this is due to the fact that repeating the task for a different edge, the previous may be part of an augmenting path s - t and thus its capacity might still change as a result. The edges with a **capacity greater than zero** after this process, will be the one part of the final debt simplified graph.

The following pseudocode illustrates the procedure.

Algorithm 2 Debt Simplification via Max-Flow

- **Step 1)** Feed the debts in the form of a directed graph to the algorithm.
- **Step 2)** Select **one of** the **non-visited edges**, say (u, v) from the graph. Reset the list `modif`, that indicate which edges are modified in Step 3).
- **Step 3)** Now with **u** as source and **u** as sink, run **Dinic**'s maxflow algorithm to determine the maximum flow of money possible from u to v.
- **Step 4)** As a result of the previous step, **update** the (u, v) debt with the maximized debt. Also, accordingly to the list `modif`, update the debts of modified edges, and properly **reset** flow and capacities in order to be able to apply Dinic's algorithm again.
- **Step 5)** Now go back to **Step 1**, and feed in the **modified graph**. Once all the edges are visited, the simplified graph is composed of the edges with a capacity greater than zero.

In addition to updating the capacities of modified edges (and also of the maximized edge), for Dinic's algorithm to work correctly we must also restore edges' flow and capacities properly. Refer to the *Settle* function in the chunk 5 in the next section.

In the following I attach the (G, D) maximization. As one can see from the result, in this case no 'summarization' was possible (since no other augmenting path exists from G to D besides the direct one)

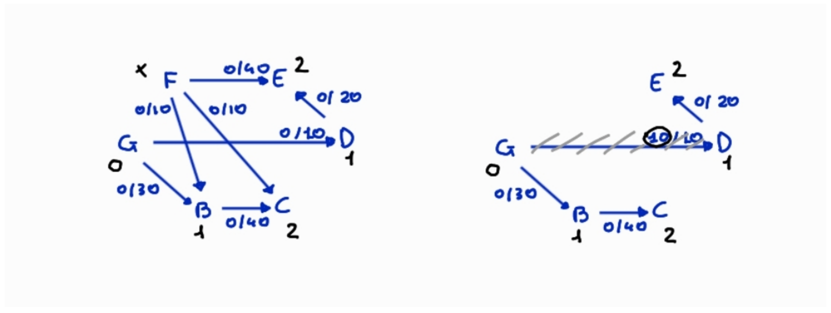


Figura 11: Toy example, (G, D) edge maximization

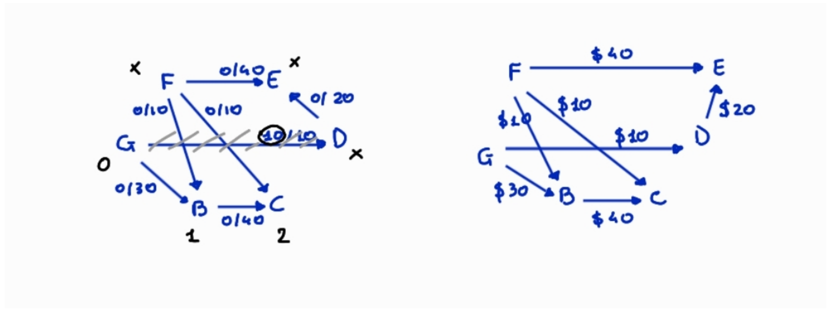


Figura 12: Toy example, (G, D) edge maximization

The other edges proceed similarly, resulting in the final debt simplified graph:

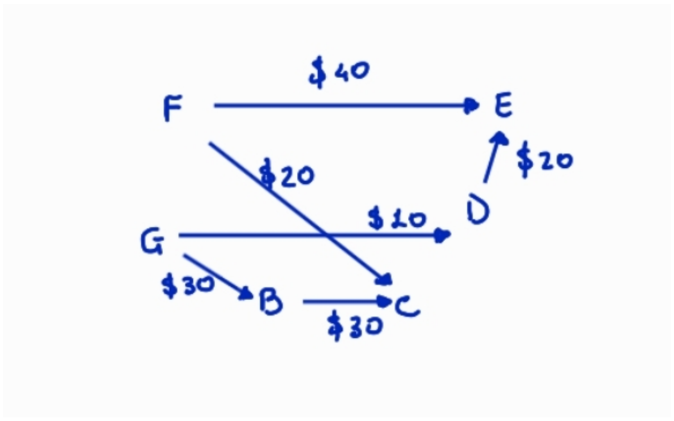


Figura 13: Toy example: Final simplified Graph

The above algorithm can be implemented as a Graph's method that resolves the debt situation, reported here.

```python
def Settle(self):
  #invert the list in order to mantain the order despite using pop(),
  #this way the operation remaining.pop() has O(1) cost
  remaining = self.edges[::-1]

  while len(remaining) > 0:

    #reset the list indicating modified edges
    self.modif = [0 for i in range(len(self.edges))]

    #pick an edge
    edge_maxim = remaining.pop()

    #maximize that edge's debt
    (v1,v2) = edge_maxim
    edge_maxflow = self.DinicMaxflow(v1, v2)

    #look in self.dicts[v1] for the index of the chosen edge
    i_edge_maxim = self.dicts[v1].get((v2))

    #update the chosen edge's debt
    self.adj[v1][i_edge_maxim].cap = edge_maxflow

    for q in [j for j in range(len(self.edges)) if (self.modif[j] > 0 and j
  ↪ != (len(self.edges)-len(remaining)-1))]:
      (orig, dest) = self.edges[q]

      #look for the right indices as before
      temp1 = self.dicts[dest].get((orig))
      temp2 = self.dicts[orig].get((dest))

      #flow pushed on an edge reduces its debt:
```

```
      #reduce (reverse edge’s flow is < 0) the edge’s debt
      self.adj[orig][temp2].cap += self.adj[dest][temp1].flow

      #flow of modified forward edges has to be restored to 0 for the Dinic algorithm to perform correctly
      self.adj[orig][temp2].flow = 0
      #similarly for modified backward edges, here also the capacity was initialized to 0
      self.adj[dest][temp1].flow = 0
      self.adj[dest][temp1].cap = 0

    #restore to 0 also the flow of the excluded forward edge above
    self.adj[v1][i_edge_maxim].flow = 0

    #restore to 0 both flow and cap of chosen edge’s reverse one
    i_edge_maxim_rev = self.dicts[v2].get((v1))
    self.adj[v2][i_edge_maxim_rev].flow = 0
    self.adj[v2][i_edge_maxim_rev].cap = 0

  #once the optimization phase is concluded, let’s look for the
  #non-zero capacity edges, in order to call nonzeroedgDraw()
  for i in range(len(self.edges)):
        (u, v) = self.edges[i]
        ind = self.dicts[u].get((v))
        if self.adj[u][ind].cap != 0:
          self.nonzeroedges.append((u,v))
```
Listing 5: Settle method implementation

4 Algorithm Analysis

Let me start by explaining the **data structures** used, crucial for the following discussion on the computational complexity.

- Why is an **adjacency list** a better choice than the alternatives for representing a graph in this case? Such a data structure, named `self.adj`, is used in the addEdge, BFS, sendFlow and Settle methods (code for the latter is provided below).

  The first one is frequently called during the Graph construction process. As shown in the relative code, each time a new transaction is encoded, two edges - a forward and a backward one - are added to the respective lists. Actually all the data structures representing a graph have $\mathcal{O}(1)$ as cost associated to this operation.

  In the sendFlow and Settle functions it is required to access a specific edge within an adjacency list. This operation would not be particularly efficient on its own, but it is optimized through the use of dictionaries.
  In the sendFlow case, **`self.posit_map`** provides the right index within the `start` list. The **`self.dicts`** has been implemented this way: `self.dicts[u][v]` stores the the position where v has been saved in the `self.adj[u]` list. This allow to retrieve the v position within this list in constant time.

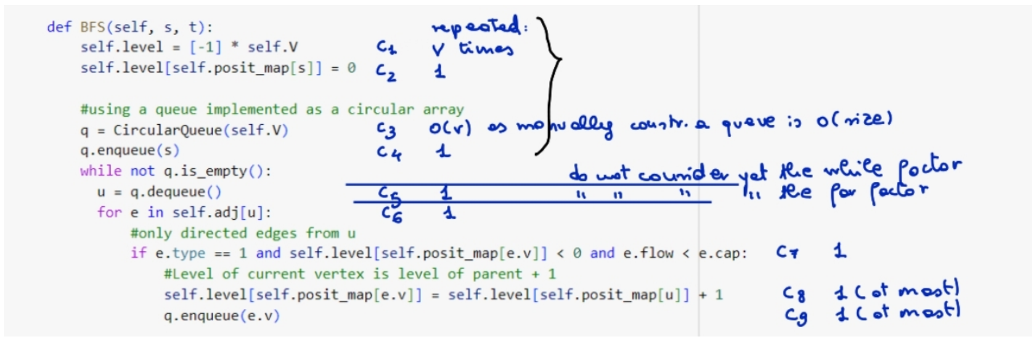
  It is actually in the BFS function that the use of an adjacency list is appreciable. Indeed in order to construct the level graph, we need to iterate on the outgoing edges, operation performed in $\mathcal{O}(deg(v))$ by this data structure. This is particularly important given the frequency of this operation in Dinic's algorithm.

- Each time a new edge is encoded, it is added to the `self.edges` list in the form (u, v), and its index within this list is saved in the **`self.edge_index` dictionary**(cf. addEdge method). Once more this allows to later retrieve the index in $\mathcal{O}(1)$ time.

- The modified BFS that defines the level graph exploits a queue, that is efficiently implemented as a **circular array**.

- In the BFS and the sendFlow functions it is often required to access the lists `start[]` and `self.level[]`, in order to retrieve some specifics of a given vertex (the level graph index for the latter). Within those we refer to the order defined by `self.posit` and `self.posit_map`. Also in this case the latter is a **dictionary** used to access indeces efficiently, in constant time.

Now we are able to evaluate this algorithm's **computational complexity**. As the BFS and sendFlow methods are called in the `DinicMaxflow` function, let's start there:



```
def BFS(self, s, t):
    self.level = [-1] * self.V
    self.level[self.posit_map[s]] = 0

    #using a queue implemented as a circular array
    q = CircularQueue(self.V)
    q.enqueue(s)
    while not q.is_empty():
      u = q.dequeue()
      for e in self.adj[u]:
          #only directed edges from u
          if e.type == 1 and self.level[self.posit_map[e.v]] < 0 and e.flow < e.cap:
              #Level of current vertex is level of parent + 1
              self.level[self.posit_map[e.v]] = self.level[self.posit_map[u]] + 1
              q.enqueue(e.v)
```

Figura 14: BFS computational cost

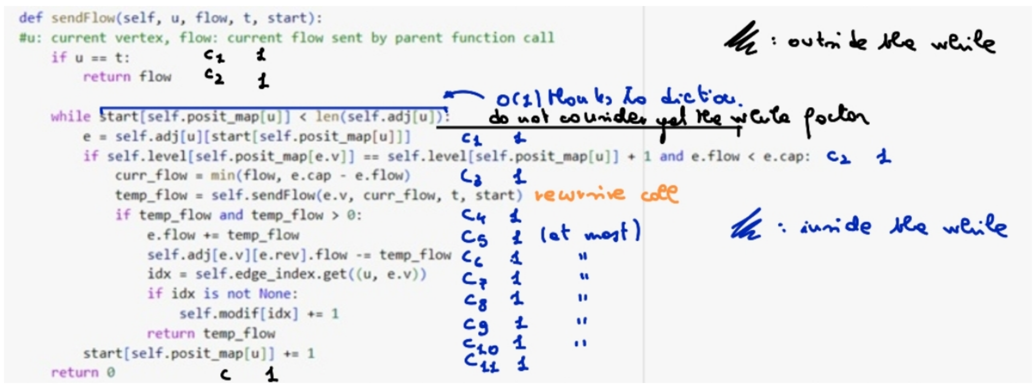
The cost inside the `for` is $\mathcal{O}(1)$. Since each `for` loop requires $deg(u)$ executions, the overall cost for the loop is: $\mathcal{O}(1)\mathcal{O}(deg(u)) = \mathcal{O}(deg(u))$.
This sums together with the constant cost of the `dequeue` operation giving an overall cost inside the `while` loop equals to $\mathcal{O}(deg(u))$. Observe how this cost is repeated for each vertex and thus in order to take care of the `while` loop we have to compute:

$$\sum_{allvertices} deg(u) = 2E = \mathcal{O}(E)$$

since each edge gives a +2 contribute to the overall degree (+1 to the degree of each vertex). Considering also the black bracket:

$$\mathcal{O}(1) + \mathcal{O}(V) + \mathcal{O}(E) = \mathcal{O}(V + E)$$

Notice how some of the constant computational costs have been possible thanks to the above mentioned dictionaries.



```
def sendFlow(self, u, flow, t, start):
#u: current vertex, flow: current flow sent by parent function call
    if u == t:
        return flow

    while start[self.posit_map[u]] < len(self.adj[u]):
        e = self.adj[u][start[self.posit_map[u]]]
        if self.level[self.posit_map[e.v]] == self.level[self.posit_map[u]] + 1 and e.flow < e.cap:
            curr_flow = min(flow, e.cap - e.flow)
            temp_flow = self.sendFlow(e.v, curr_flow, t, start)
            if temp_flow and temp_flow > 0:
                e.flow += temp_flow
                self.adj[e.v][e.rev].flow -= temp_flow
                idx = self.edge_index.get((u, e.v))
                if idx is not None:
                    self.modif[idx] += 1
                return temp_flow
        start[self.posit_map[u]] += 1
    return 0
```

Figura 15: sendFlow computational cost

Inside the `while` loop, recursive calls aside, the cost is $\mathcal{O}(1)$. This represents the cost of the function execution on a **single edge**.

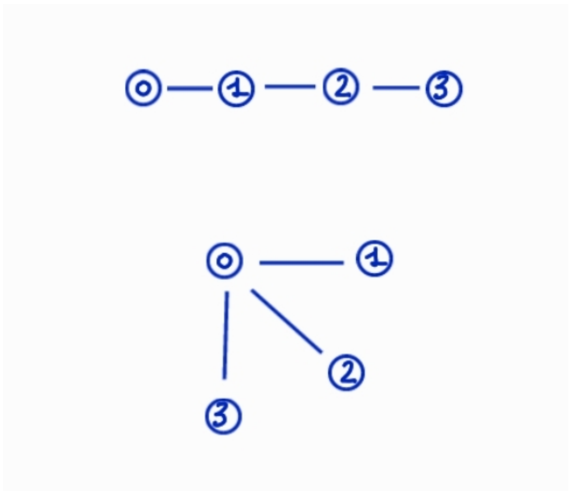


Figura 16: Level graph particular cases

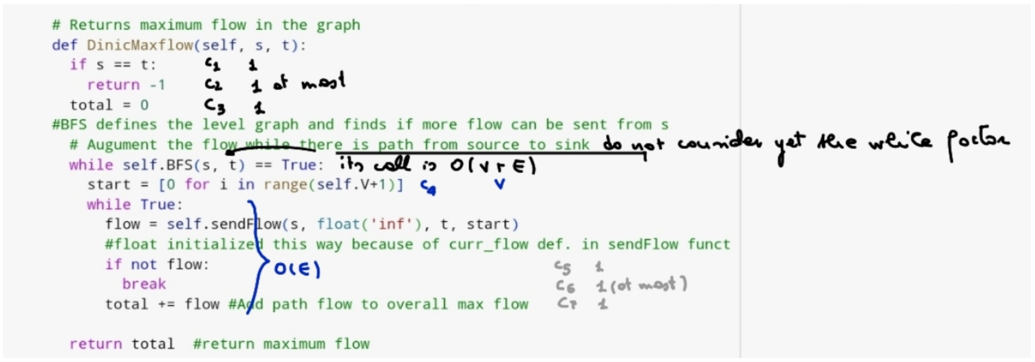
- The maximum **recursion depth** in the `sendFlow` recursive call is $\mathcal{O}(V)$, since one could imagine a level graph as the one displayed at the top. In such a case if the edges are not saturated, the recursion may involve $V - 1$ calls.
- On the other side, think about a level graph as the one at the bottom. Here the overall computational cost by the `while` loop, the one would normally identifies as $\mathcal{O}(deg(u))$, is surely $\mathcal{O}(E)$.

Recall the modified DFS structure encoded in the sendFlow function. Reaching a blocking flow is a process that operates pushing flow on each edge of an augmenting path (saturating some of them). The fundamental point is that, before a blocking flow is reached, **each edge** is considered **at most a finite number of times**.
Despite the recursion depth begin $\mathcal{O}(V)$, what actually matters in computing the cost of the method, is how many times each edge can be considered. Multiplying the cost on each edge by how many times is visited by the algorithm, one gets:

$$\mathcal{O}(1) + \mathcal{O}(E)\mathcal{O}(1) = \mathcal{O}(E)$$

where $\mathcal{O}(1)$ is due to the base case.

Now let's move to the `DinicMaxflow` method:



```
  # Returns maximum flow in the graph
  def DinicMaxflow(self, s, t):
    if s == t:
      return -1
    total = 0
  #BFS defines the level graph and finds if more flow can be sent from s
    # Augument the flow while there is path from source to sink
    while self.BFS(s, t) == True:
      start = [0 for i in range(self.V+1)]
      while True:
        flow = self.sendFlow(s, float('inf'), t, start)
        #float initialized this way because of curr_flow def. in sendFlow funct
        if not flow:
          break
        total += flow #Add path flow to overall max flow

    return total  #return maximum flow
```

Figura 17: DinicMaxflow computational cost

The inside `while` loop calls the `sendFlow` method until reaching a blocking flow, and performs the constant time operation of updating the `total` variable. Thus, from the previous reasoning, it has a cost of $\mathcal{O}(E)$. The other elements inside the inner `while` loop are:

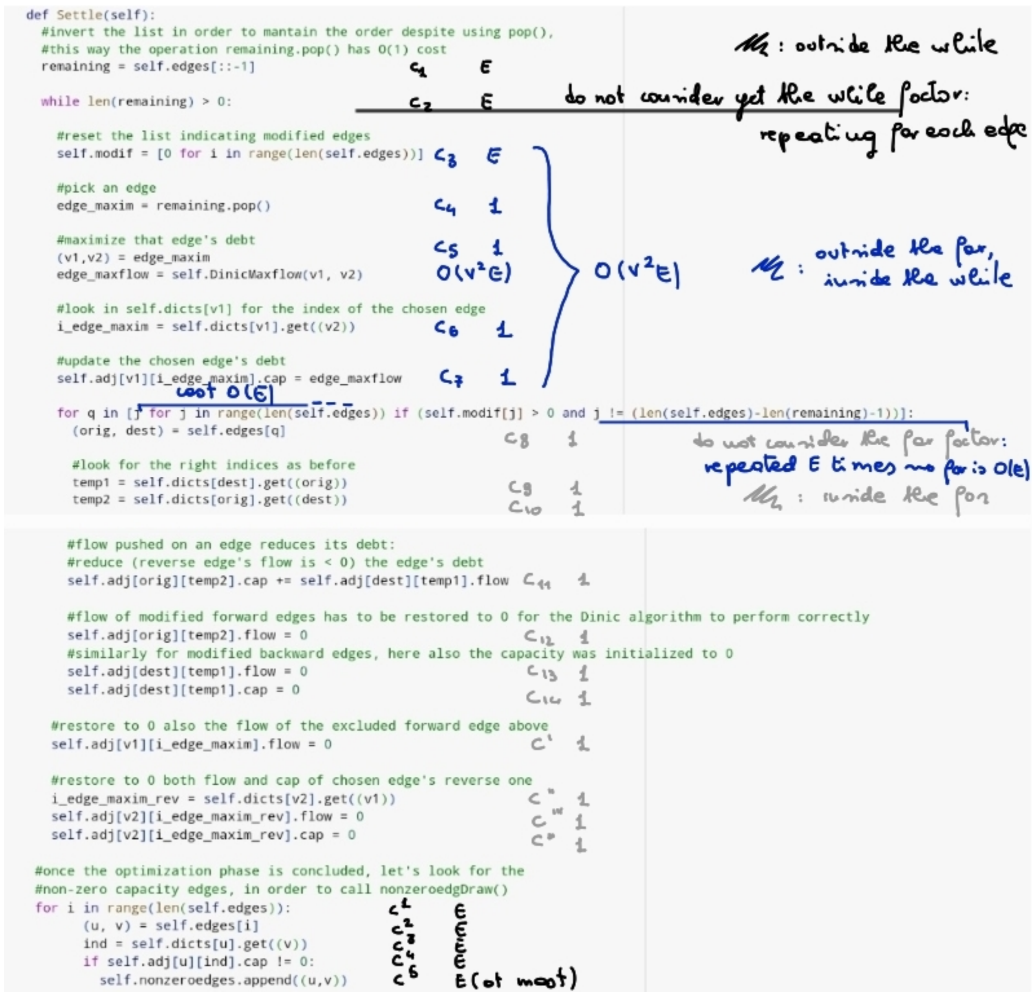
- The `start` line, cost of $\mathcal{O}(V)$
- The `BFS` call, with the related cost of $\mathcal{O}(V + E)$

The crucial point is now understanding, in the outer `while` loop, how many `BFS` calls returns `True`. Each `BFS` call constructs a new level graph. After the first have been created (and thus the sink has surely been reached), the sink level may be: $1, 2, 3, ..., V - 1$.
At each `sendFlow` call, at least one augmenting path is saturated. Thus, if the sink is still reachable, at the following `BFS` call, the minimum value for the sink is one higher than the one at the previous call. For example, if the sink level was 1 at the first call, at the second it can be one of: $2, 3, ..., V - 1$, and so on. After the maximization on the path where sink level was $V - 1$, since one of this path' edges is now saturated, there are no more paths towards the sink where flow can be pushed. Thus the next call would return `False`. The `BFS` method is called $\mathcal{O}(V)$ times.

Summing up, the computational cost of this method is:

$$\mathcal{O}(1) + \mathcal{O}(V)[\mathcal{O}(V + E) + \mathcal{O}(V) + \mathcal{O}(E)] = \mathcal{O}(VE)\mathcal{O}(V) = \mathcal{O}(V^2E)$$

The `DinicMaxflow` method is then used in the `Settle` one. Let's analyse its cost.



```
def Settle(self):
  #invert the list in order to mantain the order despite using pop(),
  #this way the operation remaining.pop() has O(1) cost
  remaining = self.edges[::-1]

  while len(remaining) > 0:

    #reset the list indicating modified edges
    self.modif = [0 for i in range(len(self.edges))]

    #pick an edge
    edge_maxim = remaining.pop()

    #maximize that edge's debt
    (v1,v2) = edge_maxim
    edge_maxflow = self.DinicMaxflow(v1, v2)

    #look in self.dicts[v1] for the index of the chosen edge
    i_edge_maxim = self.dicts[v1].get((v2))

    #update the chosen edge's debt
    self.adj[v1][i_edge_maxim].cap = edge_maxflow

    for q in [j for j in range(len(self.edges)) if (self.modif[j] > 0 and j != (len(self.edges)-len(remaining)-1))]:
      (orig, dest) = self.edges[q]

      #look for the right indices as before
      temp1 = self.dicts[dest].get((orig))
      temp2 = self.dicts[orig].get((dest))

      #flow pushed on an edge reduces its debt:
      #reduce (reverse edge's flow is < 0) the edge's debt
      self.adj[orig][temp2].cap += self.adj[dest][temp1].flow

      #flow of modified forward edges has to be restored to 0 for the Dinic algorithm to perform correctly
      self.adj[orig][temp2].flow = 0
      #similarly for modified backward edges, here also the capacity was initialized to 0
      self.adj[dest][temp1].flow = 0
      self.adj[dest][temp1].cap = 0

    #restore to 0 also the flow of the excluded forward edge above
    self.adj[v1][i_edge_maxim].flow = 0

    #restore to 0 both flow and cap of chosen edge's reverse one
    i_edge_maxim_rev = self.dicts[v2].get((v1))
    self.adj[v2][i_edge_maxim_rev].flow = 0
    self.adj[v2][i_edge_maxim_rev].cap = 0

  #once the optimization phase is concluded, let's look for the
  #non-zero capacity edges, in order to call nonzeroedgDraw()
  for i in range(len(self.edges)):
      (u, v) = self.edges[i]
      ind = self.dicts[u].get((v))
      if self.adj[u][ind].cap != 0:
        self.nonzeroedges.append((u,v))
```

Figura 18: Settle computational cost

Observe how reversing the `edges` list, we are able to later use `pop()`, instead of `pop(0)`, maintaining the order. The former executes in constant time. Starting from the inner sections, notice how the cost inside the `for` is $\mathcal{O}(1)$. Besides the $\mathcal{O}(E)$ cost for the `for` itself, another $\mathcal{O}(E)$ is required to determine the correct edges on which the previous `for` is going to operate.

The code inside the `while` loop, additionally to what said before, presents: a cost equal to $\mathcal{O}(E)$ in `self.modif`, an $\mathcal{O}(V^2E)$ relative to `DinicMaxflow`, and a global $\mathcal{O}(E)$ for the final `for`. Using the fact that $\mathcal{O}(E)$ is actually $\mathcal{O}(V^2E)$, one gets:

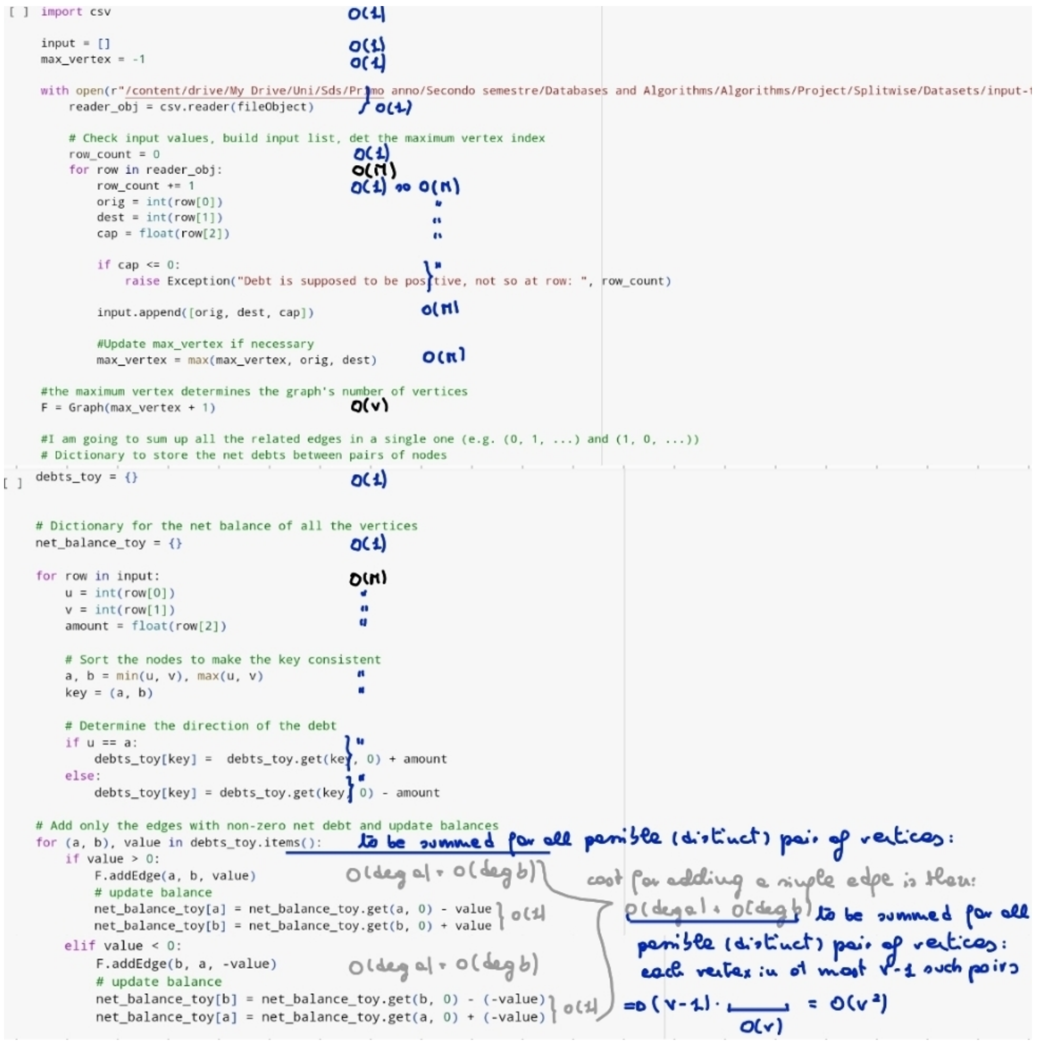
$$\mathcal{O}(E) + \mathcal{O}(E)[\mathcal{O}(V^2E) + 2\mathcal{O}(E) + \mathcal{O}(E)] = \mathcal{O}(V^2E^2)$$

Then the computational complexity for the overall algorithm is:

$$\mathcal{O}(V^2E^2)$$

In the Notebook the test datasets are imported from csv files. In principle we allow them to present different transactions for the same pair of people, but since Dinic algorithm is supposed to work with (at most) one edge for each pair of vertices, we first need to collect all the transactions between two fixed people (stored in an edge).

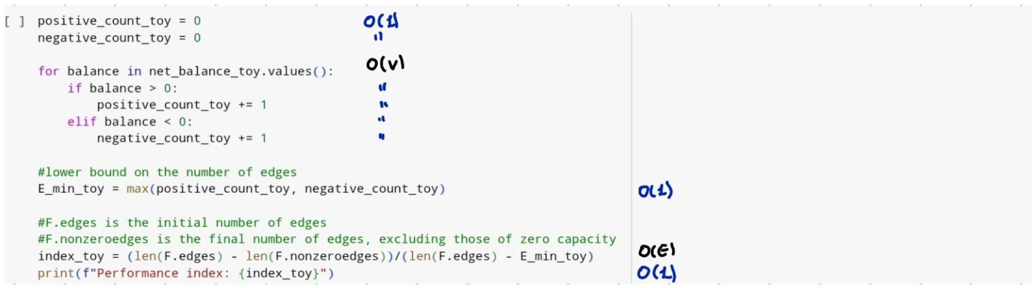
For the related code, together with the one for computing the performance index, refer to the Notebook. I proceed computing the complexity of this additional parts, still part of the overall debt simplification task.



```
[ ] import csv

    input = []
    max_vertex = -1

    with open(r"/content/drive/My Drive/Uni/Sds/Primo anno/Secondo semestre/Databases and Algorithms/Algorithms/Project/Splitwise/Datasets/input-
        reader_obj = csv.reader(fileObject)

        # Check input values, build input list, det the maximum vertex index
        row_count = 0
        for row in reader_obj:
            row_count += 1
            orig = int(row[0])
            dest = int(row[1])
            cap = float(row[2])

            if cap <= 0:
                raise Exception("Debt is supposed to be positive, not so at row: ", row_count)

            input.append([orig, dest, cap])

            #Update max_vertex if necessary
            max_vertex = max(max_vertex, orig, dest)

    #the maximum vertex determines the graph's number of vertices
    F = Graph(max_vertex + 1)

    #I am going to sum up all the related edges in a single one (e.g. (0, 1, ...) and (1, 0, ...))
    # Dictionary to store the net debts between pairs of nodes
[ ] debts_toy = {}


    # Dictionary for the net balance of all the vertices
    net_balance_toy = {}

    for row in input:
        u = int(row[0])
        v = int(row[1])
        amount = float(row[2])

        # Sort the nodes to make the key consistent
        a, b = min(u, v), max(u, v)
        key = (a, b)

        # Determine the direction of the debt
        if u == a:
            debts_toy[key] =  debts_toy.get(key, 0) + amount
        else:
            debts_toy[key] = debts_toy.get(key, 0) - amount

    # Add only the edges with non-zero net debt and update balances
    for (a, b), value in debts_toy.items():
        if value > 0:
            F.addEdge(a, b, value)
            # update balance
            net_balance_toy[a] = net_balance_toy.get(a, 0) - value
            net_balance_toy[b] = net_balance_toy.get(b, 0) + value
        elif value < 0:
            F.addEdge(b, a, -value)
            # update balance
            net_balance_toy[b] = net_balance_toy.get(b, 0) - (-value)
            net_balance_toy[a] = net_balance_toy.get(a, 0) + (-value)
```

O(1)
O(1)
O(1)
O(1)
O(1)
O(M)
O(1) ⇒ O(M)
"
"
"
"
O(M)
O(M)
O(V)
O(1)
O(1)
O(M)
"
"
"
"
"
"
"
to be summed for all possible (distinct) pair of vertices:
O(deg a) + O(deg b)
cost for adding a single edge is then:
O(deg a) + O(deg b) to be summed for all possible (distinct) pair of vertices: each vertex in at most V-1 such pairs
O(1)
O(deg a) + O(deg b)
O(1)
$\Rightarrow (V-1)\cdot \underbrace{\quad}_{O(V)} = O(V^2)$

Figura 19: Csv input computational cost



```
[ ] positive_count_toy = 0
    negative_count_toy = 0

    for balance in net_balance_toy.values():
        if balance > 0:
            positive_count_toy += 1
        elif balance < 0:
            negative_count_toy += 1

    #lower bound on the number of edges
    E_min_toy = max(positive_count_toy, negative_count_toy)

    #F.edges is the initial number of edges
    #F.nonzeroedges is the final number of edges, excluding those of zero capacity
    index_toy = (len(F.edges) - len(F.nonzeroedges))/(len(F.edges) - E_min_toy)
    print(f"Performance index: {index_toy}")
```

Figura 20: Performance index computational cost

These chunks of code comport an additional cost of $\mathcal{O}(M) + \mathcal{O}(V^2) + \mathcal{O}(E)$, where by M I denote the number of initial transactions. Thus, the overall computational complexity is:

$$\mathcal{O}(V^2 E^2) + \mathcal{O}(M)$$

5 Simulations

Now we can address two real scale instances of the problem: the `input-large` and `input` datasets. The first contains 913 debts among 11 people, while the second includes 100 debts among 20 people. These are amounts that could realistically occur over several months or even a year of recorded transactions within a typical-sized group of friends or colleagues.

Since reporting the values of all transactions after debt simplification would be of little use, I will instead focus on how effectively the algorithm reduces the number of non-zero edges, both visually and through the performance index we previously defined. In the following I present the two datasets graphs before and after the simplification.

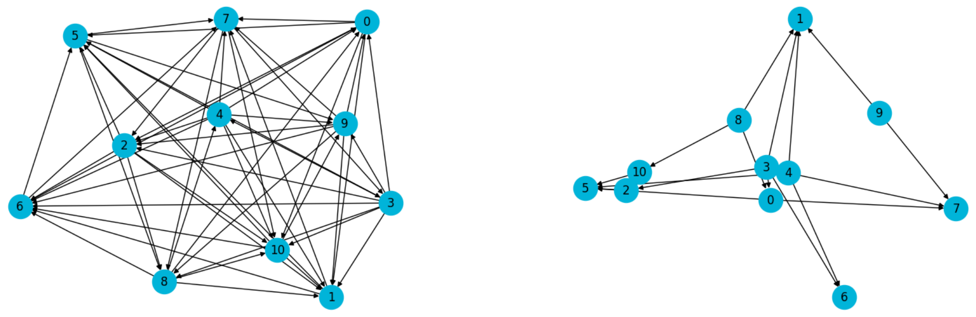


Figura 21: Index for the `input-large` dataset: 0.69

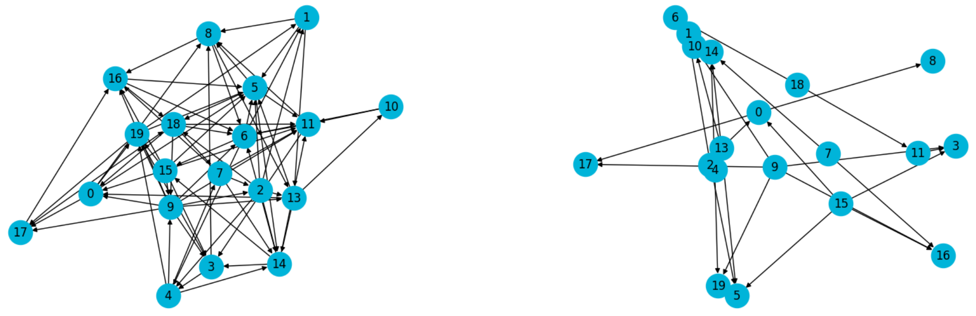


Figura 22: Index for the `input` dataset: 0.73

In the `input-large` dataset, the 55 initial debts have been reduced to 17. Instead in the `input` dataset the reduction has been from 77 to 28, as indicated in the Notebook. In both cases the performance indexes illustrate a significant reduction in the number of debts compared to the maximum possible: 69% and 73% respectively.

6 Conclusions

As I discussed in the computational complexity section, the implemented algorithm reaches a complexity of $\mathcal{O}(V^2E^2)$:

- $\mathcal{O}(V^2E)$: related to the **Dinic** implementation of the *Maximum flow problem*
- The extra $\mathcal{O}(E)$ is due to the algorithm iterating over all the **edges** in the graph

Actually there are other implementations of the *Maximum flow problem* with a complexity of $\mathcal{O}(EV)$, which can be used if further improvement is needed ([4]). If one of these is implemented, the overall algorithm's complexity would reduce to $\mathcal{O}(E^2V)$, scaling pretty well for the real-world use case of this problem.

Article [6] proposes various ideas to reduce the number of edges in the graph, which could be integrated into the proposed algorithm. To name one: for any cycle found in the debt graph, the debt of all the edges in the cycle can be reduced by the minimum debt among them. The resulting graph preserves the debt situation but has fewer edges.

On the other side, consider the real-world scenario where not only a fee must be paid for each transaction, but it may even be proportional to the amount of money being transferred (as using Paypal). In such a case, we are not only interested in reducing the number of transactions, but also in **minimizing** the total **amount of money transferred**. I would like to present an improvement for this kind of situation, proposed in [6].

6.1 Minimum weight solutions

Let me start by introducing a few useful facts:

- A **transfer** graph is a type of graph that satisfies the following condition: for every vertex i, the sum of the debts of its incoming edges *minus* the sum of the debts of its outgoing edges equals b_i (as in the graphs we have considered so far).
- The **weight** of the graph is the sum of the debts on all the edges (total amount transferred).
- A **lower bound on the weight** of a transfer graph is:

  $$\sum_{i:b_i>0} b_i \tag{1}$$

  This is due to:

  - $\sum_{i:b_i>0} b_i = -\sum_{i:b_i<0} b_i$ (see toy example in the Notebook for a quick check)
  - Therefore, the one on the left is the minimum amount that necessarily has to be exchanged.

In a transfer graph, any directed **path of length two or more can be shortened**. Consider a directed path of length two from p to r with amounts t and u, as shown on the left in the following Figure, . Let $m = t \wedge u$. Imagine a zero-labeled edge from p to r, and decrease the cycle p, q, r by m, as shown in the second graph from the left (the zero-labeled edge was reversed because, as a result of the transformation, its debt became negative). At least one of the resulting edges gets a zero debt. Which one(s), depends on how t and u compare (see the three possible cases on the right). In the case of $t = u$, the number of edges has decreased by one, whereas in the other cases it has remained the same.

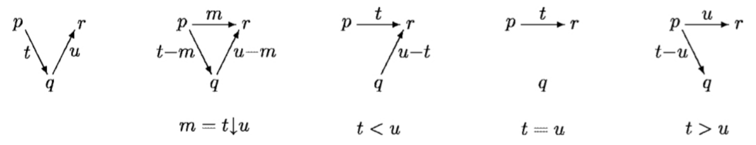


Figura 23: Shortening a path and decreasing the weight.

The important point is that, by **applying this transformation repeatedly**, a transfer graph is obtained in which each vertex has either no outgoing or no incoming edges (cf. three on the right with respect to the first on the left). Such a graph is known as a **bipartite graph**, in which the vertices are partitioned into two groups, and all edges go from one group to the other and not back or within groups. Vertices with positive balance ($b_i > 0$) have incoming edges only, points with negative balance ($b_i < 0$) have outgoing edges only. Points with zero balance ($b_i = 0$) have no edges at all.

Proposition 2 *A transfer graph with minimum weight must be bipartite*

Indeed, a graph that is not bipartite (as defined above), has a weight that can be surely reduced, as it must contain at least one path of length two. Thus, one can apply the same strategy as shown in the picture to reduce the weight.

Proposition 3 *The weight of a bipartite transfer graph is* $\sum_{i:b_i>0} b_i$

This is obtained noticing how each edge starts from a vertex with a negative balance and end by construction in a vertex with a positive balance. But all balances have to be evened out.

But on account of Equation 1, the above is the least possible weight! Thus,

All bipartite transfer graphs with this structure **achieve the minimum weight**.

Therefore, the above reasoning suggests **constructing a bipartite transfer** graph in order to also achieve the minimum weight. This can be done through an algorithm described in [6].

References

[1] Anton Cao. *The Splitwise Problem*. Accessed: 2025-04-15. 2022. URL: https://antoncao.me/blog/splitwise.

[2] GeeksforGeeks Users. *Dinic's algorithm for Maximum Flow*. Accessed: 2025-04-12. 2023. URL: https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/.

[3] GeeksforGeeks Users. *Minimize Cash Flow among a given set of friends who have borrowed money from each other*. Accessed: 2025-03-09. 2020. URL: https://www.geeksforgeeks.org/minimize-cash-flow-among-given-set-friends-borrowed-money/.

[4] Mithun Mohan K. *Algorithm Behind Splitwise's Debt Simplification Feature*. Accessed: 2025-03-07. 2019. URL: https://medium.com/@mithunmk93/algorithm-behind-splitwises-debt-simplification-feature-8ac485e97688.

[5] Splitwise. *Debts made simple*. Accessed: 2025-03-08. Sept. 2012. URL: https://blog.splitwise.com/2012/09/14/debts-made-simple/.

[6] Tom Verhoeff. *Settling multiple debts efficiently: An invitation to computing science*. Accessed: 2025-04-08. 2004. URL: https://www.researchgate.net/publication/220396130_Settling_Multiple_Debts_Efficiently_An_Invitation_to_Computing_Science/.

[7] WilliamFiset. *Dinic's Algorithm — Network Flow — Graph Theory*. Accessed: 2025-04-10. 2019. URL: https://www.youtube.com/watch?v=M6cm8UeeziI&pp=0gcJCdgAo7VqN5tD.