

# Minimização de Valores de Arestas em um Grafo

Pedro Vanzella

19 de novembro de 2015

## Resumo

Uma solução para o problema de minimização do valor de imposto pago sobre transferências bancárias é apresentado. Utiliza-se um grafo para representar o conjunto de transferências, e minimiza-se o valor das arestas, através da busca por transitividades.

## 1 Introdução

Uma recente mudança na regulamentação de impostos reativou uma antiga taxa sobre operações financeiras. Esta taxa, chamada de CPMF, incide em 1% sobre toda e qualquer transação bancária.

Um banco teve a idéia de minimizar o valor total pago deste imposto através de atalhos em transferências realizadas internamente.

Por exemplo, digamos que haja cinco correntistas, 1, 2, 3, 4 e 5, e haja as seguintes transferências entre eles:

1 transfere \$500 para 2.

2 transfere \$230 para 3.

3 transfere \$120 para 4.

1 transfere \$120 para 4.

2 transfere \$200 para 5.

É possível fazer quatro transferências, respeitando os valores iniciais e finais de saldo das contas destes cinco correntistas, mas minimizando o valor de cada transferência, de modo a pagar menos imposto:

- 1 transfere \$70 para 2
- 1 transfere \$110 para 3
- 1 transfere \$240 para 4
- 1 transfere \$200 para 5

Podemos ver que, em ambos os casos, o total enviado e o total recebido não foi alterado - apenas os valores parciais mudaram e, com eles, o valor pago em impostos.

Do ponto de vista dos correntistas, nada mudou - *e.g.* o extrato do correntista 1 ainda mostrará duas transferências, uma de \$500 para o correntista 2 e uma de \$120 para o correntista 4 - , mas internamente as transferências realizadas foram bastante diferentes. Podemos ver uma representação gráfica disto na Figura 1.

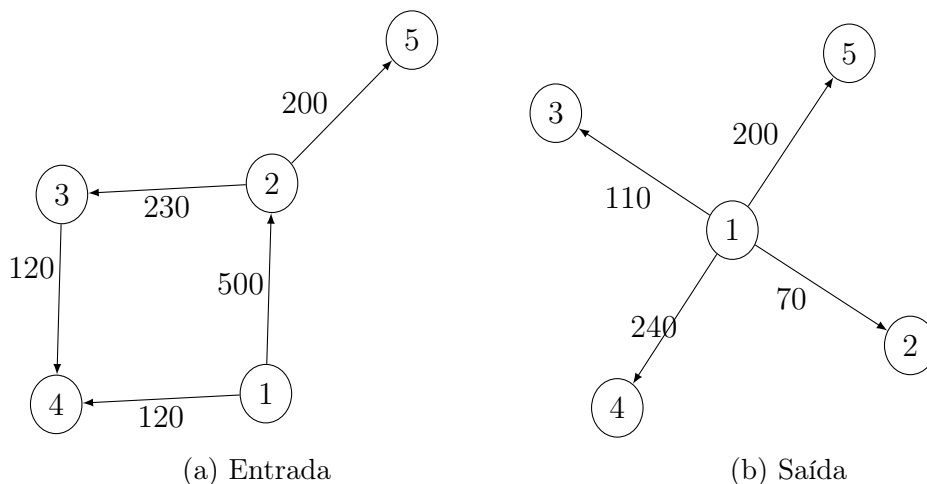


Figura 1: Representação da entrada e da saída como grafos

## 2 Entrada

O arquivo de entrada está no formato mostrado na Figura 2. A primeira linha tem dois valores: a quantidade de correntistas e a quantidade de transações descritas no arquivo. Como veremos na Seção 3, estas informações não serão necessárias.

5	5
1	2 500
2	3 230
3	4 120
1	4 120
2	5 200

As linhas seguintes têm três valores cada: o correntista que originou a transação, o correntista de destino da transação, e o valor da transação. Por exemplo, na linha 2 da Figura 2, lemos “*Uma transferência de 500 da conta do correntista 1 para a conta do correntista 2.*”.

Figura 2: Arquivo de entrada

Este arquivo de entrada representa o exemplo da Seção 1 e da Figura 1a.

### 3 Estrutura de Dados

A estrutura escolhida para representar o grafo foi a de Lista de Adjacências, como mostrado na Figura 3.

Outras alternativas, como a representação em Matriz de Adjacências também são possíveis, acarretando custos diferentes para acessos e remoções. No entanto, o acesso aos adjacentes de um nodo por Lista de Adjacentes se dá em  $O(adj)$ , enquanto por Matriz de Adjacências se dá em  $O(n)$ , onde  $n$  é o número de nodos do grafo. Como espera-se que haja menos arestas em um nodo do que nodos no grafo, a performance da Lista de Adjacências tende a ser ligeiramente melhor para esta aplicação.

No entanto, o fator principal para a escolha da representação por Lista de Adjacências foi a facilidade de se trabalhar com ela.

```

class Graph:
2   public list<Node> nodes

4   class Node:
        public int val
6        public list<Edge> edges

8   class Edge:
        public Node from
10        public Node to
        public int val
12

```

Figura 3: Representação das classes do grafo

Para ler o arquivo de entrada e criar os nodos e arestas, utilizamos o Algoritmo 1. Veja que ele está na classe **Mardita**, que contém uma instância do grafo.

---

**Algoritmo 1:** Criação de Nodos e Arestas

---

**Classe:** Mardita

**Entrada:** Arquivo como o da Figura 2

**Saída:** Instância da classe **Graph**

**para cada** linha  $l$  no arquivo, exceto a primeira **faça**

```

    partes ← l.separa(' ');    /* Separa a linha nos espaços */
    nodo_a ← self.graph.add_node(partes[0])
    nodo_b ← self.graph.add_node(partes[1])
    nodo_a.add_edge(nodo_b, partes[2]); /* Liga A com B, com
    valor partes[2] */

```

---

Onde `add_node` está descrito no Algoritmo 2 e `add_edge` está descrito no Algoritmo 3.

---

**Algoritmo 2:** Criação de Nós

---

**Classe:** Graph

**Entrada:** val: Inteiro, representando o nome do Nó

**Saída:** Instância da classe **Node**

**para todo**  $n$  *em*  $self.nodes$  **faça**

**se**  $n.val = val$  **então**

**retorna**  $n$  ;       /\* Se o nó já existe, retorna ele \*/

$n \leftarrow \text{Graph.Node}(val)$  ;       /\* Chama o construtor de Node \*/

$self.nodes.add(n)$  ;       /\* Adiciona à lista de nós \*/

**retorna**  $n$

---

O Algoritmo 2, que pertence à classe **Graph**, primeiro verifica se já há um nó com este nome em sua coleção de nós. Caso haja, retorna ele. Se não houver, chama o construtor da classe **Node** para criar um novo nó, adiciona à sua coleção e então retorna o nó criado.

A primeira vista, poderíamos ter utilizado um *set* em vez de uma lista para armazenar a coleção de nós, dado que não queremos dois nós iguais nela. No entanto, a unicidade garantida seria do objeto nó, quando queremos na verdade a unicidade do nome do nó.

---

**Algoritmo 3:** Criação de Arestas

---

**Classe:** Node

**Entrada:** to: Nó de origem; val: Inteiro, representando o valor da aresta

**Saída:** Instância da classe **Edge**

**para cada**  $e$  *em*  $self.edges$  **faça**

**se**  $e.to = to$  **então**

**retorna**  $e.update(val)$  ;       /\* Se a aresta já existe,  
        aumenta seu valor \*/

$e \leftarrow \text{Graph.Edge}(self, to, val)$  ;       /\* Cria nova Edge \*/

$self.edges.add(e)$  ;       /\* Adiciona à coleção de arestas \*/

**retorna**  $e$

---

O Algoritmo 3 é parecido com o Algoritmo 2, pois ele verifica a unicidade da aresta. A diferença é que arestas são consideradas iguais, neste problema caso suas origens e destinos sejam iguais. Como estamos verificando todas as arestas que partem de um nó, basta comparar o destino.

O construtor da aresta recebe três parâmetros: *de onde*, *para onde* e o *valor* da aresta.

Caso a aresta já exista, soma-se o valor das transações, de modo a criar apenas uma aresta entre dois nodos. Um exemplo disto pode ser visto na Figura 4. O grafo da Figura 4a representa duas transações, com a mesma origem e o mesmo destino. Ao somar-se o valor de suas arestas, temos o grafo da Figura 4b. Este é o que é utilizado pelo algoritmo.

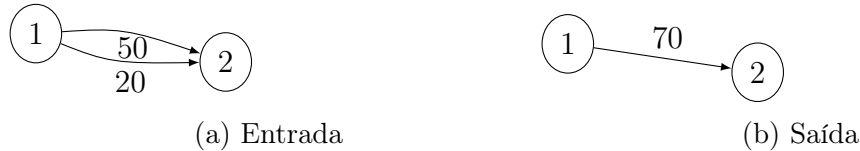


Figura 4: Consolidação de arestas com mesma origem e destino

## 4 Algoritmo

Há duas coisas a serem feitas para resolver o problema: precisamos calcular quanto imposto é pago (Seção 4.1) e reduzir o número de arestas no grafo, bem como seus valores (Seção 4.2).

### 4.1 Cálculo de Total de Imposto Pago

Este algoritmo é executado duas vezes - uma antes de reduzir-se as arestas, e uma após, de modo a sabermos qual foi a economia.

---

#### Algoritmo 4: Cálculo de Imposto Pago

---

**Classe:** Mardita

**Entrada:** Todas as arestas do grafo

**Saída:** Total de imposto pago

$total \leftarrow 0$

**para todo**  $e$  *em*  $self.graph.edges$  **faça**

$total \leftarrow total + e.valor$

**retorna**  $total \times 0.01$

---

No Algoritmo 4 vemos como o total de imposto é calculado. Apenas acumula-se o valor de todas as arestas e multiplica-se por 0.01, que é o percentual do imposto.

Nota-se que está acessando-se a propriedade `edges` da classe **Graph**, mas a mesma não parecia ter acesso às arestas, conforme visto na Figura 3.

De fato, a lista de arestas está na classe **Node**. Para termos acesso a elas, basta termos um método na classe **Graph** que itera por todos os nodos, coletando todas as arestas. A unicidade das arestas é garantida no momento de inserção, então pode-se fazer como é visto no Algoritmo 5.

---

**Algoritmo 5:** Coleção de todas as arestas

---

**Classe:** Graph

**Entrada:** Uma instância da classe `textbfGraph`

**Saída:** Uma lista de instâncias da classe **Edge**

inicializa `edges` como uma lista vazia

**para todo**  $n$  em  $self.nodes$  **faça**

**para todo**  $e$  em  $n.edges$  **faça**

        └ adiciona  $e$  em `edges`

**retorna** `edges`

---

## 4.2 Redução das Arestas

Este é o algoritmo principal, onde o problema é de fato solucionado. O pseudocódigo pode ser visto no Algoritmo 6.

---

**Algoritmo 6:** Redução de Arestas

---

**Classe:** Mardita

**para todo**  $u$  em  $self.graph.nodes$  **faça**

$vs \leftarrow$  adjacentes de  $u$

**enquanto**  $vs$  não estiver vazia **faça**

$v \leftarrow vs.pop()$  ;                      /\* Remove um elemento de  $vs$  \*/

**para todo**  $a$  adjacente a  $v$  **faça**

**se** valor de  $\langle v, a \rangle <$  valor de  $\langle u, v \rangle$  **então**

$tmp \leftarrow$  valor de  $\langle v, a \rangle$ ; remove  $\langle v, a \rangle$ ; valor de  $\langle u, v \rangle$

$\leftarrow$  diminui de  $tmp$ ; cria aresta  $\langle u, a \rangle$  com valor  $tmp$ ;

**senão**

**se**  $v$  está em  $vs$  **então**

                    remove  $v$  de  $vs$  ;    /\*  $v$  pode ter sido conectado

                    novamente graças a um ciclo \*/

$tmp \leftarrow$  valor de  $\langle u, v \rangle$ ; remove  $\langle u, v \rangle$ ; valor de  $\langle v, a \rangle$

$\leftarrow$  diminui de  $tmp$ ; cria aresta  $\langle u, a \rangle$  com valor  $tmp$ ;

$vs \leftarrow a$  ; /\* Nodo  $a$  é agora é adjacentes de  $u$  \*/

            ;

---

A idéia do Algoritmo 6 é encontrar transitividade entre os nodos - isto é, para um grafo  $G = \{n, e\}$  com  $n = \{A, B, C\}$  e  $e = \{(A, B, x), (B, C, y)\}$ , gerar uma nova aresta  $(A, C, x - y)$ , remover a aresta  $(B, C, x)$  e alterar o valor da aresta  $(A, B)$  para  $y - x$ .

Há dois casos possíveis: o valor da primeira aresta pode ser maior que o da segunda (Figura 5a) ou o valor da primeira aresta pode ser menor ou igual ao da segunda (Figura 6a).

Na Figura 5 vemos o primeiro caso. Aqui pega-se o valor da aresta  $(2, 3)$ , diminui-se ela da aresta  $(1, 2)$ , remove-se  $(2, 3)$  e cria-se  $(1, 3)$ , com o valor que se tinha em  $(1, 2)$ . O resultado disto, aplicado à Figura 5a pode ser visto na Figura 5b.

Caso já exista uma aresta  $(1, 3)$ , isto não é um problema, já que o Algoritmo 3 cuida para que, ao adicionar uma aresta já existente, o valor da aresta seja apenas aumentado.



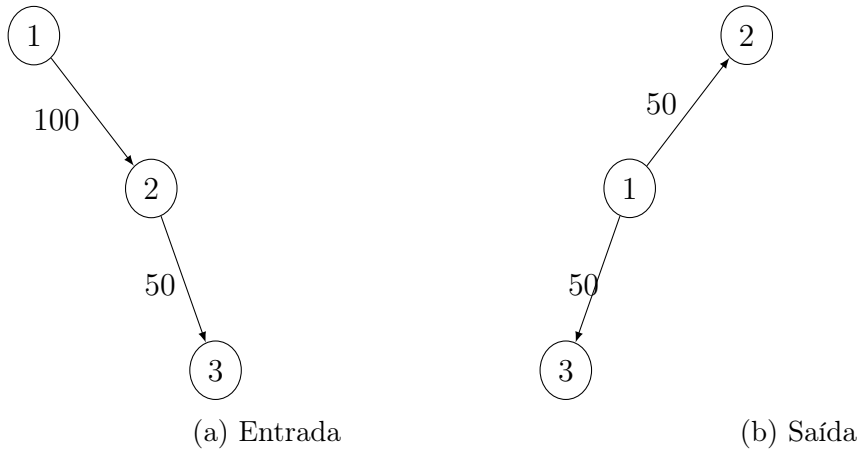


Figura 5: Redução aplicada a um grafo simples.

Na Figura 6 vemos o segundo caso. A entrada (Figura 6a) tem uma aresta  $(1, 2)$  com valor inferior à aresta  $(2, 3)$ . Neste caso, pegamos o valor de  $(1, 2)$  e removemos esta aresta. Então, subtraímos este valor da aresta  $(2, 3)$  e criamos a aresta  $(1, 3)$  com este mesmo valor. Isto gera o resultado visto na Figura 6b.

Novamente não precisamos nos preocupar ao adicionar uma aresta com o caso da mesma já existir, já que o Algoritmo 3 já cuida disto.

Há ainda um outro detalhe neste segundo caso: é possível que a aresta intermediária (no caso da Figura 6, a aresta 2), ter sido adicionada novamente à lista de adjacentes da primeira aresta (pelo *senão* do primeiro caso). Neste caso, é importante verificarmos a existência dela e removê-la da lista, para evitarmos acessar um ponteiro inválido ao tentar encontrar uma aresta entre  $u$  e  $v$  numa iteração futura do laço **enquanto**.

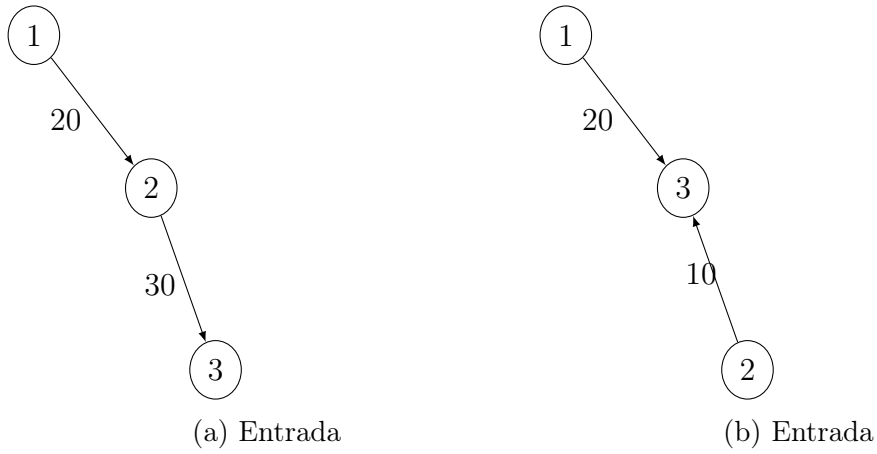


Figura 6: Redução aplicada a um grafo onde o valor de  $\langle v, a \rangle$  é superior ao de  $\langle u, v \rangle$

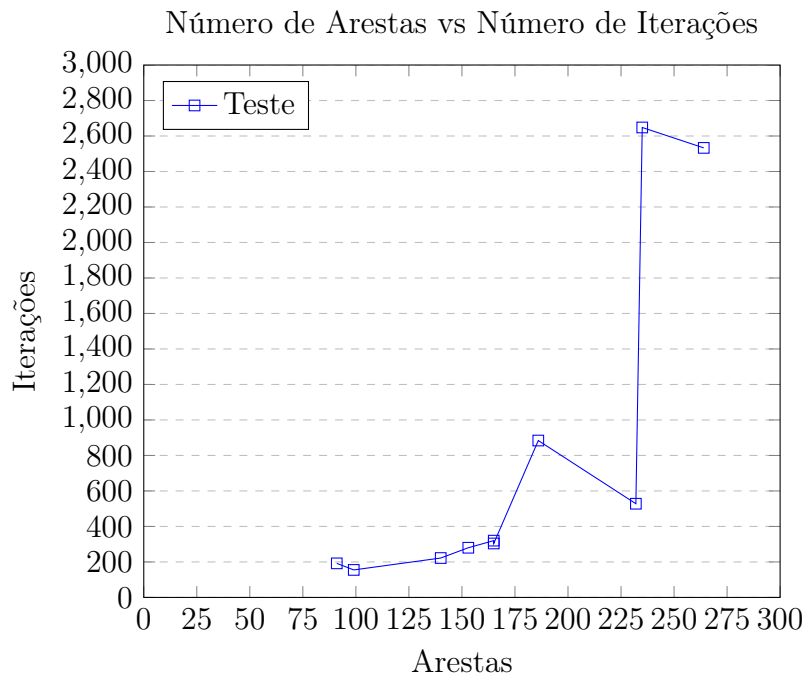
Pode-se facilmente validar o algoritmo com um teste de mesa na Figura 5, na Figura 6, ou mesmo na Figura 1.

Olhando-se para a Figura 5, vemos que os saldos dos correntistas são os mesmos (*i.e.*, o correntista 1 teve uma redução de 100 em seu saldo, o correntista 2 teve um aumento de 50, bem como o correntista 3). A única coisa que mudou entre a Figura 5a e a Figura 5b foram os valores e os destinos das transferências. Somando-se e multiplicando pelo valor do imposto, vemos que na Figura 5a pagaríamos 1.50 de imposto (*i.e.*, 1% de 150), enquanto na Figura 5b pagaríamos 1.00 de imposto (*i.e.*, 1% de 100), o que representa uma economia de 0.50.

## 5 Resultados

Teste	Economia	Transações	Iterações
1	5321.10	264	2533
2	4077.60	235	2648
3	2478.32	165	304
4	2225.54	140	222
5	1462.09	99	155
6	2921.33	186	884
7	1033.03	91	162
8	4124.53	232	528
9	2791.65	165	320
10	2428.67	153	280

Tabela 1: Resultados dos testes da Turma 128



É possível ver na Tabela 1 que há uma economia significativa em cada um dos casos de teste, e o processamento ocorre em um tempo bem aceitável.

Vemos também que o tempo de processamento é proporcional ao número

de transações. Isto faz sentido, dado que cada transação é uma aresta no grafo.

De fato, o algoritmo passará por cada aresta uma vez, e apenas uma vez. Isto daria uma complexidade de  $O(e)$ , onde  $e$  é o número de arestas. No entanto, novas arestas são criadas, e estas devem ser verificadas por transitividade, de modo a otimizar ao máximo as transações. Como cada par de arestas pode gerar uma nova aresta, a complexidade fica  $O(e + \frac{e}{2})$ . No entanto, a notação  $O$  despreza os termos constantes, e ficamos novamente com  $O(e)$ .

## 6 Conclusão

Há, possivelmente, a possibilidade de reduzir mais ainda algumas arestas. Esta é apenas uma solução possível, e não necessariamente a melhor.