

Minimização de Valores de Arestas em um Grafo

Pedro Vanzella

17 de novembro de 2015

1 Introdução

Uma recente mudança na regulamentação de impostos reativou uma antiga taxa sobre operações financeiras. Esta taxa, chamada de CPMF, incide em % 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.

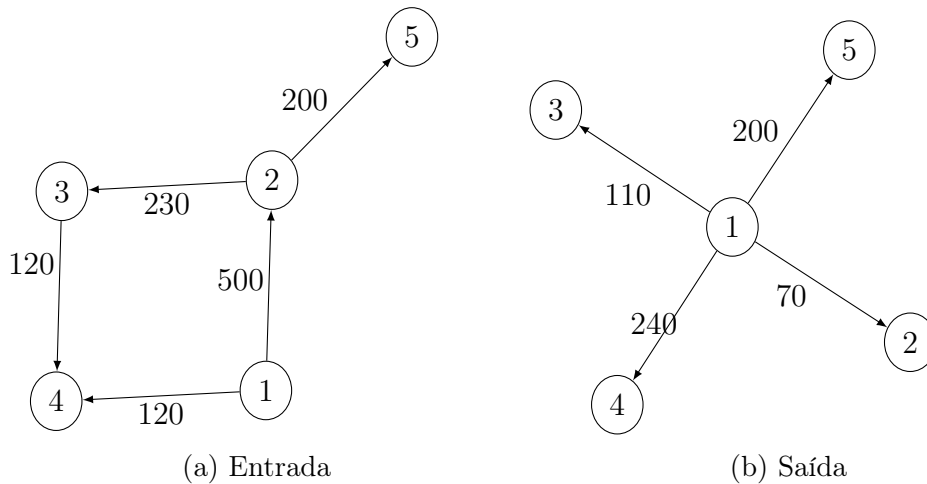


Figura 1: Representação da entrada e da saída como grafos

2 Entrada

O arquivo de entrada é algo 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 Sessão 3, estas informações não serão necessárias.

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

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

Figura 2: Arquivo de entrada

exemplo, na linha 2 da Figura 2, lemos “*Uma transferência de 500 da conta do correntista 1 para a conta do correntista 2.*”.

3 Estrutura de Dados

Inicialmente, pensou-se em utilizar *hashes* um de nodos e um de arestas. O problema com isto é que, ao iterar por um *hash*, não se pode alterar seu tamanho.

Resolveu-se então utilizar-se listas de adjacência, com a estrutura mostrada na Figura 3.

```
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 ???. Veja que ele está na classe **Mardita**, que contém uma instância do grafo.

Algoritmo 1: Criação de Nós 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 na Figura ?? e `add_edge` está descrito na Figura ??.

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 */
     $n$  ← 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.

Caso a implementação tivesse sido feita com um hash, o algoritmo seria como o descrito no Algoritmo 3.

Algoritmo 3: Criação de Nodo, caso a classe seja implementada com um Hash

Classe: Graph

Entrada: val: Inteiro, representando o nome do Nodo

self.nodes[val] = True

Veja que esta versão de `add_node` não precisa verificar a existência do nodo. No entanto, também não há uma classe **Node**, e não retornamos nada. O modo de acesso seria ligeiramente diferente.

Algoritmo 4: Criação de Arestas

Classe: Node

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

Saída: Instancia 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 4 é parecido com o Algoritmo 2, pois ele verifica a unicidade da aresta. A diferença é que arestas são consideradas iguais caso suas origens e destinos sejam iguais, para este problema. Como estamos verificando todas as arestas que partem de um nodo, basta comparar o destino.

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

4 Algoritmo

Há duas coisas a serem feitas para resolver o problema: precisamos calcular quanto imposto é pago (Sessão 4.1) e reduzir o número de arestas do grafo (Sessão 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.

```
1 float Mardita::total_tax_payed():
    total = 0
3     Para todo e em self.graph.edges:
        total += e.valor
5     return total * 0.01
```

Figura 4: Algoritmo de cálculo do total de imposto pago

Na Figura 4 vemos como o total de imposto é calculado. Apenas soma-se o valor de todas as arestas e multiplica-se por 0.01, que é o valor 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 na Figura 5.

```
1 List<Edge> Graph::edges():
    edges = [] // Lista vazia
3     Para todo n em self.nodes:
        para todo e em n.edges:
5         edges.add(e)
    return edges
```

Figura 5: Algoritmo que coleta todas as arestas de todos os nodos

4.2 Redução das Arestas

Este é o algoritmo principal, onde o problema é de fato solucionado. O pseudocódigo pode ser visto na Figura 6.

```

void Mardita::reduce_edges():
2   Para todo nodo u em self.graph.nodes:
    Para cada nodo v adjacente a u:
4       Para cada nodo a adjacente a v:
            se valor(Aresta<v, a>) < valor(Aresta<u, v>):
6                 tmp = valor(Aresta<v, a>)
                 remove Aresta<v, a>
8                 valor(Aresta<u, v>) diminui de tmp

10                se ja existe aresta entre u e a:
                    valor(Aresta<u, a>) aumenta de tmp
12            senao:
                    cria Aresta<u, a> com valor tmp

```

Figura 6: Algoritmo que reduz as arestas

A idéia do algoritmo de Redução de Arestas 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 $y - x$.

Para isto, encontra-se os “amigos dos amigos” de cada nodo no grafo (linhas 2-4 da Figura 6). A verificação da linha 5 é importante para que não criemos uma transação de valor negativo.

Caso a verificação da linha 5 seja positiva, podemos atualizar o valor da primeira aresta (u, v) , subtraindo o valor da aresta (v, a) . Também removemos a aresta (v, a) do grafo.

O próximo passo é criar a aresta (u, a) . Há um porém: a aresta (u, a) pode já existir. Neste caso, apenas soma-se o valor da aresta (v, a) . Caso a aresta não exista, cria-se uma aresta (u, a) com o valor da aresta (v, a) .

Um exemplo mínimo disto pode ser visto na Figura 7.

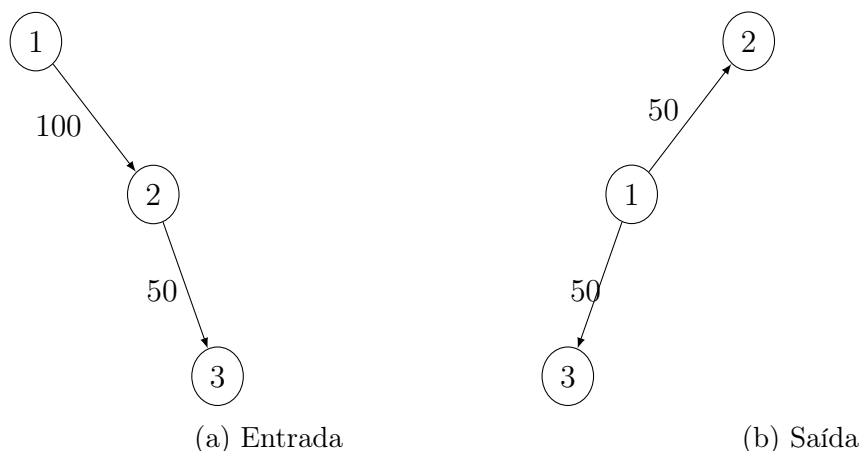


Figura 7: Redução aplicada a um grafo simples.

Pode-se facilmente validar o algoritmo com um teste de mesa na Figura 7, ou mesmo na Figura 1.

Olhando-se para a Figura 7, 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 7a e a Figura 7b foram os valores e os destinos das transferências. Somando-se e multiplicando pelo valor do imposto, vemos que na Figura 7a pagaríamos 1.50 de imposto (*i.e.*, 1% de 150), enquanto na Figura 7b pagaríamos 1.00 de imposto (*i.e.*, 1% de 100).

4.3 Detalhes de Implementação

Note que há um detalhe de implementação no algoritmo da Figura 6: dependendo da linguagem, ao ler os adjacentes de um nodo, teremos uma cópia da lista de adjacentes. Ao inserir uma nova aresta em um nodo, esta lista possivelmente não seria alterada, e é importante analisarmos esta nova aresta também.

Neste caso, deve ser utilizada a versão da Figura 8.

Vemos na Figura 8 algumas diferenças que auxiliam na implementação do algoritmo. Mais especificamente, em vez de simplesmente iterarmos por todos os adjacentes de u , adicionamos todos eles a uma lista (linha 3), e vamos removendo-os um a um, até que a lista esvazie (linha 4). Sempre que uma aresta nova é criada, adicionamos esta aresta à lista **vs** (linha 16).


```

1 void Mardita::reduce_edges():
    Para todo nodo u em self.graph.nodes:
3     vs = adjacentes(u)
    Enquanto vs nao esta vazia
5     v = vs.pop()
    Para cada nodo a adjacente a v:
7     se valor(Aresta<v, a>) < valor(Aresta<u, v>):
        tmp = valor(Aresta<v, a>)
9     remove Aresta<v, a>
        valor(Aresta<u, v>) diminui de tmp
11
        se ja existe aresta entre u e a:
13            valor(Aresta<u, a>) aumenta de tmp
        senao:
15            cria Aresta<u, a> com valor tmp
            vs.add(a)

```

Figura 8: Algoritmo que reduz as arestas, versão com uma lista auxiliar

Esta versão do algoritmo também resolveria o problema apresentado na Sessão 3 com *hashes*.

5 Resultados

Teste	Economia	Transações	Tempo de Execução
1	3980.31	264	0.17s
2	3329.03	235	0.17s
3	1905.99	165	0.08s
4	1545.48	140	0.06s
5	925.53	99	0.06s
6	2288.34	186	0.10s
7	609.39	91	0.05s
8	3029.33	232	0.14s
9	1991.52	165	0.09s
10	1912.52	153	0.07s

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.