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

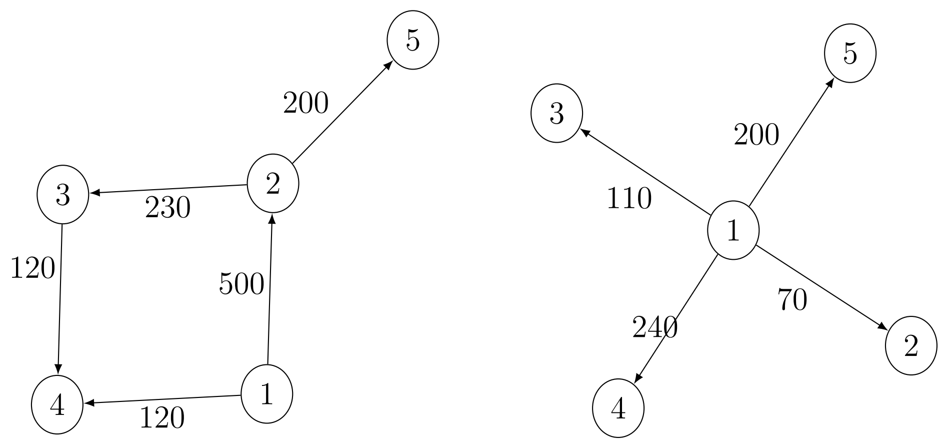
1 transfere $110 para 3

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.



(a) Entrada

(b) Saída

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:
    public list<Node> nodes

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

    class Edge:
        public Node from
        public Node to
        public int val

```

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 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 na Figura **??** e `add_edge` está descrito na Figura **??**.

Algoritmo 2: Criação de Nodos

```
Classe: Graph
Entrada: val: Inteiro, representando o nome do Nodo
Saída: Instância da classe Node
para todo n em self.nodes faça
    se n.val = val então
        retorna n ;     /* Se o nodo já existe, retorna ele */
    n ← Graph.Node(val) ;     /* Chama o construtor de Node */
    self.nodes.add(n) ;         /* Adiciona à lista de nodos */
    retorna n
```

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

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

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 ← 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.

Algoritmo 5: Cálculo de Imposto Pago

Classe: Mardita
Entrada: Todas as arestas do grafo
Saída: Total de imposto pago
total ← 0
para todo *e em self.graph.edges* **faça**
  total ← total + e.valor
retorna *total* × 0.01

No Algoritmo 5 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 no Algoritmo 6.

Algoritmo 6: Coleção de todas as arestas

Classe: Node
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 na Figura 4.

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 $e =$

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

Figura 4: Algoritmo que reduz as arestas

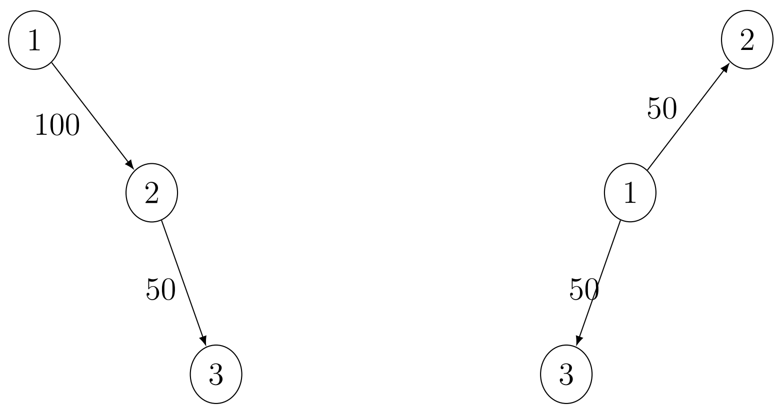
$(A, B, x), (B, C, y)$, gerar uma nova aresta $(A, C, x - y)$, remover a aresta (B, C, x) e alterar o valor da aresta (A, B) para $y - x$.

Para isto, encontra-se os “amigos dos amigos” de cada nodo no grafo (linhas 2-4 da Figura 4). 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 5.



(a) Entrada

(b) Saída

Figura 5: Redução aplicada a um grafo simples.

Pode-se facilmente validar o algoritmo com um teste de mesa na Figura 5, 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 impost (*i.e.*, 1% de 100).

4.3 Detalhes de Implementação

Note que há um detalhe de implementação no algoritmo da Figura 4: 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 6.

Vemos na Figura 6 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).

```
void Mardita::reduce_edges():
    Para todo nodo u em self.graph.nodes:
        vs = adjacentes(u)
        Enquanto vs nao esta vazia
            v = vs.pop()
            Para cada nodo a adjacente a v:
                se valor(Aresta<v, a>) < valor(Aresta<u, v>):
                    tmp = valor(Aresta<v, a>)
                    remove Aresta<v, a>
                    valor(Aresta<u, v>) diminui de tmp

                    se ja existe aresta entre u e a:
                        valor(Aresta<u, a>) aumenta de tmp
                    senao:
                        cria Aresta<u, a> com valor tmp
                        vs.add(a)
```

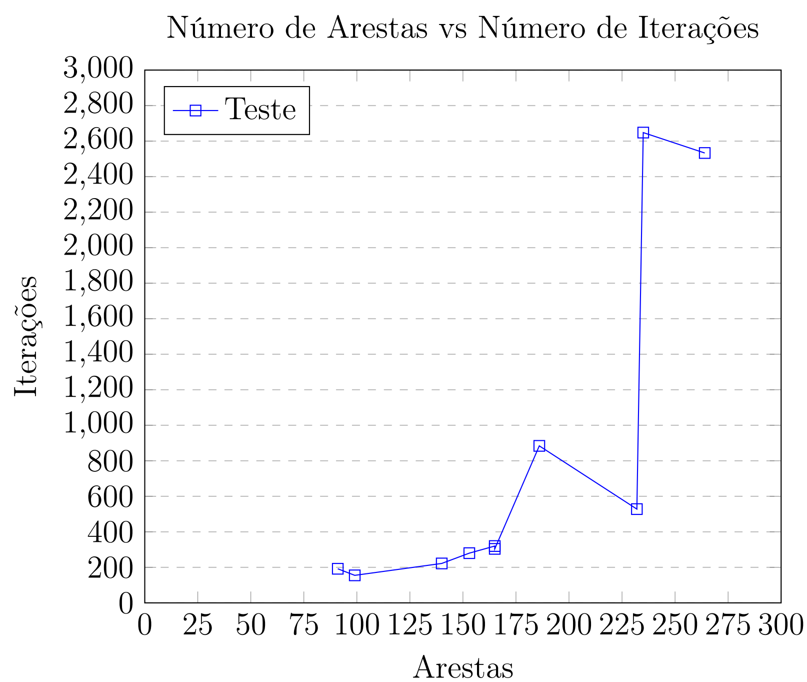
Figura 6: Algoritmo que reduz as arestas, versão com uma lista auxiliar

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

5 Resultados

Teste	Economia	Transações	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 despresa 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.