Minimização de Valores de Arestas em um Grafo

Pedro Vanzella

14 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

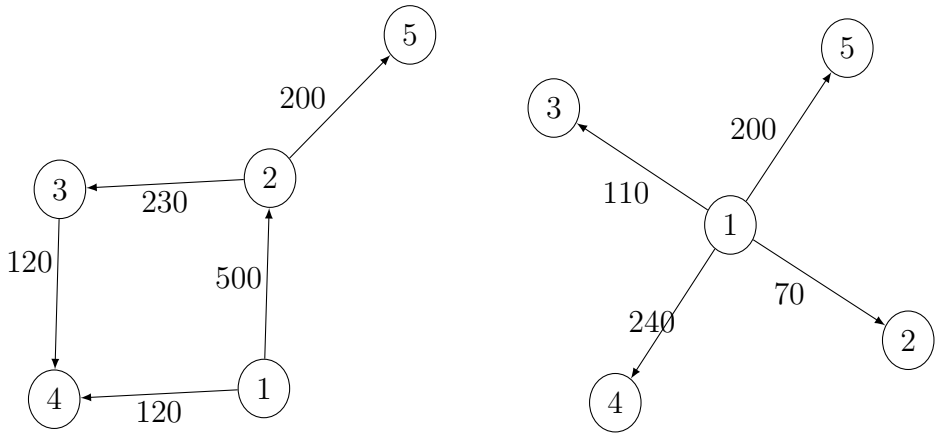
1 transfere $110 para 3

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

```
1  5 5
2  1 2 500
3  2 3 230
4  3 4 120
5  1 4 120
6  2 5 200
7
```

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 da Figura 4. Veja que ele está na classe **Mardita**, que contém uma instância do grafo.

Onde add_node está descrito na Figura 5 e add_edge está descrito na Figura 7.

O algoritmo add_node (Figura 5), 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

```
void Mardita::read_file(arquivo)
    Para cada linha l no arquivo, exceto a primeira:
        partes = l.separa(' ')

        nodo_a = self.graph.add_node(partes[0])
        nodo_b = self.graph.add_node(partes[1])

        nodo_a.add_edge(nodo_b, partes[2])
```

Figura 4: Algoritmo de criação de nodos e arestas

```
Node Graph::add_node(val):
    para todo n em self.nodes:
        se n.val == val:
            return n
    n = Graph.Node(val)
    self.nodes.add(n)
    return n
```

Figura 5: Algoritmo de criação de nodos e arestas

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 na Figura 6.

```
void Graph::add_node(val):
    self.nodes[val] = True
```

Figura 6: Algoritmo de criação de nodos e arestas

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.

O algoritmo add_edge (Figura 7) é parecido com o algoritmo add_node

```
Edge Node::add_edge(to, val):
    para cada e em self.edges:
        se e.to == to:
            return e

    e = Graph.Edge(self, to, val)
    self.edges.add(e)
    return e
```

Figura 7: Algoritmo de criação de nodos e arestas

(Figura 5), 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.

Na Figura 8 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 9.

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

Figura 8: Algoritmo de cálculo do total de imposto pago

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

Figura 9: 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 10.

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 10). 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.

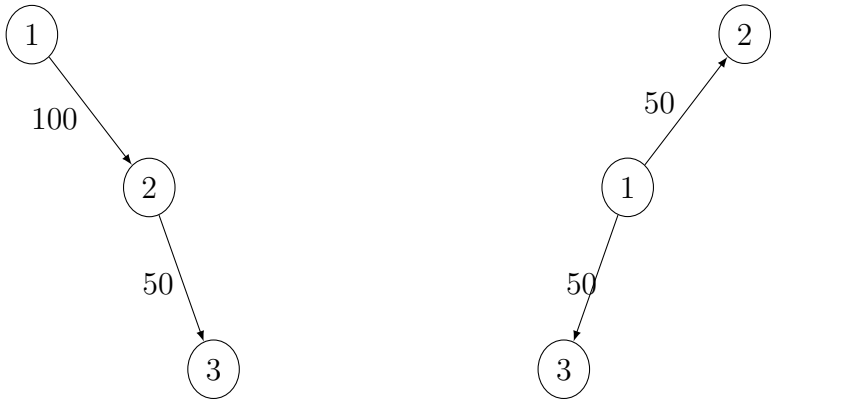
O próximo passo é criar a aresta (u, a). Há um porém: a aresta (u, a) pode já existir. Neste caso, apenas soma-se o valor da aresta (v, a). Caso a aresta não exista, cria-se uma aresta (u, a) com o valor da aresta (v, a).

Um exemplo mínimo disto pode ser visto na Figura 11.

```
void Mardita::reduce_edges():
    Para todo nodo u em self.graph.nodes:
        Para cada nodo v adjacente a u:
            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
```

Figura 10: Algoritmo que reduz as arestas



(a) Entrada (b) Saída

Figura 11: Redução aplicada a um grafo simples.

Pode-se facilmente validar o algoritmo com um teste de mesa na Figura 11, ou mesmo na Figura 1.

Olhando-se para a Figura 11, 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 11a e a Figura 11b foram os valores e os destinos das transferências. Somando-se e multiplicando pelo valor do imposto, vemos

que na Figura 11a pagaríamos 1.50 de imposto (*i.e.*, 1% de 150), enquanto na Figura 11b pagaríamos 1.00 de impost (*i.e.*, 1% de 100).

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