Minimização de Valores de Arestas em um Grafo

Pedro Vanzella

13 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

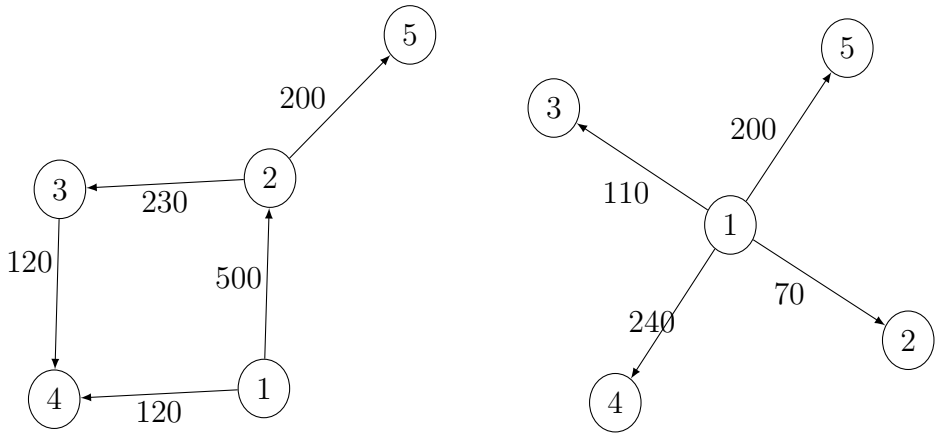
1 transfere $110 para 3

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

5 Resultados