Exercício 7.28 (Tardos)

Alice Duarte Scarpa, Bruno Lucian Costa

2015-06-23

1 Enunciado

Um grupo de estudantes está escrevendo um módulo para preparar cronogramas de monitoria. O protótipo inicial deles funciona do seguinte modo: O cronograma é semanal, de modo que podemos nos focar em uma única semana.

- O administrador do curso escolhe um conjunto de k intervalos disjuntos de uma hora de duração $I_1, I_2, \ldots, I_k$, nos quais seria possível que monitores dessem suas monitorias; o cronograma final consistirá de um subconjunto de alguns (mas geralmente não todos) esses intervalos.
- Cada monitor então entra com seu horário semanal, informando as horas em que ele está disponível para monitorias.
- O administrador então especifica, para parâmetros a, b e c, que cada monitor deve dar entre a e b horas de monitoria por semana, e que um total de c horas de monitoria deve ser dado semanalmente.

O problema é escolher um subconjunto dos horários (intervalos) e atribuir um monitor a cada um desses horários, respeitando a disponibilidade dos monitores e as restrições impostas pelo administrador.

a) Dê um algoritmo polinomial que ou constrói um cronograma válido de horas de monitoria (especificando que monitor cobre quais horários) ou informa que não há cronograma válido.

b) O algoritmo acima tornou-se popular, e surgiu a vontade de controlar também a densidade das monitorias: dado números d_i, com i entre 1 e 5, queremos um cronograma com pelo menos d_i horários de monitoria no dia da semana i. Dê um algoritmo polinomial para resolver o problema com essa restrição adicional.

2 Solução força-bruta

2.1 Algoritmo

2.2 Implementação

2.3 Complexidade

3 Solução usando fluxo

3.1 Introdução

Queremos modelar esse problema como um problema de fluxo. Para isso vamos começar com algumas definições de fluxo.

3.1.1 Definições

Uma rede de fluxo é um grafo direcionado $G = (V, E)$ com as seguintes propriedades:

- Existe um único vértice *fonte* $s \in V$. Nenhuma aresta entra em s.
- A cada aresta e está associada uma capacidade inteira c_e e uma demanda d_e tal que $c_e \geq d_e \geq 0$.
- Existe um único vértice *dreno* $t \in V$. Nenhuma aresta sai de t.

Um fluxo f de s a t é uma função $f\colon E \to R^+$ que associa a cada aresta e um valor real não-negativo $f(e)$ tal que:

1. $\forall e \in E, d_e \leq f(e) \leq c_e$
2. Para todo nó $v \notin \{s, t\}$:

$$\sum_{e \text{ chegando em } v} f(e) = \sum_{e \text{ saindo de } v} f(e)$$

$f(e)$ representa o fluxo que vai passar pela aresta e. O valor de um fluxo é o total que parte da fonte s, isso é:

$$\text{Valor}(f) = \sum_{e \text{ saindo de } s} f(e)$$

3.1.2 Representação

Podemos usar programação orientada a objetos para nos ajudar na representação da rede de fluxo, simplificando o algoritmo. TODO: explicar a parte de já construir o grafo reverso.

Vamos usar uma classe para representar arestas. Uma aresta tem as propriedades: vértice de origem, vértice de destino, capacidade (e reversa e fluxo, talvez)

```
class Aresta():
    def __init__(self, origem, destino, capacidade, demanda):
        self.origem = origem
        self.destino = destino
        self.capacidade = capacidade
        self.demanda = demanda
        self.reversa = None
```

Agora que temos a classe Aresta, vamos usá-la para auxiliar na representação de uma rede de fluxo também como objeto.

Uma rede de fluxo tem duas propriedades: adjacências, um dicionário que mapeia cada vértice às arestas que saem dele e fluxo, representando as arestas do grafo e fluxo, representando... O construtor da classe inicializa as duas propriedades como dicionários vazios.

Vamos precisar dos seguintes métodos na nossa classe RedeDeFluxo:

- `novo_vertice(v)`: Adiciona o vértice v à rede
- `nova_aresta(origem, destino, capacidade)`: Adiciona uma nova aresta a rede. Também cria a aresta reversa.
- `novo_fluxo(f, e)`: Adiciona um fluxo f à aresta e
- `encontra_arestas(v)`: Retorna as arestas que partem do vértice v
- `valor_do_fluxo(fonte)`: Encontra o valor do fluxo, como definido em (3.1.1).

```
class RedeDeFluxo():
    def __init__(self):
        self.adj = {}
        self.fluxo = {}
```

```python
def novo_vertice(self, v):
    self.adj[v] = []

def nova_aresta(self, origem, destino, capacidade, demanda):
    aresta = Aresta(origem, destino, capacidade, demanda)
    self.adj[origem].append(aresta)

    # Criando a aresta reversa
    aresta_reversa = Aresta(destino, origem, 0, -1*demanda)
    self.adj[destino].append(aresta_reversa)

    # Marcando aresta e aresta_reversa como reversas uma da outra
    aresta.reversa = aresta_reversa
    aresta_reversa.reversa = aresta

def novo_fluxo(self, e, f):
    self.fluxo[e] = f

def encontra_arestas(self, v):
    return self.adj[v]

def valor_do_fluxo(self, fonte):
    valor = 0
    for aresta in self.encontra_arestas(fonte):
        valor += self.fluxo[aresta]
    return valor
```

3.2 Modelando o problema com fluxos

Os dois itens do problema podem ser reduzidos a encontrar um fluxo válido em uma rede usando construções semelhantes.

Para o item a), construimos o grafo da seguinte forma:

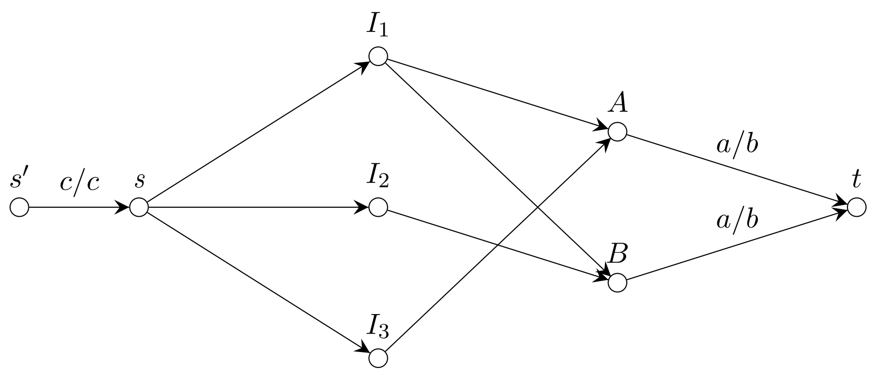
- Criamos um vértice s representando a fonte e um vértice t representando o dreno
- Para cada intervalo $I_i \in I_1, I_2, \ldots, I_k$ escolhido pelo administrador, criamos um vértice I_i e uma aresta (s, I_i) capacidade 1 e demanda 0
- Para cada monitor $T_i \in T_1, T_2, \ldots, T_m$ criamos um vértice T_i. Se o monitor está disponível para dar monitoria no intervalo I_j criamos

uma aresta de (I_j, T_i) de demanda 0 e capacidade 1. Para cada monitor também criamos uma aresta (T_i, t) de demanda a e capacidade b.

- Para garantir que a solução final terá exatamente c horas de monitoria, criamos uma nova fonte s' e uma aresta (s', s) com demanda e capacidade c.

TODO: argumentar que soluções para esse problema são equivalentes a soluções do problema original

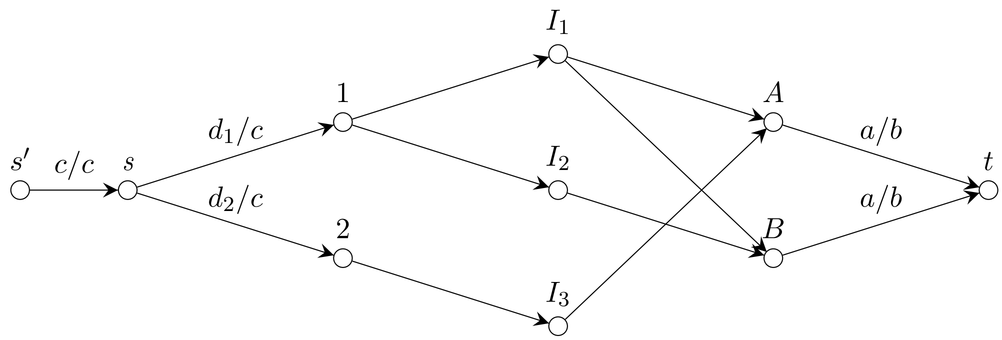
O caso com 3 intervalos e 2 monitores (A e B) em que o monitor A está disponível nos intervalos 1 e 2 e o monitor B está disponível nos horários 1 e 3 está representado abaixo. Os rótulos das arestas são da forma demanda/capacidade. As arestas sem rótulo tem demanda 0 e capacidade 1.



A única diferença na construção do item b é que, ao invés de ligarmos s diretamente aos intervalos de monitoria, ligamos s a cada dia da semana i com demanda d_i e capacidade c e depois criamos uma aresta com demanda 0 e capacidade 1 de cada dia da semana para os intervalos que são naquele dia.

TODO: argumento que isso dá a solução certa

Abaixo está o mesmo exemplo do item a) com dias da semana. Para deixar a visualização mais simples estamos colocando aqui apenas dois dias da semana.



3.3 Implementação

3.3.1 Fluxo máximo

Vamos começar estudando o problema de encontrar o fluxo máximo de uma rede G em que $d_e = 0\ \forall e \in E\ f$. Vamos implementar aqui o algoritmo de Ford-Fulkerson para resolver esse problema.

O algoritmo tem 2 partes:

1. Dado um caminho P e partindo de um fluxo inicial f, obter um novo fluxo f' expandindo f em P
2. Partindo do fluxo $f(e) = 0$, expandir o fluxo enquanto for possível

- Primeira parte:

O gargalo de um caminho é … Definimos aqui uma função que encontra o gargalo do caminho

```
def encontra_gargalo(self, caminho):
    residuos = []
    for aresta in caminho:
        residuos.append(aresta.capacidade - self.fluxo[aresta])
    return min(residuos)
```

Expandir o caminho é …

```
def expande_caminho(self, caminho):
    gargalo = self.encontra_gargalo(caminho)
    for aresta in caminho:
        self.fluxo[aresta] += gargalo
        self.fluxo[aresta.reversa] -= gargalo
```

Com isso temos a parte 1 do algoritmo.

Para a parte 2, vamos precisar criar um fluxo f com $f(e) = 0$ para toda aresta e. Podemos fazer isso utilizando o seguinte método na classe RedeDeFluxo():

```
def cria_fluxo_inicial(self):
    for vertice, arestas in self.adj.iteritems():
        for aresta in arestas:
            fluxo[aresta] = 0
```

Retorna um caminho de fonte a dreno passando pelos vértices em caminho

```
def encontra_caminho(self, fonte, dreno, caminho):
    if fonte == dreno:
        return caminho
    for aresta in self.encontra_arestas(fonte):
        residuo = aresta.capacidade - self.fluxo[aresta]
        if residuo > 0 and aresta not in caminho:
            resp = self.encontra_caminho(aresta.destino, dreno, caminho + [aresta])
            # TODO: explicar essa parte
            if resp != None:
                return resp
```

Com todas as funções auxiliares prontas, podemos finalmente definir a função que encontra o fluxo máximo.

```
def fluxo_maximo(self, fonte, dreno):
    self.cria_fluxo_inicial()
    caminho = self.encontra_caminho(fonte, dreno, [])
    while caminho is not None:
        self.expande_caminho(caminho)
        caminho = self.encontra_caminho(fonte, dreno, [])
    return self.valor_do_fluxo(fonte)
```

3.3.2 Fluxo válido com demandas não-nulas

O nosso objetivo é encontrar um fluxo válido f para uma rede $G = (V, E)$ no caso em que as demandas são positivas.

Vamos construir uma rede $G' = (V', E')$ com um valor associado d tal que $d_e = 0\ \forall e \in E'$ de tal forma que um fluxo válido para G existe se e somente se o valor do fluxo máximo em G' é d. Em caso afirmativo, podemos construir um fluxo válido f para G rapidamente a partir de qualquer fluxo máximo f' de G'.

Construimos G' da seguinte forma:

- Criamos um vértice em G' para cada vértice G
- Adicionamos uma fonte adicional F e um dreno adicional D a G'
- Definimos o saldo de cada vértice $v \in V$ como:

$$\text{saldo}(v) = \sum_{e \text{ saindo de } v} d_e - \sum_{e \text{ chegando em } v} d_e$$

- Se saldo(v) > 0 adicionamos uma aresta $(v, D, \text{saldo}(v), 0)$ a G'
- Se saldo(v) < 0 adicionamos uma aresta $(F, v, -\text{saldo}(v), 0)$ a G'
- Para cada aresta $e = (\text{origem}, \text{destino}, \text{capacidade}, \text{demanda}) \in E$, crie uma aresta $e' = (\text{origem}, \text{destino}, \text{capacidade} - \text{demanda}, 0)$ em G'

3.4 Complexidade