Trabalho de Estruturas de Dados e Algoritmos

Alice Duarte Scarpa, Bruno Lucian Costa

2015-06-23

1 Exercício 7.28 (Tardos)

1.1 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.

1.2 Introdução

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

1.2.1 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:

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

TODO: definir circulação

1.2.2 Representação

Podemos usar [[TODO: colocar uma referencia de OOP][programação orientada a objetos] para nos ajudar na representação da rede de fluxo, simplificando o algoritmo.

Vamos usar uma classe para representar arestas. Uma aresta é inicializada com as propriedades: vértice de origem, vértice de destino, capacidade e demanda.

Para facilitar o processamento futuro, vamos adicionar também as propriedades reversa e original. Reversa aponta para uma outra aresta reversa à atual, a propriedade original é uma flag indicando se a aresta pertence à rede original ou não.

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

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 TODO: explicar isso

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 (1.2.1).

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

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

```python
    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, -demanda)
        self.adj[destino].append(aresta_reversa)
        aresta_reversa.original = False

        # 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
```

1.3 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

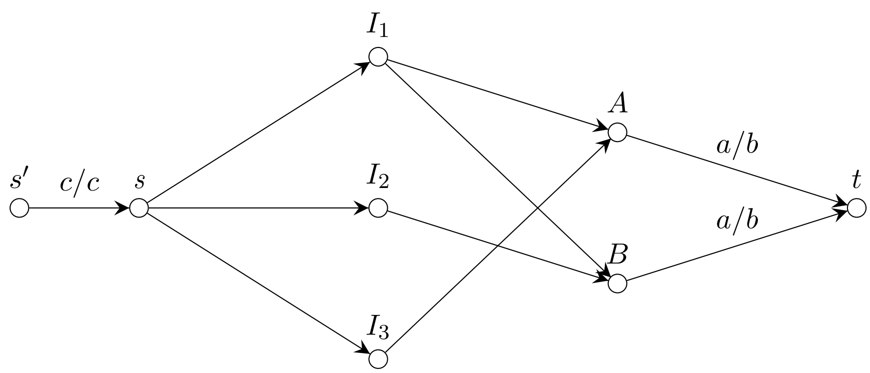
uma aresta de (I_j, T_i) de demanda 0 e capacidade 1. Para cada monitor também criamos uma aresta (T_i, t) de demanda a e capacidade b.

- 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.

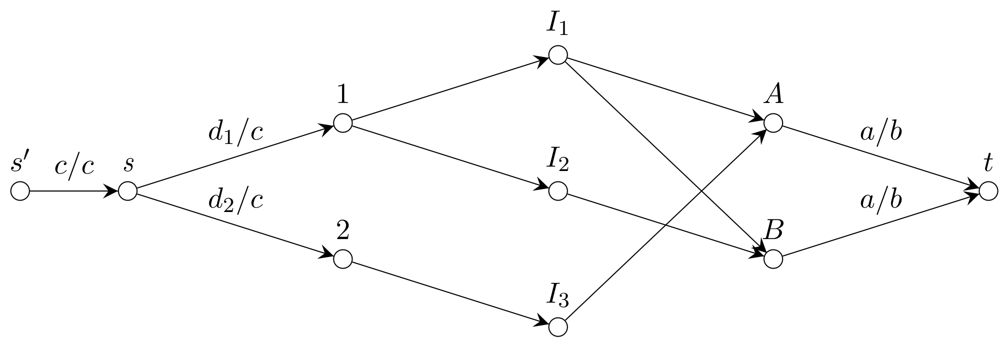
TODO: circulação



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.



1.4 Implementação

1.4.1 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 TODO: definir gargalo, explicar o código a seguir 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 TODO: explicar o que é expandir o caminho,

```
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:
            self.fluxo[aresta] = 0
```

```
None
```

TODO: explicar porque precisamos desse método e como ele funciona Retorna um caminho de fonte a dreno passando pelos vértices em caminho É uma DFS

```
def encontra_caminho(self, fonte, dreno, caminho, visitados):
    if fonte == dreno:
        return caminho

    visitados.add(fonte)

    for aresta in self.encontra_arestas(fonte):
        residuo = aresta.capacidade - self.fluxo[aresta]
        if residuo > 0 and aresta.destino not in visitados:
            resp = self.encontra_caminho(aresta.destino,
                                         dreno,
                                         caminho + [aresta],
                                         visitados)
            # 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.

TODO: explicar o algoritmo de fluxo máximo

```
def fluxo_maximo(self, fonte, dreno):
    self.cria_fluxo_inicial()

    caminho = self.encontra_caminho(fonte, dreno, [], set())
    while caminho is not None:
```

```
        self.expande_caminho(caminho)
        caminho = self.encontra_caminho(fonte, dreno, [], set())
    return self.valor_do_fluxo(fonte)
```

1.4.2 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:

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

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

Codificando a construção acima:

```
def cria_rede_com_demandas_nulas(G):
    G_ = RedeDeFluxo()
    G_.novo_vertice('F')
    G_.novo_vertice('D')
    d = 0

    for vertice, arestas in G.adj.iteritems():
        G_.novo_vertice(vertice)
        saldo = sum(e.demanda for e in arestas)
```

```
        if saldo > 0:
            G_.nova_aresta(vertice, 'D', saldo, 0)
            d += saldo
        elif saldo < 0:
            G_.nova_aresta('F', vertice, -saldo, 0)

    for arestas in G.adj.values():
        for a in arestas:
            if a.original:
                G_.nova_aresta(a.origem,
                               a.destino,
                               a.capacidade - a.demanda,
                               0)
    return G_, d
```

TODO: provar que soluções de um são também soluções do outro

1.5 Complexidade

TODO: calcular a complexidade do algoritmo

1.6 Rodando o algoritmo

1.6.1 Item A

A seguinte tabela mostra a disponibilidade dos monitores nos horários escolhidos pelo administrador:

	Ana	Bia	Caio	Davi	Edu	Felipe	Gabi	Hugo	Isa
Seg 10h				x					
Seg 14h						x	x	x	x
Seg 21h	x			x					
Ter 10h	x	x		x					
Ter 16h			x						
Ter 20h							x		x
Qua 9h						x			
Qua 17h			x						
Qua 19h								x	
Qui 7h		x				x			
Qui 13h							x		
Qui 19h		x			x			x	
Sex 7h			x		x				
Sex 11h	x				x				x
Sex 21h			x			x			x

As outras regras para monitoria estão na tabela abaixo:

Min de horas por monitor	1
Max de horas por monitor	3
Horas de monitoria	10

Podemos carregar as informações das tabelas para criar uma rede como descrita em TODO: colocar a referencia certa.

```
# Lendo a tabela de disponibilidade
intervalos = collections.OrderedDict()
monitores = horarios[0][1:]

for disponibilidade in horarios[1:]:
    intervalos[disponibilidade[0]] = []
    for i, slot in enumerate(disponibilidade[1:]):
        if slot != '':
            intervalos[disponibilidade[0]].append(monitores[i])
```

Lendo a tabela de regras

```
min_horas = regras[0][1]
max_horas = regras[1][1]
total_horas = regras[2][1]
```

Criando uma rede para o problema com os dados fornecidos

```python
def cria_rede(intervalos, monitores, min_horas, max_horas, total_horas):
    G = RedeDeFluxo()
    G.novo_vertice('Fonte')
    G.novo_vertice('Dreno')
    G.nova_aresta('Dreno', 'Fonte', total_horas, total_horas)

    # Criando um vertice para cada monitor e ligando esse vertice ao dreno
    for monitor in monitores:
        G.novo_vertice(monitor)
        G.nova_aresta(monitor, 'Dreno', max_horas, min_horas)

    for intervalo, monitores_disponiveis in intervalos.iteritems():
        # Criando um vertice para cada intervalo e conectando a fonte a
        # cada um dos intervalos
        G.novo_vertice(intervalo)
        G.nova_aresta('Fonte', intervalo, 1, 0)

        # Conectando o intervalo a cada monitor disponivel nele
        for monitor in monitores_disponiveis:
            G.nova_aresta(intervalo, monitor, 1, 0)

    return G
```

Agora é só rodar o algoritmo com o grafo obtido:

```python
G = cria_rede(intervalos, monitores, min_horas, max_horas, total_horas)
G_, d = cria_rede_com_demandas_nulas(G)
fluxo = G_.fluxo_maximo('F', 'D')
if fluxo == d:
    tabela_de_monitores = []
    for horario in intervalos:
        for w in G_.adj[horario]:
            if G_.fluxo[w] == 1:
                tabela_de_monitores.append([w.origem, w.destino])
    return tabela_de_monitores
else:
    return 'Impossivel'
```

No final, obtemos ou 'Impossível' se não existir um horário compatível ou uma tabela com um horário que atende a todas as restrições.

Para a tabela acima:

1.6.2 Item b

No item b, além de todas as restrições do item a, há também a restrição de mínimo de horas por dia da semana.

Vamos expressar a nova restrição com uma tabela:

Seg	1
Ter	1
Qua	2
Qui	1
Sex	1

Parsear a nova tabela é simples:

```
minimo_por_dia = {}
for dia in min_por_dia:
    minimo_por_dia[dia[0]] = dia[1]
```

A única função que precisamos alterar do item a é a função `cria_rede`, que agora tem que lidar com a construção mencionada em TODO.

```
def cria_rede(intervalos, monitores, min_horas,
              max_horas, total_horas, minimo_por_dia):
    G = RedeDeFluxo()
    G.novo_vertice('Fonte')
    G.novo_vertice('Dreno')
    G.nova_aresta('Dreno', 'Fonte', total_horas, total_horas)

    # Criando um vertice para cada monitor e ligando esse vertice ao dreno
    for monitor in monitores:
        G.novo_vertice(monitor)
        G.nova_aresta(monitor, 'Dreno', max_horas, min_horas)

    # Criando um vertice para cada dia e uma aresta da Fonte ao dia
    # com demanda igual ao minimo de horas de monitoria para aquele dia
    # e capacidade suficientemente grande (vamos usar o total de horas)
    dias = minimo_por_dia.keys()
    for dia in dias:
        G.novo_vertice(dia)
        G.nova_aresta('Fonte', dia, total_horas, minimo_por_dia[dia])

    for intervalo, monitores_disponiveis in intervalos.iteritems():
```

```
        # Encontrando o dia do intervalo
        for dia in dias:
            if intervalo.startswith(dia):
                dia_do_intervalo = dia

        # Criando um vertice para cada intervalo e conectando o dia do intervalo
        # a cada um dos intervalos
        G.novo_vertice(intervalo)
        G.nova_aresta(dia_do_intervalo, intervalo, 1, 0)

        # Conectando o intervalo a cada monitor disponivel nele
        for monitor in monitores_disponiveis:
            G.nova_aresta(intervalo, monitor, 1, 0)

    return G
```

Seg 10h	Davi
Seg 14h	Isa
Seg 21h	Ana
Ter 10h	Bia
Ter 16h	Caio
Qua 9h	Felipe
Qua 17h	Caio
Qua 19h	Hugo
Qui 13h	Gabi
Sex 7h	Edu

2 Exercício 6.30 (Papadimitriou)

2.1 Enunciado

Reconstruindo árvores filogenéticas pelo método da máxima parcimônia

Uma árvore filogenética é uma árvore em que as folhas são espécies diferentes, cuja raiz é o ancestral comum de tais espécies e cujos galhos representam eventos de especiação.

Queremos achar:

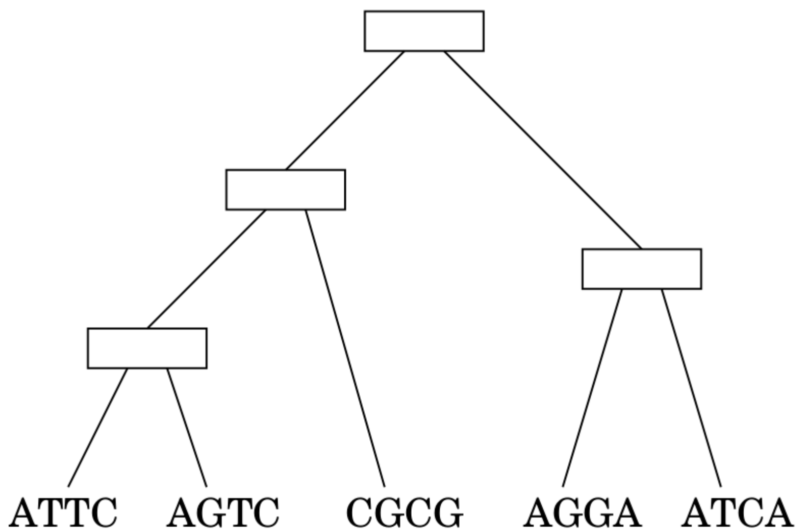
- Uma árvore (binária) evolucionária com as espécies dadas
- Para cada nó interno uma string de comprimento k com a sequência genética daquele ancestral.

Dada uma árvore acompanhada de uma string $s(u) \in \{A, C, G, T\}^k$ para cada nó $u \in V(T)$, podemos atribuir uma nota usando o método da máxima parcimônia, que diz que menos mutações são mais prováveis:

$$\text{nota}(T) = \sum_{(u,v) \in E(T)} (\text{número de posições em que } s(u) \text{ e } s(v) \text{ diferem}).$$

Achar a árvore com nota mais baixa é um problema difícil. Aqui vamos considerar um problema menor: Dada a estrutura da árvore, achar as sequências genéticas $s(u)$ para os nós internos que dêem a nota mais baixa.

Um exemplo com $k = 4$ e $n = 5$:



1. Ache uma reconstrução para o exemplo seguindo o método da máxima parcimônia.
2. Dê um algoritmo eficiente para essa tarefa.

2.2 Solução

A nota final de uma árvore é a soma da nota de cada letra. Podemos calcular a resposta para cada letra independentemente e depois concatenar as respostas para obter a árvore final.

Nós vamos usar um algoritmo de programação dinâmica para encontrar o valor das folhas intermediárias em uma árvore P em que cada folha tem valor A, G, T ou C.

Vamos representar a nossa árvore como um objeto:

```python
class Arvore:
    def __init__(self, pai):
        self.filhos = []
        self.valor = ""
        self.pai = pai
```

Vamos computar $melhor\_nota[v, \ell]$ como a melhor maneira de preencher os nós da sub-árvore enraizada em v, dado que o pai de v tem valor ℓ. Também preencheremos $melhor\_letra[v, \ell]$ com um valor possível de uma configuração otimal. Guardaremos tais valores em dicionários.

```python
melhor_nota = {}
melhor_letra = {}
```

Vamos computar $melhor\_nota$ de baixo para cima. Então, o caso base para esse algoritmo é a resposta para as folhas, isto é, $melhor\_nota[\text{folha}][\ell]$.

Uma sub-árvore que contém apenas uma folha e seu pai vai ter nota 0 se a folha e o pai tiverem ambos o mesmo valor (A, G, T ou C) ou nota 1 se os dois tiverem valores diferentes:

$$melhor\_nota[folha][\ell] = \begin{cases} 0 \text{ se } folha.valor = \ell \\ 1 \text{ caso contrário} \end{cases}$$

Além disso, não temos escolha para o valor otimal:

$$melhor\_valor[folha][\ell] = folha.valor$$

Tendo o caso base, podemos computar $melhor\_nota[v][\ell]$ assumindo que $melhor\_nota[w][\ell]$ já foi computado para todo w filho de v e $\ell \in \{A, G, T, C\}$.

Dado que o pai de v tem valor ℓ, a melhor nota para a sub-árvore enraizada em v quando o valor de v é igual a m é:

$$[\ell \neq m] + \sum_{w \text{ filho de } v} melhor\_nota[w][m]$$

Onde

$$[\ell \neq m] = \begin{cases} 0 \text{ se } m = \ell \\ 1 \text{ caso contrário} \end{cases}$$

Queremos escolher um valor $m \in \{A, G, T, C\}$ para v que minimize a nota final da sub-árvore. Então:

$$melhor\_nota[v][\ell] = \min_{m \in \{A,G,T,C\}} \left([\ell \neq m] + \sum_{w \text{ filho de } v} melhor\_nota[w][m] \right)$$

e $melhor\_letra[v][\ell]$ é um valor de m que atinge o mínimo acima.

Implementando o que descrevemos recursivamente, obtemos a seguinte função:

```
def calcula_melhor_nota(v, l):
    if not v.filhos:
        melhor_nota[v, l] = 1 if l != v.valor else 0
        melhor_letra[v, l] = v.valor
        return melhor_nota[v, l]

    melhor_nota[v, l] = 100000

    for m in ['A', 'G', 'T', 'C']:
        nota_atual = sum(calcula_melhor_nota(w, m) for w in v.filhos)
        if m != l:
            nota_atual += 1

        if nota_atual < melhor_nota[v, l]:
            melhor_nota[v, l] = nota_atual
            melhor_letra[v, l] = m

    return melhor_nota[v, l]
```

Sabendo calcular $melhor\_nota[v][\ell]$ para todos os vértices exceto a raiz podemos encontrar a nota da árvore como o mínimo entre os possíveis valores para a raiz:

$$\min_{\ell \in \{A,G,T,C\}} \sum_{v \text{ filho da raiz}} melhor\_nota[v][\ell]$$

Um valor ótimo para a raiz é um valor de ℓ para o qual o mínimo acima é atingido. Preencheremos *raiz.label* com tal valor, como abaixo:

```
melhor_nota_raiz = 100000
for l in ['A', 'G', 'T', 'C']:
    nota_atual_raiz = sum(calcula_melhor_nota(w, m) for w in v.filhos)
```

```
    if nota_atual_raiz < melhor_nota_raiz:
        raiz.valor = l
        melhor_custo_raiz = nota_atual_raiz
```

Tendo o valor da raiz já determinado e os valores de *melhor_letra*, podemos preencher os valores dos nós internos:

```
def preenche_dado_pai(v):
    v.valor = melhor_letra[v, v.pai.valor]
    for w in v.filhos:
        preenche_dado_pai(w)

for w in raiz.filhos:
    preenche_dado_pai(w)
```

2.3 Rodando o algoritmo

2.3.1 Formato Newick

Um formato muito usado para árvores em bioinformática é o formato Newick. Assim como as *s*-expressions do LISP, ele usa o fato de que parênteses podem ser usados para especificar uma árvore.

TODO: especificar o formato, referência do formato

1. Parseando o formato Newick

   O primeiro passo é notar que (gato, rato) é equivalente a (gato)(rato), então podemos transformar uma estrutura com vírgulas em uma estrutura que só contém parênteses.

   TODO: explicar o código

   ```
   def parseia_newick(string):
       string = string.replace(',', ')(').replace(';', '')

       em_construcao = collections.deque()
       em_construcao.append(Arvore(None))

       for ch in string:
           if ch == '(':
               pai_atual = em_construcao[-1]
               filho_novo = Arvore(pai_atual)
               pai_atual.filhos.append(filho_novo)
   ```

```
            em_construcao.append(filho_novo)
        elif ch == ')':
            em_construcao.pop()
        else:
            em_construcao[-1].valor += ch

    assert len(em_construcao) == 1
    return em_construcao[0]
```

2. Separando e concatenando árvores

   As árvores no nosso algoritmo só tem uma letra por nó, mas nós recebemos apenas uma árvore com toda a string de DNA.

   Precisamos de um método para capaz de criar uma árvore para cada carácter. A seguinte DFS cria a árvore das i-ésimas letras:

```
def separa_arvore(indice, origem):
    copia_origem = Arvore(None)
    if len(origem.valor):
        copia_origem.valor = origem.valor[indice]

    for filho in origem.filhos:
        copia_filho = separa_arvore(indice, filho)
        copia_filho.pai = copia_origem
        copia_origem.filhos.append(copia_filho)

    return copia_origem
```

   Depois de rodar o algoritmo, vamos querer juntar as árvores para encontrar os valores dos nós intermediários. Podemos fazer isso com uma DFS e `reduce`.

```
def concatena_arvores(arvores):
    fusao = Arvore(None)
    fusao.valor = reduce(lambda string, arv: string + arv.valor,
        arvores, "")

    for i in xrange(len(arvores[0].filhos)):
        fusao_filho = concatena_arvores(
            map(lambda arvore: arvore.filhos[i], arvores))
        fusao_filho.pai = fusao
        fusao.filhos.append(fusao_filho)
```

```
        return fusao
```

2.3.2 Rodando o algoritmo com os dados do problema

2.3.3 Rodando o algoritmo com dados reais

Obtemos os dados no formato Newick do Rosalind, uma plataforma de ensino de bioinformática.

2.4 Extensões

Ao fazer esse exercício, notamos que a árvore já é uma entrada do problema. Como é possível obter a árvore de menor valor a partir das espécies

Esse problema é NP-completo [TODO: colocar referência] e o melhor algoritmo conhecido é [TODO]

3 Exercício 6.3 (Papadimitriou)

3.1 Enunciado

O Yuckdonald's está considerando abrir uma cadeia de restaurantes em Quaint Valley Highway (QVG). Os n locais possíveis estão em uma linha reta, e as distâncias desses locais até o começo da QVG são, em milhas e em ordem crescente, $m_1, m_2, \ldots, m_n$. As restrições são as seguintes:

- Em cada local, o Yuckdonald's pode abrir no máximo um restaurante. O lucro esperado ao abrir um restaurante no local i é p_i, onde $p_i > 0$ e $i = 1, 2, \ldots, n$.

- Quaisquer dois restaurantes devem estar a pelo menos k milhas de distância, onde k é um inteiro positivo.

Dê um algoritmo eficiente para computar o maior lucro total esperado, sujeito às restrições acima.

4 Exercício 4.5 (Tardos)

4.1 Enunciado

Vamos considerar uma rua campestre longa e quieta, com casas espalhadas bem esparsamente ao longo da mesma. (Podemos imaginar a rua como um grande segmento de reta, com um extremo leste e um extremo oeste.) Além disso, vamos assumir que, apesar do ambiente bucólico, os residentes de todas essas casas são ávidos usuários de telefonia celular.

Você quer colocar estações-base de celulares em certos pontos da rodovia, de modo que toda casa esteja a no máximo quatro milhas de uma das estações-base. Dê um algoritmo eficiente para alcançar esta meta, usando o menor número possível de bases.

5 Exercício 8.19 (Tardos)

5.1 Enunciado

Um comboio de navios chega ao porto com um total de n vasilhames contendo tipos diferentes de materiais perigosos. Na doca, estão m caminhões, cada um com capacidade para até k vasilhames. Para cada um dos dois problemas, dê um algoritmo polinomial ou prove NP-completude:

- Cada vasilhame só pode ser carregado com segurança em alguns dos caminhões. Existe como estocar os n vasilhames nos m caminhões de modo que nenhum caminhão esteja sobrecarregado, e todo vasilhame esteja num caminhão que o comporta com segurança?

- Qualquer vasilhame pode ser colocado em qualquer caminhão, mas alguns pares de vasilhames não podem ficar juntos num mesmo caminhão. Existe como estocar os n vasilhames nos m caminhões de modo que nenhum caminhão esteja sobrecarregado e que nenhum dos pares proibidos de vasilhames esteja no mesmo caminhão?

5.2 Item a

Uma solução força-bruta para esse problema seria:

- Estenda a lista de vasilhames com vasilhames vazios, até que ela tenha tamanho mk (a capacidade total de todos os caminhões). Vasilhames vazios podem ser transportados em qualquer caminhão (e correspondem a um lugar sobrando no mesmo).

- Para cada uma das $(mk)!$ ordenações da lista acima, considere que os k primeiros vasilhames vão para o primeiro caminhão, os k próximos para o segundo e assim até o final da lista. Se cada vasilhame estiver em um camihão que o comporta com segurança, retorne essa solução, se não, tente com uma nova ordem.

Esse algoritmo faz $(mk)!$ iterações do loop principal no pior caso, cada iteração tem custo mk para conferir se é uma solução válida. Isso dá uma complexidade total de $O(mk(mk)!)$

Esse é um algoritmo super-exponencial para o problema, mas isso não significa que o problema é NP-completo. Na verdade, como veremos a seguir, esse problema não é NP-completo pois aceita uma solução polinomial usando fluxos.

5.2.1 Solução com fluxos

Podemos transformar esse problemas em um problema de encontrar o fluxo máximo de um grafo usando a seguinte construção:

- Criamos um vértice s representando a fonte e um vértice t representando o dreno
- Para cada vasilhame $v_i \in \{v_1, v_2, \ldots, v_n\}$ criamos um vértice v_i e uma aresta (s, v_i) capacidade 1
- Para cada caminhão $C_i \in \{C_1, C_2, \ldots, C_m\}$ criamos um vértice C_i. Se o vasilhame v_j puder ser transportado com segurança no caminhão C_i criamos uma aresta (v_j, C_i) de capacidade 1. Para cada caminhão criamos também uma aresta (C_i, t) de capacidade k.

Dessa forma, existe uma configuração possível de caminhões se e somente se o fluxo máximo tem valor m.

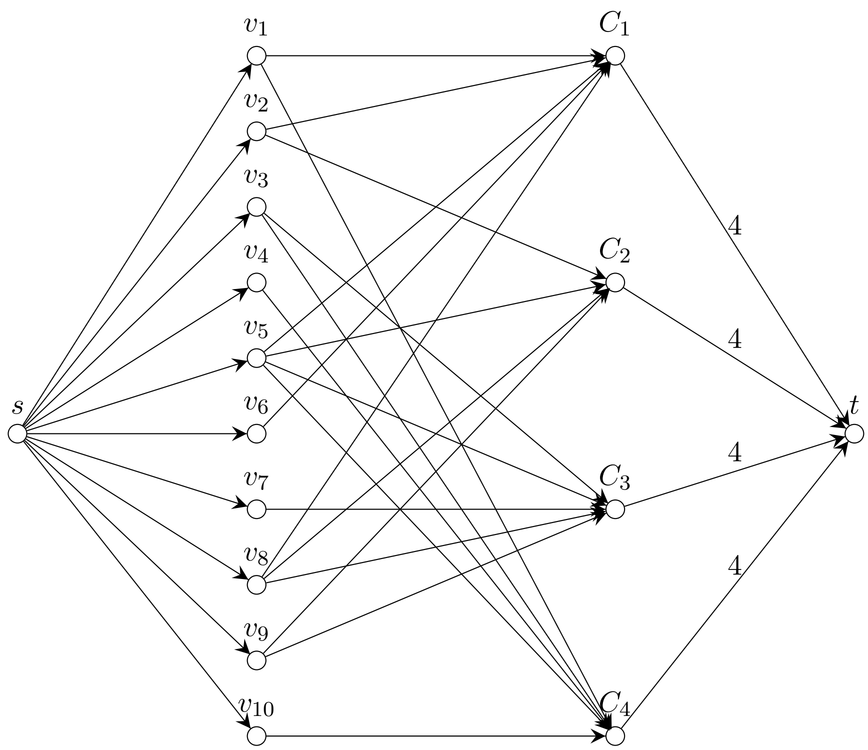
De fato, se encontramos um fluxo máximo de valor m então exatamente uma aresta com origem em cada vasilhame terá fluxo 1. Se colocarmos cada vasilhame no caminhão de destino da aresta de fluxo 1 obtemos um posicionamento válido. Por outro lado, se existe um arranjo válido, colocando 1 de fluxo nas arestas entre os vasilhames e os caminhões que os contém nesse arranjo obtemos um fluxo de valor m.

Para a seguinte situação:

Capacidade	3
Total de cam.	4
Total de vas.	10

Vasilhame	Cam. seguros
1	1, 4
2	1, 2
3	3, 4
4	4
5	1, 2, 3, 4
6	1
7	3
8	1, 2, 3
9	2, 3
10	4

A construção seria como ilustrado na figura abaixo. Omitimos as capacidades iguais a 1 para não poluir demais a imagem.



1. Implementação

   Primeiramente, precisamos ser capazes de ler a tabela acima para passar os valores para o nosso algoritmo.

```
capacidade_por_caminhao = regras[0][1]
total_de_vasilhames = regras[2][1]

vasilhames = collections.OrderedDict()
caminhoes = []
for line in seguros[1:]:
    # Nomeando os vasilhames
    vasilhame = 'v_%s' % line[0]
    vasilhames[vasilhame] = []
    for caminhao in str(line[1]).split(','):
        nome = 'C_%s' % caminhao.strip()
        vasilhames[vasilhame].append(nome)
        if nome not in caminhoes:
            caminhoes.append(nome)
```

Vamos usar a classe RedeDeFluxo, que definimos para a questão 7.28.

```
def cria_grafo(vasilhames, caminhoes, capacidade_por_caminhao):
    G = RedeDeFluxo()
    G.novo_vertice('Fonte')
    G.novo_vertice('Dreno')

    # Criando um vertice para cada caminhao e ligando esse vertice ao dreno
    for caminhao in caminhoes:
        G.novo_vertice(caminhao)
        G.nova_aresta(caminhao, 'Dreno', capacidade_por_caminhao, 0)

    for vasilhame, caminhoes in vasilhames.iteritems():
        # Criando um vertice para cada vasilhame e conectando a fonte a
        # cada um dos vasilhames
        G.novo_vertice(vasilhame)
        G.nova_aresta('Fonte', vasilhame, 1, 0)

        # Conectando o vasilhame a cada caminhao que pode transporta-lo
        for caminhao in caminhoes:
            G.nova_aresta(vasilhame, caminhao, 1, 0)

    return G
```

Como nesse problema as demandas já são 0, podemos aplicar Ford-Fulkerson diretamente, usando a mesma implementação que fizemos para o exercício 7.28.

Podemos então rodar Ford-Fulkerson e ver se o fluxo máximo encontrado é igual ao total de vasilhames. Se for, isso significa que o problema tem uma solução, que vamos retornar. Caso contrário não existe arranjo possível.

```
G = cria_grafo(vasilhames, caminhoes, capacidade_por_caminhao)
fluxo = G.fluxo_maximo('Fonte', 'Dreno')
if fluxo == total_de_vasilhames:
    tabela_de_vasilhames = []
    for vasilhame in vasilhames:
        for w in G.adj[vasilhame]:
            if G.fluxo[w] == 1:
                tabela_de_vasilhames.append([w.origem, w.destino])
    return tabela_de_vasilhames
else:
    return 'Impossivel'
```

A solução para a nossa entrada:

2. Complexidade

   O algoritmo de Ford-Fulkerson tem complexidade $O(VF)$ TODO: referência ao 7.28, em que V é a quantidade de vértices e F é o maior valor possível para o fluxo.

   No caso, o número de vértices é $m + n + 2$ e o maior fluxo possível é n, totalizando uma complexidade $O(n(m + n))$, o que é polinomial na entrada.

5.3 Item b

Vamos mostrar que é possível reduzir uma instância do 3-SAT a um problema de colocar vasilhames em caminhões seguindo as restrições do enunciado. De modo que, como 3-SAT é NP-completo, nosso problema também é.

5.3.1 Definindo 3-SAT

3-SAT é o problema de dado um conjunto de variáveis $v_1, \ldots, v_x$ e cláusulas $K_1, \ldots, K_y$, onde cada cláusula é constituída de no máximo três elementos do universo de *literais*, $\{v_1, \overline{v_1}, \ldots, v_x, \overline{v_x}\}$, encontrar uma atribuição de valores Verdadeiro/Falso para cada variável que faça com que todas as claúsulas sejam verdadeiras.

5.3.2 3-SAT $\rightarrow$ Caminhões

Dada uma instância do 3-SAT, vamos construir uma instância do problema enunciado que admite solução se e somente se tal instância do 3-SAT admite solução.

Primeiramente, note que podemos assumir que nosso problema de 3-SAT tem pelo menos quatro variáveis, adicionando variáveis que não aparecem em cláusula alguma se necessário.

1. Construção

   Vamos começar a construção sem as cláusulas:

   - São $x+1$ caminhões, cada um de capacidade $3x+y$
   - Existe um vasilhame para cada um dos $2x$ literais em $\{v_1, \overline{v_1}, \ldots, v_x, \overline{v_x}\}$. Esses vasilhames têm o mesmo nome do literal a que correspondem.
   - Existe um vasilhame adicional w_i para cada variável v_i

   Queremos criar restrições entre os vasilhames de modo que, em uma atribuição válida:

   (a) x dos caminhões contenham exatamente um literal verdadeiro cada.
   (b) O caminhão restante contenha todos os literais falsos;

   Para garantir as condições acima, vamos criar os seguintes conflitos:

   - w_i conflita com w_j para todo $i \neq j$.
   - w_i e conflita com v_j e com $\overline{v_j}$, se $i \neq j$.
   - v_i conflita com $\overline{v_i}$.

   Com isso, garantimos as seguintes propriedades:

   (P_1) Como todos os w_i conflitem entre si, é necessário um caminhão por w_i.
   (P_2) Como w_i conflita com todos os literais tais que $i \neq j$, o caminhão que contém w_i só pode conter literais correspondentes a i-ésima variável.
   (P_3) v_i e $\overline{v_i}$ não podem estar ambas no caminhão do w_i, pois elas conflitam entre si.

Mais ainda, *exatamente* um elemento do par $\{v_i, \overline{v_i}\}$ está no caminhão do w_i numa atribuição válida: Se nenhuma delas estivesse no caminhão do w_i, estariam ambas no único caminhão que não contém nenhum w (pois todos os outros caminhões contém um w_j com $i \neq j$, o que conflita com v_i e $\overline{v_i}$), o que também não pode acontecer.

Agora, vamos adicionar as cláusulas à nossa construção:

- Existe um vasilhame para cada uma das y cláusulas $K_1, \ldots, K_y$

Com seguinte conflito:

- K_i conflita com v_j se $v_j \notin K_i$. Similarlmente, K_i conflita com $\overline{v_j}$ se $\overline{v_j} \notin K_i$.

Ou seja, permitimos colocar o vasilhame da cláusula K_i num caminhão apenas se a cláusula contém todos os literais que vão viajar no caminhão.

Dessa forma, uma cláusula nunca pode viajar no caminhão dos literais falsos, pois cada a cláusula contém no máximo três literais e temos no mínimo quatro literais falsos, de modo que há garantidamente um literal que não aparece na cláusula e portanto conflita com ela.

2. Obtendo uma solução

   Para completar a nossa redução, precisamos de duas coisas:

   - A partir de uma solução do problema dos caminhões que construimos, encontrar em tempo polinomial uma solução do 3-SAT correspondente
   - Provar que quando nenhuma solução do problema dos caminhões existe o 3-SAT também não tem solução

   (a) Solução caminhões $\rightarrow$ Solução 3-SAT

   Se existe uma solução para o problema, então todo caminhão que contém um vasilhame do tipo w_i também contém um vasilhame correspondente a um literal; esse literal será marcado como verdadeiro. Todos os outros literais serão marcados como falsos. Essa marcação é consistente, pois para cada $i \in \{1, 2, \ldots, x\}$ exatamente um literal entre $v_i, \overline{v_i}$ que está no mesmo caminhão que w_i. Como todas as cláusulas têm que estar em um caminhão que contém um vasilhame do tipo w_i e esse caminhão tem que conter um literal que está na claúsula, essa marcação faz com que todas as cláusulas sejam verdadeiras.

(b) $\nexists$ solução caminhões $\Rightarrow \nexists$ solução 3-SAT

É mais fácil provar a contrapositiva, isso é, $\exists$ solução 3-SAT $\Rightarrow \exists$ solução caminhões.

Seja S os conjuntos dos literais verdadeiros na solução do 3-SAT. Então:

- $S \subset \{v_1, \overline{v_1}, \ldots, v_x, \overline{v_x}\}$
- $\forall\ 1 \leq i \leq x, |S \cap \{v_i, \overline{v_i}\}| = 1$
- $\forall\ 1 \leq j \leq y, S \cap K_j \neq \emptyset$

Podemos construir uma solução válida para o problema dos caminhões da seguinte forma:

- $\forall\ 1 \leq i \leq x$, coloque o vasilhame w_i no caminhão C_i
- $\forall\ 1 \leq i \leq x$, coloque o vasilhame $S \cap \{v_i, \overline{v_i}\}$ em C_i
- $\forall\ 1 \leq j \leq y$, seja i o menor valor tal que v_i ou $\overline{v_i}$ está em $S \cap K_j$. Coloque o vasilhame K_j em C_i.
- Coloque todos os literais em $\{v_1, \overline{v_1}, \ldots, v_x, \overline{v_x}\} - S$ no caminhão C_{x+1}

Isso respeita todas as restrições. De fato, cada K_j está num caminhão que só contém um vasilhame correspondente a um literal e, pelo item 3, o vasilhame do literal não conflita com o vasilhame da cláusula. Além disso, cada w_i está em seu próprio caminhão, nenhum par $\{v_i, \overline{v_i}\}$ aparece num mesmo caminhão e nenhum w_i aparece no mesmo caminhão de um literal v_j ou $\overline{v_j}$ com $i \neq j$.