Universidade do Porto L:EIC

# Desenho de Algoritmos 2021/22

## Exame (30.06.2022) - Parte Prática *90 minutos*

## Uma resolução

**1.** **a)** A estratégia *greedy* enunciada não garante a solução ótima. Por exemplo, para $L(3,3)$ o valor ótimo é 10, mas a solução obtida pela estratégia teria valor 9, porque enviaria as três caixas para a loja 3.

**b)** Resposta possível em C++:

```
#include <vector>
using namespace std;

#define NMAX 30

pair<int,vector<pair<int,int> > >  distrGreedy(int N, int M, int V[][NMAX])
{
  int j, k, jmax, maxval, dshops[M];

  // find the largest number each shop accepts
  for(j = 0; j < M; j++) {
    for(k = N; k > 0 && V[j][k-1] == 0 ; k--) ;
    dshops[j] = k;
  }

  // apply the greedy strategy
  vector<pair<int,int> >  sol;
  int lval = 0;
  while (N > 0) {
    maxval = 0; // best offer
    jmax = -1;  // shop making best offer
    for (j=0; j < M; j++)
      if (dshops[j] > 0)  {      // shop j is available
        if (dshops[j] > N) dshops[j] = N;      // because only N boxes remain
        if (V[j][dshops[j]-1] > maxval) {
          maxval = V[j][dshops[j]-1];
          jmax = j;
        }
    }
    if (maxval == 0) break;   // no shop accepts deliveries
    N -= dshops[jmax];
    lval += maxval;
    sol.push_back(make_pair(jmax+1,dshops[jmax]));
    dshops[jmax] = 0;   // no further deliveries to jmax
  }

  if (N == 0)   return make_pair(lval,sol);
  return make_pair(0,NULL);
}
```

**c)** Sabendo que $v_{jN} > 0$, para todo $j$, podemos definir $L(k, j)$, para $1 \leq j \leq M$ e $0 \leq k \leq N$, pela recorrência:

$$\begin{array}{ll} L(0,j) = 0 & \text{se } 1 \leq j \leq M \\ L(k,1) = v_{1,k} & \text{se } 1 \leq k \leq N \\ L(k,j) = \max\limits_{0 \leq p \leq k} (v_{jp} + L(k-p, j-1)) & \text{se } 1 \leq k \leq N \text{ e } 2 \leq j \leq M \end{array}$$

Para calcular $L(N, M)$ e $S(N, M)$, podemos definir o algoritmo seguinte, baseado em programação dinâmica, segundo uma abordagem *bottom-up*:

DISTROTIMA$(V, N, M)$

1 Para $k \leftarrow 1$ até $N$ fazer
2     $L[k] \leftarrow V[1, k]$;
3     $S[k] \leftarrow \{(1, k)\}$;
4 Para $j \leftarrow 2$ até $M$ fazer
5     Para $k \leftarrow N$ até $1$ **com decremento de 1** fazer
6         Para $p \leftarrow 1$ até $k - 1$ fazer     /* $k$ começa em 1 pois, inicialmente, $L[k]$ é já $v_{j,0} + L(k, j-1)$ */
7             Se $V[j, p] + L[k-p] > L[k]$ então
8                 $L[k] \leftarrow V[j, p] + L[k-p]$;   $S[k] \leftarrow \{(j, p)\} \cup S[k-p]$;
9         Se $V[j, k] > L[k]$ então     /* enviar as $k$ caixas para $j$ e zero para as lojas $1..(j-1)$ */
10             $L[k] \leftarrow V[j, k]$;   $S[k] \leftarrow \{(j, k)\}$;
11 retornar $(L(N), S(N))$;

Observação: Assumimos indexação a partir de 1 e que $L$ e $S$ são *arrays* com $N$ posições. Na linha 4, para $j$ fixo, $L[k]$ e $S[k]$ têm $L(k, j-1)$ e $S(k, j-1)$, para $1 \leq k \leq N$. Para obter $L(k, j)$ e $S(k, j)$, precisamos dos valores de $L(t, j-1)$ e $S(t, j-1)$, para $0 \leq t \leq k$. Assim, se, para cada $j$, a atualização de $L$ e $S$ for efetuada por ordem decrescente de $k$ (ciclo na linha 5), os dois *arrays* $L$ e $S$ são suficientes. Em alternativa, poder-se-ia usar dois *arrays* para $L$ e dois para $S$, para ter os valores anteriores e os novos (o que requer uma cópia em cada iteração). Nos dois casos, utiliza-se $\Theta(N)$ posições de memória adicional. Utilizar matrizes $N \times M$ para $L$ e $S$ seria desperdício de memória.

**d)** Assumindo que P $\neq$ NP, o problema não é NP-completo, pois resolve-se em tempo polinomial. O enunciado de 1c) afirma que existe um algoritmo $O(N^2 M)$ para obter o montante máximo, para qualquer instância (o tamanho do input é $O(NM)$). Para resolver o **problema de decisão** de 1d), basta comparar $T$ com o valor que DISTROTIMA$(V, N, M)$ retorna para $L(N, M)$. Se $T$ for menor ou igual, a resposta é "sim". Se não for, a resposta é "não". Tal algoritmo de decisão é polinomial pois DISTROTIMA$(V, N, M)$ é polinomial, por ser $O(N^2 M)$, como pedido em 1c).

**2.** **a)** Seja $J$ o conjunto das lojas a que a empresa deveria enviar caixas e seja $c_j$ o número de caixas que a loja $j$ deve receber, para $j \in J$, **segundo o definido por** $S(N, M)$. Assumimos que $A$ e $B$ são as transportadoras e que $A$ entrega o maior número de caixas, se necessário (i.e., se o ótimo não for zero).

- **Variáveis de decisão**: $y_{ij} \in \{0, 1\}$ será 1 sse a empresa $i$ entrega à loja $j$, com $i \in \{A, B\}$ e $j \in J$.
- **Dados:** $S(N, M)$, que, para facilitar, definimos pelo conjunto $J$ e os valores $c_j$, para $j \in J$.
- **Modelo do problema:**

$$\textbf{minimizar} \sum_{j \in J} c_j y_{Aj} - \sum_{j \in J} c_j y_{Bj} \quad \textbf{sujeito a}$$

$$\left| \begin{array}{ll} y_{ij} \in \{0, 1\}, \text{ para todo } j \in J \text{ e } i \in \{A, B\} & \\ y_{Aj} + y_{Bj} = 1, \text{ para todo } j \in J & \text{(exatamente uma das empresas entregará a } j\text{)} \\ \sum\limits_{j \in J} c_j y_{Aj} \geq \sum\limits_{j \in J} c_j y_{Bj} & \text{(a empresa } A \text{ entregará mais caixas, se necessário)} \end{array} \right.$$

**Modelo II: alternativa, que corresponde a substituir $y_{Bj}$ por $1 - y_{Aj}$ no anterior:**

- **Variáveis**: $y_j \in \{0, 1\}$ será 1 sse a empresa $A$ entregar à loja $j$, para $j \in J$.
- **Dados**: idêntico ao modelo anterior.
- Modelo do problema:

$$\textbf{minimizar} \sum_{j\in J} c_j y_j - \sum_{j \in J} c_j(1-y_j) \quad \textbf{sujeito a}$$

$$\left| \begin{array}{ll} y_j \in \{0,1\}, \text{ para todo } j \in J & \\ \displaystyle\sum_{j\in J} c_j y_j \geq \sum_{j\in J} c_j(1-y_j) & \qquad \text{(a empresa } A \text{ entregará mais caixas se necessário)} \end{array} \right.$$

**Modelo III (alternativo simplificado)**: Os dados e as variáveis de decisão são idênticos ao anterior. Assumimos que $N$ é obtido de $S(N, M)$.

$$\textbf{minimizar} \sum_{j\in J} c_j y_j \quad \textbf{sujeito a}$$

$$\left| \begin{array}{l} 2\displaystyle\sum_{j\in J} c_j y_j \geq N \\ y_j \in \{0,1\}, \text{ para todo } j \in J \end{array} \right.$$

Observação: No Modelo III, usámos o facto de $\sum_{j\in J} c_j = N$, o que é verdade por os valores a entregar definirem $S(N, M)$, sendo $N$ uma constante no modelo. A expressão da função objetivo seria

$$2\sum_{j\in J} c_j y_j - \sum_{j\in J} c_j$$

e ficaria $2\sum_{j\in J} c_j y_j - N$. Pode ser simplificada como indicámos porque, se multiplicarmos uma função $f$ por uma constante **positiva** qualquer, $bf(x)$ é mínimo para $x$ se e só se $f(x)$ é mínimo para $x$. A constante $N$ pode ser retirada porque, se adicionarmos uma constante $b \in \mathbb{R}$ qualquer a $f$, também $f(x)$ é mínimo sse $f(x) + b$ é mínimo.

Para a instância $N = 30$, $M = 10$ e $S = \{(1, 6), (2, 4), (4, 7), (5, 4), (7, 1), (9, 8)\}$, o modelo concretiza-se em:

$$\textbf{minimizar} \;\; 6y_1 + 4y_2 + 7y_4 + 4y_5 + y_7 + 8y_9 \quad \textbf{sujeito a}$$

$$\begin{cases} 2(6y_1 + 4y_2 + 7y_4 + 4y_5 + y_7 + 8y_9) \geq 30 \\ y_j \in \{0,1\}, \text{ para todo } j \in \{1,2,4,5,7,9\} \end{cases}$$

Para $N = 12$, $M = 2$ e $S = \{(1, 2), (2, 10)\}$, seria:

$$\textbf{minimizar} \;\; 2y_1 + 10y_2 \quad \textbf{sujeito a}$$

$$\begin{cases} 2(2y_1 + 10y_2) \geq 12 \\ y_1, y_2 \in \{0,1\} \end{cases}$$

No **primeiro caso**, uma solução ótima é $y_1 = y_2 = y_5 = y_7 = 1$ e $y_4 = y_9 = 0$, e o valor ótimo (i.e., o valor da função objetivo) para o Modelo II é zero. Essa solução determina que as entregas às lojas 1, 2, 5 e 7 são efetuadas pela empresa $A$ e as restantes (i.e., as entregas às lojas 4 e 9) são efetuadas por $B$. Também $y_1 = y_2 = y_5 = y_7 = 0$ e $y_4 = y_9 = 1$ seria uma solução ótima, correspondendo a trocar $A$ com $B$ na anterior. Existem outras soluções ótimas (por exemplo $y_1 = y_7 = y_9 = 1$ e $y_4 = y_2 = y_5 = 0$). No **segundo caso**, só há uma solução ótima para o modelo, $y_1 = 0$, $y_2 = 1$, sendo 8 a diferença em valor absoluto entre as duas empresas (i.e., 8 é o valor mínimo da função objetivo, se considerarmos o Modelo II).

**b)** <u>Algoritmo *greedy* para "equilibrar" as entregas:</u> Começar por ordenar as lojas por ordem decrescente de $c_j$, para $j \in J$. Considerando as lojas por essa ordem, para cada $j$, atribuir a entrega da loja $j$ à empresa $A$ se o número total de caixas já atribuídas a $A$ for inferior ou igual ao total atribuído a $B$; caso contrário, atribuir a $B$. Atualizar os totais de caixas atribuídas a $A$ e $B$. Se no fim $B$ estiver com mais caixas, basta trocar $A$ por $B$, para continuar a garantir que $A$ seria a que teria mais caixas.

A complexidade temporal é dominada pelo passo de ordenação, porque a fase de atribuição é $\Theta(m)$, sendo $m = |J|$. A ordenação pode ser realizada em $\Theta(m \log m)$, por exemplo, por *mergesort*, o que faria com que o algoritmo fosse $\Theta(m \log m)$.

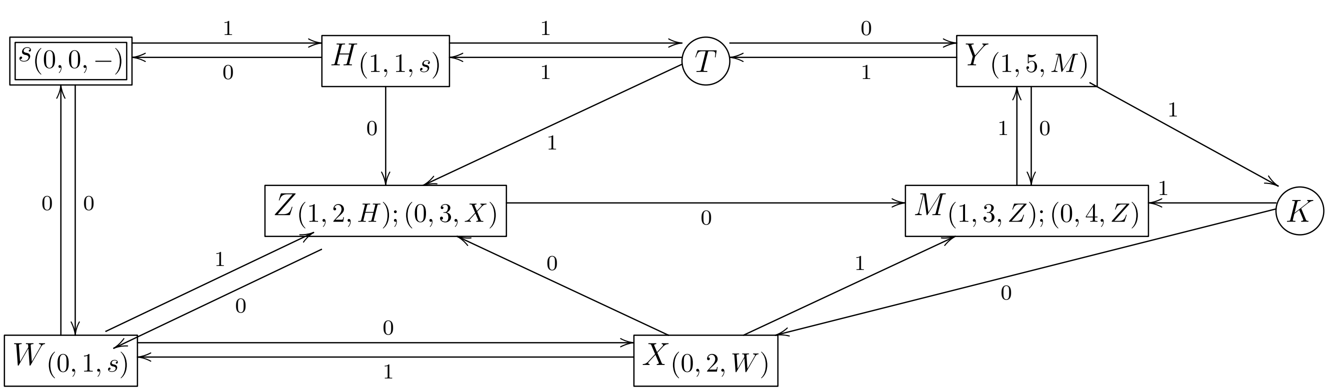
<u>Observação:</u> Para instância $N = 30$, $M = 10$ e $S = \{(1,6),(2,4),(4,7),(5,4),(7,1),(9,8)\}$, a solução obtida por este algoritmo é $y_9 = 1$, $y_4 = 0$, $y_1 = 0$, $y_2 = 1$, $y_5 = 1$, $y_7 = 0$. Esta solução não é ótima. O valor da função objetivo no Modelo II é 2 (e, como vimos acima, poderia ser zero).

**Sobre estratégias alternativas:** Notar que uma distribuição alternada, sem análise dos totais já acumulados, pode ir agravando o desequilíbrio.

**c)** Assumindo que P $\neq$ NP, nenhum problema NP-completo pode ser resolvido em tempo polinomial. Se o problema de otimização enunciado pudesse ser resolvido em tempo polinomial, também se podia decidir PARTITION em tempo polinomial. Mas, é conhecido que PARTITION é NP-completo.

<u>Resposta melhor:</u> Uma instância de PARTITION pode ser vista como uma instância do problema de entregas. Decidir PARTITION corresponde a determinar se se pode atribuir as entregas de modo que as duas empresas entreguem exatamente o mesmo número de caixas (ou seja, decidir se o ótimo do problema de otimização que formulámos é zero). Portanto, se tivermos um algoritmo polinomial para obter o ótimo, temos um algoritmo polinomial para decidir PARTITION, o que não é possível se P $\neq$ NP.

## 3. a)




No início, a fila só tem $s : (0,0,-)$.

Sai $s : (0,0,-)$. Coloca $H : (1,1,s)$ e $W : (0,1,s)$ na fila.

Sai $H : (1,1,s)$. Coloca $Z : (1,2,H)$ na fila. Não coloca $T$ porque ficaria com 2 constrangimentos. Não coloca $s$ porque $s$ já está "bloqueado" por ter tuplo com distância 0 e 0 constrangimentos.

Sai $W : (0,1,s)$. Coloca $X : (0,2,W)$ na fila. Não coloca $Z : (1,2,W)$ porque já tem $Z : (1,2,H)$. Não coloca $s$ porque $s$ está "bloqueado"

Sai $Z : (1,2,H)$. Coloca $M : (1,3,Z)$ na fila. Não coloca $W$ porque $W$ já está "bloqueado", porque tem tuplo com 0 constrangimentos (e menor distância).

Sai $X : (0,2,W)$: Coloca $Z : (0,3,X)$ na fila. Não coloca $M : (1,3,X)$ porque $M$ já tem $(1,3,Z)$. Não coloca $W$ porque $W$ já está "bloqueado".

Sai $M : (1,3,Z)$. Não coloca nada (o caminho para $Y$ teria dois constrangimentos).

Sai $Z : (0,3,X)$. Coloca $M : (0,4,Z)$ na fila. Não coloca $W$ porque $W$ já está "bloqueado".

Sai $M : (0,4,Z)$. Coloca $Y : (1,5,M)$ na fila.

Sai $Y : (1,5,M)$. Não coloca nada (caminhos para $T$ e $K$ teriam 2 constrangimentos). A fila fica vazia.

**b)** *Algoritmo adaptado de pesquisa em largura.* Um nó $v$ pode entrar duas vezes na fila, se o número de constrangimentos na primeira vez que for visitado for 1. Se for zero na primeira visita, fica "bloqueado". Na implementação, tal bloqueio corresponde a marcar $visitado[v]$ com 2.

```
BFS_VISIT_DIST_PROBS(s, G, sols)
  Para cada v ∈ G.V fazer
      visitado[v] ← 0;
      sols[v] ← {};
  visitado[s] ← 2;
  sols[s] = {(0, 0, NULL)};
  Q ← MKEMPTYQUEUE();
  Q.ENQUEUE( (s, 0, 0) );
  Repita
      t ← Q.DEQUEUE();
      v ← t.vert;
      Para cada w ∈ G.Adjs[v] fazer
          Se visitado[w] < 2 então
              c ← p((v, w)) + t.constrs;
              Se c ≤ 1 ∧ (visitado[w] = 0 ∨ c = 0) então
                  sols[w] ← sols[w] ∪ {(c, t.compr + 1, v)};
                  Se c = 0 então visitado[w] ← 2;
                  senão visitado[w] ← 1;
                  Q.ENQUEUE( (w, c, t.compr + 1) );
  até (Q.QUEUEISEMPTY() = true);
```

Para um terno $t = (v, c, d)$ na fila: $t.vert$ é o nó $v$ a que se refere, $t.compr$ o comprimento $d$ do caminho e $t.constrs$ o número de constragimentos $c$. Notar que $t.vert$ é o nó e não o que o antecede no caminho. Na descrição em 3a), os ternos que referimos são as `sols[v]`, que associámos aos nós, no desenho do grafo.

Observação: Numa implementação do algoritmo, em vez de colocar $(w, c, t.compr + 1)$ na fila, podiamos colocar um par $(w, r)$ formado pelo identificador do nó $w$ e o identificador $r$ do terno $(c, t.compr + 1, v)$, que define a solução correspondente, evitando a duplicação de informação.

**(Fim)**