

INSTITUTO SUPERIOR TÉCNICO

**Análise e Síntese de Algoritmos**

Ano Lectivo 2019/2020

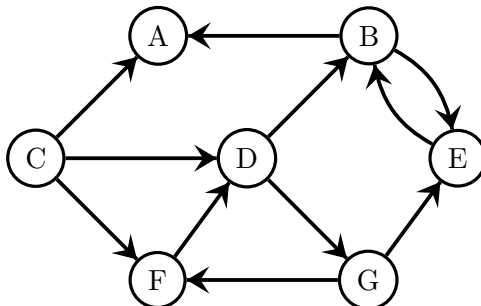
Exame de Época Especial

---

**RESOLUÇÃO**

---

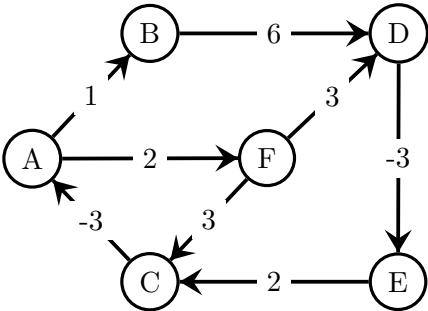
**I.a)** Considere o grafo dirigido da figura.



Indique os componentes fortemente ligados do grafo pela ordem segundo a qual são identificados pelo algoritmo e o valor *low* calculado para cada vértice. Considere que o tempo de descoberta *d* começa em 1.

	A	B	C	D	E	F	G
<i>low()</i>							
SCCs :							

I.b) Considere a aplicação do algoritmo de Johnson ao grafo dirigido e pesado da figura.



Calcule os valores de  $h(u)$  para todos os v rtices  $u \in V$  do grafo. Calcule tamb m os pesos de todos os arcos ap s a repesagem.

	A	B	C	D	E	F
$h()$						

$\hat{w}(A, B)$	$\hat{w}(A, F)$	$\hat{w}(B, D)$	$\hat{w}(C, A)$	$\hat{w}(D, E)$	$\hat{w}(E, C)$	$\hat{w}(F, C)$	$\hat{w}(F, D)$

**I.c)** Considere a rede de fluxo da figura onde  $s$  e  $t$  são respectivamente os vértices fonte e destino na rede. Aplique o algoritmo Relabel-To-Front na rede de fluxo. Considere que as listas de vizinhos dos vértices intermédios são as seguintes:

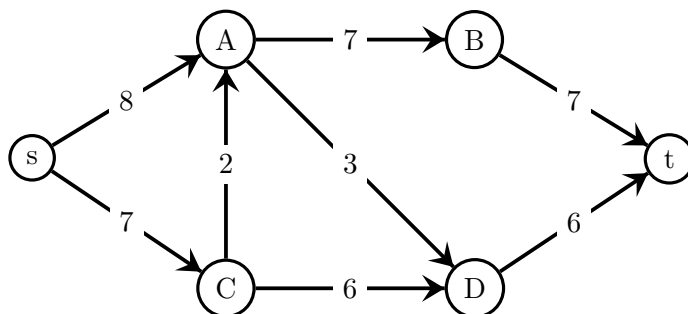
$N[A] = \langle B, D, s, C \rangle$

$N[B] = \langle t, A \rangle$

$N[C] = \langle D, A, s \rangle$

$N[D] = \langle t, A, C \rangle$

e que a lista de vértices inicial é  $L = \langle C, A, B, D \rangle$ .



Indique a altura final de cada vértice. Indique ainda um corte mínimo da rede, o valor do fluxo máximo, e a sequência de diferentes configurações de  $L$ .

	s	A	B	C	D	t
$h()$						
Corte :	/				f(S,T) =	
L:						

**I.d)** Considere o problema de compressão de dados de um ficheiro usando a codificação de Huffman. Indique o código livre de prefixo óptimo para cada carácter num ficheiro com 400 caracteres com a seguinte frequência de ocorrências:

$$f(a) = 34, f(b) = 20, f(c) = 5, f(d) = 11, f(e) = 13, f(f) = 17.$$

Quando constrói a árvore, considere o bit 0 para o nó com menor frequência. Indique também o total de bits no ficheiro codificado.

	a	b	c	d	e	f
Codificação						
Total Bits						

**1.e)** Considere o padrão  $P = abaaab$  e construa o autômato finito que emparelhe este padrão numa cadeia de caracteres. Indique o estado resultante das seguintes transições:

$\delta(0, b)$	$\delta(1, a)$	$\delta(2, b)$	$\delta(3, b)$	$\delta(4, b)$	$\delta(5, a)$	$\delta(6, a)$	$\delta(6, b)$

Ilustre a sua aplicação no seguinte texto  $T = aabaaabaaabaaba$  e indique o número total de matches.

[illegible]

**I.f)** Considere o problema de multiplicar cadeias de matrizes. O objetivo é determinar por que ordem devem ser feitas as multiplicações por forma a minimizar o número total de multiplicações escalares que precisam de ser efetuadas.

Considere uma sequência com 4 matrizes  $A_1(3 \times 2)$ ,  $A_2(2 \times 5)$ ,  $A_3(5 \times 1)$ ,  $A_4(1 \times 4)$ , com as respectivas dimensões entre parênteses. Resolva este problema preenchendo a matriz  $m[i, j]$  que guarda o menor número de multiplicações escalares que precisam de ser efectuadas para obter o produto das matrizes de  $A_i$  a  $A_j$ . Indique os valores de  $m[1, 3]$ ,  $m[1, 4]$  e  $m[2, 4]$ . Indique também a colocação de parênteses que obtém o valor indicado em  $m[1, 4]$ . Em caso de empate associe à esquerda.

$m[1, 3]$	$m[1, 4]$	$m[2, 4]$	Parênteses

**II. (2 + 1,5 + 1,5 + 2 + 1,5 + 1,5 = 10 val.)**

**II.a)** Considere a seguinte implementação naif de uma fila de prioridade mínima baseada em listas simplesmente ligadas. Uma fila de prioridade é guardada em memória como uma lista de nós, cada qual associado a uma prioridade `pri` e a um identificador `id`. A implementação é composta pelas funções: `Insert(Lst lst, Lst node)` que insere o nó `node` na lista `lst`; `Remove(Lst lst, int i)` que remove o nó com identificador `i` da lista `lst`; `ExtractQueue(Queue q)` que remove o nó com prioridade mínima da fila de prioridade `q`; e `DecreaseKey(Queue q, Lst node, int pri)` que diminui a prioridade do nó `node` para `pri` na fila de prioridade `q`.

```
typedef struct Node {
    int id;
    int pri;
    struct Node* next;
} *Lst;

typedef struct QueueNode {
    Lst hd;
} *Queue;

int ExtractQueue(Queue q) {
    if (q->hd == NULL) return -1;

    Lst hd = q->hd;
    q->hd = hd->next;
    return hd->id;
}

Lst Remove(Lst lst, int id) {
    if (lst == NULL) return lst;

    if (lst->id == id) return lst->next;

    Lst prev = lst;
    Lst cur = lst->next;
    while (cur != NULL) {
        if (cur->id == id) {
            prev->next = cur->next;
            break;
        }
        prev = cur;
        cur = cur->next;
    }
    return lst;
}

Lst Insert (Lst lst, Lst node) {
    if (lst == NULL) return node;
    if (node->pri <= lst->pri) {
        node->next = lst;
        return node;
    } else {
```



```

        Lst ret = Insert(lst->next, node);
        lst->next = ret;
        return lst;
    }
}

void DecreaseKey (Queue q, Lst node, int pri) {
    Lst lst = Remove(q->hd, node->id);
    node->pri = pri;
    lst = Insert(lst, node);
    q->hd = lst;
}

```

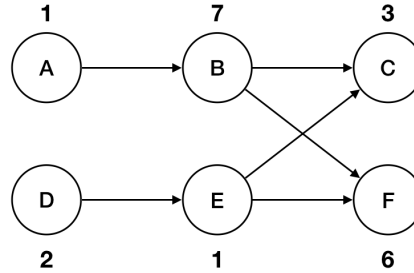
1. Determine o menor majorante assintótico para funções **Insert**, **Remove**, **ExtractQueue** e **DecreaseKey** em função do número de elementos,  $n$ , contidos na fila de prioridade ou lista que respectivamente recebem como argumento. Deve indicar para cada uma das funções a equação do tempo que expressa o número instruções executadas em função do tamanho do input (i.e.  $T(n) = \dots$ ).
2. Admitindo que esta implementação é utilizada como parte de uma implementação do Algoritmo de Dijkstra, qual seria a complexidade assintótica deste algoritmo? Justifique a resposta.

### Solução:

1. As equações do tempo e respectivos majorantes seguem em baixo:
  - $T_{EQ}(n) = O(1)$ , which implies that: **ExtractQueue**  $\in O(1)$ .
  - $T_R(n) = \sum_{i=1}^n O(1) = O(n)$ , which implies that: **Remove**  $\in O(n)$ .
  - $T_I(n) = O(1) + T_I(n-1) = \sum_{i=1}^n O(1) = O(n)$ , which implies that: **Insert**  $\in O(n)$ .
  - $T_{DK}(n) = O(1) + T_R(n) + T_I(n-1) = O(1) + O(n) + O(n-1) = O(n)$ , which implies that: **DecreaseKey**  $\in O(n)$ .
2. Com esta implementação naif, o Algoritmo de Dijkstra terá complexidade  $O(E.V)$ , dado que o algoritmo de Dijkstra executa  $|E|$  relaxações, sendo que em cada relaxação é efectuada uma chamada á função **DecreaseKey** que tem complexidade linear no número de vértices contidos na fila de prioridade.



**II.b)** Suponha dado um grafo dirigido  $G = (V, E)$  em que cada nó  $u \in V$  é associado a um valor inteiro,  $p_u$ , que dizemos ser o preço do nó. Para cada nó  $u \in V$  definimos o custo de  $u$ ,  $c_u$ , como sendo o preço do nó mais caro atingível a partir de  $u$ , incluindo o próprio  $u$ . Por exemplo, no grafo em baixo, cujos nós estão anotados com os respectivos preços, temos que:  $c_A = 7$ ,  $c_B = 7$ ,  $c_C = 3$ ,  $c_D = 6$ ,  $c_E = 6$ , e  $c_F = 6$ .



1. Proponha um algoritmo que, dado um grafo dirigido acíclico (DAG) e um vector de preços associando um preço a cada nó, calcula o correspondente vector de custos. Indique a complexidade assintótica do algoritmo proposto.

*Pista:* Comece por determinar a ordem segundo a qual deve visitar os nós para o cálculo dos respectivos custos.

2. A partir do algoritmo proposto na alínea anterior, proponha um algoritmo que funciona para qualquer grafo dirigido. Indique a complexidade assintótica do algoritmo proposto.

*Pista:* Lembre-se que todo o grafo dirigido cíclico induz um grafo dirigido acíclico.

### Solução:

1. Dado um grafo dirigido acíclico  $G = (V, E)$  e um vector de preços  $\vec{p}$ :

- Determinar uma ordenação topológica dos vértices de  $G$  usando uma DFS (complexidade:  $O(V + E)$ ).
- Percorrer os vértices de  $V$  por ordem topológica inversa, calculando o custo de cada vértice  $u \in V$  de acordo com a seguinte fórmula:

$$c_u = \max(p_u, \max\{c_v \mid v \in N[u]\})$$

Onde  $N[u]$  denota o conjunto dos vizinhos de  $u$  em  $G$  (complexidade:  $O(V + E)$ ).

Complexidade do algoritmo:  $O(V + E)$ .

2. Dado um grafo dirigido não necessariamente acíclico  $G = (V, E)$  e um vector de preços  $\vec{p}$ :

- Calcular os SCCs de  $G$  usando o algoritmo de Tarjan ou o algoritmo de Kosaraju (complexidade:  $O(V + E)$ ).
- Utilizar o algoritmo da alínea anterior para calcular o custo de cada SCC (complexidade:  $O(V + E)$ ).
- Atribuir a cada vértice do grafo original o custo do SCC correspondente (complexidade:  $O(V)$ ).

Complexidade do algoritmo:  $O(V + E)$ .



**II.c)** Existem diversos eventos para os quais a procura de bilhetes é bastante superior aos disponíveis. Um desses eventos é o Mundial de Futebol. Suponha que no calendário temos um conjunto de  $m$  jogos,  $\{J_1, \dots, J_m\}$ , tal que para cada jogo  $J_i$  (com  $1 \leq i \leq m$ ) existem  $b_i$  bilhetes disponíveis. Existe um conjunto de  $n$  pessoas interessadas em adquirir bilhetes,  $\{P_1, \dots, P_n\}$ , tal que cada pessoa  $P_k$  (com  $1 \leq k \leq n$ ) pode pedir 1 bilhete para cada jogo que lhe interessa assistir. No entanto, devido a uma procura elevada de bilhetes foi decidido atribuir a cada pessoa apenas 1 bilhete para um único jogo. Pretende-se calcular a atribuição de bilhetes a pessoas.

1. Modele o problema da atribuição de bilhetes como um problema de fluxo máximo. A resposta deve incluir o procedimento utilizado para determinar a atribuição de bilhetes a pessoas.
2. Admitindo que o número de pessoas interessadas em assistir a jogos é muito superior ao número de jogos ( $n \gg m$ ) e também superior ao número de bilhetes disponíveis por jogo, indique o algoritmo que utilizaria para a calcular o fluxo máximo, bem como a respectiva complexidade assintótica medida em função dos parâmetros do problema (número de jogos,  $m$ , e número de pessoas interessadas,  $n$ ). De entre os algoritmos de fluxo estudados nas aulas deve escolher aquele que garanta a complexidade assintótica mais baixa para o problema em questão.  
*Nota:* A resposta deverá necessariamente incluir as expressões que definem o número de vértices e de arcos da rede de fluxo proposta ( $|V|$  e  $|E|$ , respectivamente) em função dos parâmetros do problema.

### Solução:

1. *Construção da rede de fluxo*  $G = (V, E, w, s, t)$ . Na construção da rede de fluxo consideramos um vértice por jogo, um vértice por pessoa e dois vértices adicionais  $s$  e  $t$ , respectivamente a fonte e o sumidouro. Formalmente:

$$\bullet V = \{J_i \mid 1 \leq i \leq m\} \cup \{P_k \mid 1 \leq k \leq n\} \cup \{s, t\}$$

•

$$\begin{aligned} E = & \{(s, P_i, 1) \mid 1 \leq i \leq n\} & P_i \text{ só tem direito a um único bilhete} \\ & \cup \{(P_i, J_k, 1) \mid J_k \in \mathbf{wishes}(P_i)\} & P_i \text{ deseja assistir ao jogo } J_k \\ & \cup \{(J_k, t, b_k) \mid 1 \leq k \leq m\} & \text{Existem } b_k \text{ bilhetes disponíveis para } J_k \end{aligned}$$

Onde  $\mathbf{wishes}(P_i)$  denota o conjunto dos jogos que a pessoa  $P_i$  está interessada em assistir. Uma vez calculado o fluxo máximo,  $f^*$ , é atribuído à pessoa  $P_i$  um bilhete para o jogo  $J_k$  se  $f^*(P_i, J_k) = 1$ .

2. *Complexidade.*

- $|V| = m + n + 2 \in O(n + m)$
- $|E| \leq n + m \cdot n + m \in O(n \cdot m)$
- $|f^*| = \sum_{k=1}^m b_k \leq \sum_{k=1}^m n = O(n \cdot m)$
- Ford-Fulkerson:  $O((n \cdot m)^2)$
- RF:  $O((n + m)^3)$

A melhor solução consiste em usar um algoritmo baseado no método de Ford Fulkerson.



**II.d)** Uma empresa utiliza quatro fábricas para produzir automóveis. A produção de cada automóvel em cada fábrica requer recursos, os quais se encontram divididos entre horas de mão de obra, unidades de materiais utilizados, e unidades de poluição produzida, como se ilustra na tabela em baixo:

Fábrica	Mão de obra	Materiais	Poluição
1	2	3	15
2	3	4	10
3	4	5	9
4	5	6	7

As restrições de produção são as seguintes:

1. Por acordo com os sindicatos, a fábrica 3 produz pelo menos 400 automóveis.
2. A mão de obra total não pode exceder as 3300 horas.
3. Os materiais disponíveis não podem exceder as 4000 unidades.
4. A poluição total produzida (por mês) não pode exceder as 12000 unidades; além disso, a poluição produzida pela fábrica 4 não pode exceder as 500 unidades por esta se encontrar muito próxima de uma povoação.

O objectivo é maximizar a produção de automóveis, tendo em conta as restrições existentes.

1. Formule o programa linear que permite resolver este problema.
2. A solução básica inicial do programa linear é exequível? Caso não seja, formule o programa linear auxiliar.
3. Formule o programa linear dual.
4. É sabido que, no pior caso, o algoritmo Simplex pode realizar um número exponencial de operações de pivotagem. Para qual dos programas lineares (primal ou dual) esse número é mais baixo? Justifique.

**Solução:**

1. Programa linear primal:

$$\begin{array}{rcllcl}
 \max & x_1 & & +x_2 & +x_3 & +x_4 & & \\
 \text{s.a} & & & & & x_3 & \geq & 400 \\
 & 2x_1 & & +3x_2 & +4x_3 & +5x_4 & \leq & 3300 \\
 & 3x_1 & & +4x_2 & +5x_3 & +6x_4 & \leq & 4000 \\
 & 15x_1 & & +10x_2 & +9x_3 & +7x_4 & \leq & 12000 \\
 & & & & & +7x_4 & \leq & 500 \\
 & & & x_1, x_2, x_3, x_4 & \geq & & & 0
 \end{array}$$

2. A solução básica inicial não é exequível (a primeira restrição não é satisfeita). Programa linear auxiliar:

$$\begin{array}{rcllcl}
 \max & & & & & -x_0 & & \\
 \text{s.a} & & & & & -x_3 & -x_0 & \leq -400 \\
 & 2x_1 & & +3x_2 & +4x_3 & +5x_4 & -x_0 & \leq 3300 \\
 & 3x_1 & & +4x_2 & +5x_3 & +6x_4 & -x_0 & \leq 4000 \\
 & 15x_1 & & +10x_2 & +9x_3 & +7x_4 & -x_0 & \leq 12000 \\
 & & & & & +7x_4 & -x_0 & \leq 500 \\
 & & & x_1, x_2, x_3, x_4, x_0 & \geq & & & 0
 \end{array}$$

3. Programa linear dual:

$$\begin{array}{rcccccccl}
 \max & -400y_1 & +3300y_2 & +400y_3 & +12000y_4 & +500y_5 & & & \\
 \text{s.a} & & +2y_2 & 3y_3 & +15y_4 & & & \geq & 1 \\
 & & +3y_2 & +4y_3 & +10y_4 & & & \geq & 1 \\
 & -y_1 & +4y_2 & +5y_3 & +9y_4 & & & \geq & 1 \\
 & & +5y_2 & +6y_3 & +7y_4 & +7y_5 & \geq & 1 \\
 & & & y_1, y_2, y_3, y_4 & & & \geq & 0
 \end{array}$$

4. É idêntico em ambos os casos.

- Programa primal:  $\binom{9}{4} = \frac{9!}{4!*5!}$
- Programa dual:  $\binom{9}{5} = \frac{9!}{4!*5!}$






1. Seja  $\mathbf{sub}(i)$  o máximo de entre os valores de todas as subquências contíguas de  $S$  que terminam na posição  $i$ . Defina  $\mathbf{sub}(i)$  recursivamente completando os campos em baixo.

$$\text{sub}(i) = \begin{cases} S[i] & \text{se } \boxed{\phantom{x}} \\ \phantom{S[i]} & \text{caso contrário} \end{cases}$$

- ```

MaxSubSeq( $S[1..n]$ )
  let  $sub[0..n]$  be a vector of size  $n+1$ 
   $sub[0] = 0$ 
  for  $i = 1$  to  $n$  do
    
  endfor
  return  $sub$ 

```

- Solução:**

- ```

MaxSubSeq( $S[1..n]$ )
  let  $sub[0..n]$  be a vector of size  $n+1$ 
   $sub[0] = 0$ 
  for  $i = 1$  to  $n$  do
    if ( $sub[i - 1] \leq 0$ ) then
       $sub[i] = S[i]$ 
    else
       $sub[i] = sub[i - 1] + S[i]$ 
    endif
  endfor
  return  $sub$ 

```

Complexidade:  $O(n)$

3. A partir do vector  $sub$ , calculamos o valor da subsequência contígua de  $S$  de valor máximo da seguinte maneira:

$$\max(0, \max\{sub(i) \mid 1 \leq i \leq n\})$$

**II.f)** O professor da disciplina de XPTO gerou aleatoriamente  $n$  de testes,  $T = \{t_1, \dots, t_n\}$ , para avaliar automaticamente as soluções submetidas pelos alunos para um dos projectos da disciplina. Além disso, o professor implementou duas soluções canónicas para o problema proposto, respectivamente com  $m_1$  e  $m_2$  linhas de código.

Dadas as limitações de performance do sistema utilizado para avaliar as soluções submetidas, o professor quer agora seleccionar um subconjunto dos testes gerados para serem de facto utilizados na avaliação. Para isso, determinou para cada teste as linhas das soluções canónicas activadas pela sua execução; formalmente, cada teste  $t_i \in T$  é associado a dois conjuntos  $L_i^1 \subseteq \{1, \dots, m_1\}$  e  $L_i^2 \subseteq \{1, \dots, m_2\}$ , correspondentes às linhas das soluções canónicas 1 e 2 activadas pelo teste. O professor quer agora determinar o mais pequeno conjunto de testes que garante a cobertura total de todas as linhas das duas soluções canónicas. Formalmente, este problema pode ser modelado através do seguinte problema de decisão:

$$\mathbf{TestSuite} = \{\langle \mathcal{L}, k, m_1, m_2 \rangle \mid \exists i_1, \dots, i_k. |\cup_{1 \leq j \leq k} L_{i_j}^1| = m_1 \wedge |\cup_{1 \leq j \leq k} L_{i_j}^2| = m_2\}$$

onde:  $\mathcal{L} = \{(L_i^1, L_i^2) \mid t_i \in T\}$  e  $k$  corresponde ao número máximo de testes que o professor admite seleccionar.

1. Mostre que o problema **TestSuite** está em **NP**.
2. Mostre que o problema **TestSuite** é **NP**-difícil por redução a partir do problema da *Cobertura de Conjuntos*, que é sabido tratar-se de um problema **NP**-difícil e que se define em baixo.

*Problema da Cobertura de Conjuntos:* Seja  $\mathcal{X} = \{X_i \mid 1 \leq i \leq n\}$  uma família de conjuntos e  $X$  o conjunto correspondente à união de todos os conjuntos de  $\mathcal{X}$ , i.e.  $X = \cup \mathcal{X}$ ; o problema da cobertura de conjuntos, **SetCover**, define-se formalmente da seguinte maneira:

$$\mathbf{SetCover} = \{\langle \mathcal{X}, k \rangle \mid \exists i_1, \dots, i_k. \cup_{1 \leq j \leq k} X_{i_j} = X\}$$

**Solução:**

1. O algoritmo de verificação recebe como input uma possível instância  $\langle \mathcal{L}, k, m_1, m_2 \rangle$  do problema e um conjunto de índices  $I$  (o certificado), sendo que cada índice  $i \in I$  corresponde a um teste  $t_i$ . O algoritmo tem de verificar que  $|I| = k$  e que  $\cup_{i \in I} |L_i^1| = m_1$  e  $\cup_{i \in I} |L_i^2| = m_2$ . Observamos que:
  - O certificado tem tamanho  $O(n)$ ;
  - *Algoritmo de verificação:* o algoritmo mantém em memória dois vectores  $\vec{v}_1$  e  $\vec{v}_2$ , respectivamente de tamanho  $m_1$  e  $m_2$ , que guardam as linhas das soluções canónicas 1 e 2 que já foram cobertas pelos testes analisados; i.e.  $\vec{v}_1[i] = 1$  sse a linha  $i$  do solução canónica 1 já foi coberta pelos testes analisados. Os vectores  $\vec{v}_1$  e  $\vec{v}_2$  são criados com todas as posições a 0. Para cada teste  $t_i$ , o algoritmo percorre as linhas do teste e coloca as respectivas posições dos vectores  $\vec{v}_1$  e  $\vec{v}_2$  a 1. Isto faz-se em tempo linear no número de linhas do teste. A complexidade do algoritmo de verificação é portanto:

$$\begin{aligned} \sum_{i \in I} O(\text{size}(t_i)) &\leq \sum_{i \in I} O(\min(m_1, m_2)) \\ &\leq O(|I| * (\min(m_1, m_2))) \\ &\leq O(n * \min(m_1, m_2)) \end{aligned}$$

2. Dada uma possível instância  $\langle \mathcal{X}, k \rangle$  do problema **SetCover**, temos de construir uma instância  $\langle \mathcal{L}, k, m_1, m_2 \rangle$  do problema **TestSuite**. Intuitivamente, admitimos

que a união de todos os conjuntos em  $\mathcal{X}$  corresponde ao conjunto de linhas da solução canónica 1 e que a solução canónica 2 é vazia. Formalmente:

$$\langle \mathcal{X}, k \rangle \in \mathbf{SetCover} \Leftrightarrow \langle \mathcal{L}, k, m_1, m_2 \rangle \in \mathbf{TestSuite}$$

onde:

- $\mathcal{L} = \{(X, \emptyset) | X \in \mathcal{X}\}$
- $m_1 = |\cup_{X \in \mathcal{X}} X|$
- $m_2 = 0$

Complexidade da redução:  $O(n.m)$ , onde  $n$  é o número de conjuntos em  $\mathcal{X}$  e  $m$  o tamanho da união de todos os conjuntos de  $\mathcal{X}$ .

