2º Teste de Análise e Desenho de Algoritmos

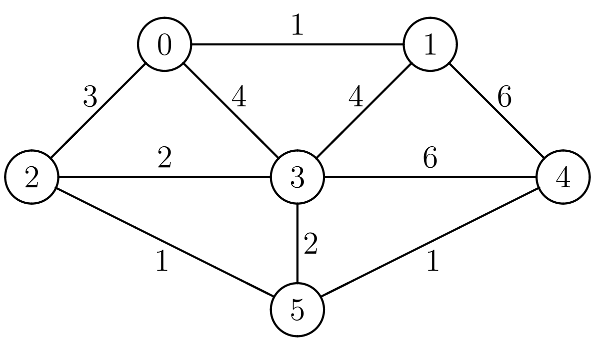
Departamento de Informática da FCT NOVA

9 de Junho de 2016

Responda a **perguntas** diferentes em **folhas** diferentes.

Se precisar de folhas, peça ao docente.

1. [4 valores] Suponha que se executa o algoritmo de Prim com o grafo esquematizado na figura.



Assumindo que a origem é o vértice 0 (ou seja, que o método *G.aNode()* retorna 0):

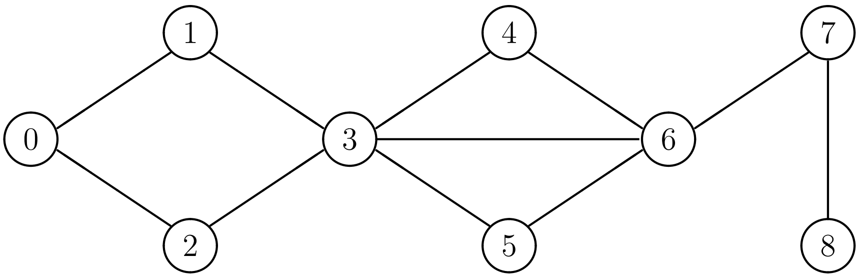
- indique a ordem pela qual os arcos são inseridos no resultado (i.e., no vetor *mst*);
- represente a árvore mínima de cobertura encontrada, desenhando os vértices e os arcos do grafo da forma usual;
- indique o custo da árvore encontrada.

2. [6 valores] Sejam $G = (V, A)$ um grafo não orientado e não pesado, e k e p dois inteiros positivos. Uma *componente conexa de G de cardinalidade k e profundidade p* é um subconjunto de vértices, $V' \subseteq V$, que verifica as duas seguintes propriedades:

- $|V'| = k \quad (V'$ tem k elementos);
- Para quaisquer vértices distintos $x, y \in V'$, existe um caminho de x para y em G cujo comprimento não excede p.

Por exemplo, $X = \{3, 4, 5, 6, 7\}$ é uma componente conexa de cardinalidade 5 e profundidade 2 do grafo esquematizado na figura. X tem 5 vértices e, entre dois quaisquer vértices distintos de X, há (pelo menos) um caminho de comprimento inferior ou igual a 2, como se pode verificar pelo seguinte conjunto de caminhos de G:

3 4	3 5	3 6	3 6 7
	4 3 5	4 6	4 6 7
		5 6	5 6 7
			6 7



O **Problema da Componente Conexa** formula-se da seguinte forma.
Dados um grafo não orientado e não pesado $G = (V, A)$ e dois inteiros positivos k e p, existe uma componente conexa de G de cardinalidade k e profundidade p?

Prove que o Problema da Componente Conexa é NP-completo.
Sugestão: Assuma que o grafo está implementado em matriz de adjacências. Também pode assumir que há um algoritmo que calcula o comprimento dos caminhos mais curtos entre dois (quaisquer) vértices do grafo cuja complexidade temporal é $O(|V|^2)$.

3. [6 valores] A classe *TriCounter* implementa contadores "ternários em base 2". Cada contador guarda uma sequência da forma $t_k\, t_{k-1} \cdots t_1\, t_0$ cujos elementos são os números -1, 0 ou 1 (com $k \geq 0$). O número representado pela sequência é

$$t_k\, 2^k \;+\; t_{k-1}\, 2^{k-1} \;+\; \cdots \;+\; t_1\, 2^1 \;+\; t_0\, 2^0\,.$$

Por exemplo, as sequências $0\,0\,1\,1$ e $0\,1\,0\,-1$ representam ambas o número 3, porque:

$$\begin{aligned} 3 &= 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + (-1) \times 2^0 \end{aligned}$$

Considere a função $\Phi(C)$, que atribui a cada objeto C da classe *TriCounter* o número de valores diferentes de zero guardados no vetor $C.counter$. Se p representar o número de posições de $C.counter$ que têm o valor 1 e n representar o número de posições de $C.counter$ que têm o valor -1:

$$\Phi(C) = p + n\,.$$

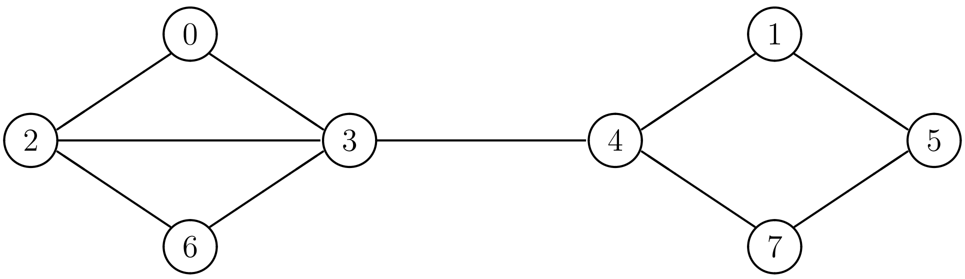
Prove que Φ é uma função potencial válida e calcule as complexidades amortizadas dos métodos *increment* e *decrement*, justificando. No estudo da complexidade amortizada do método *increment*, assuma que não é levantada a exceção, mas analise separadamente os casos em que a última atribuição (`counter[pos]++`): substitui o valor -1 pelo valor 0; substitui o valor 0 pelo valor 1. No estudo da complexidade amortizada do método *decrement*, assuma que não é levantada a exceção, mas analise separadamente os casos em que a última atribuição (`counter[pos]--`): substitui o valor 0 pelo valor -1; substitui o valor 1 pelo valor 0.

```java
public class TriCounter {

    // Invariant: any value stored in counter is -1, 0 or 1.
    private int[] counter;

    public TriCounter( int length ) {
        counter = new int[length];
        // All counter positions are set to 0.
    }

    public void increment( ) throws RuntimeException {
        int pos = 0;
        while ( pos < counter.length && counter[pos] == 1 )
            counter[pos++] = 0;

        if ( pos == counter.length )
            throw new RuntimeException("Counter overflow");

        counter[pos]++; // Last assignment.
    }

    public void decrement( ) throws RuntimeException {
        int pos = 0;
        while ( pos < counter.length && counter[pos] == -1 )
            counter[pos++] = 0;

        if ( pos == counter.length )
            throw new RuntimeException("Counter underflow");

        counter[pos]--; // Last assignment.
    }

}
```

4. [4 valores] Uma rede de computadores pode ser modelizada por um grafo não orientado e conexo.[1] Quando se adota encaminhamento multi-caminho, o tráfego entre cada par de nós (a origem o e o destino d dos pacotes) é encaminhado por vários caminhos, escolhidos de forma a satisfazer diversas propriedades: reduzir o tempo total de transmissão, equilibrar a distribuição da carga na rede, aumentar a tolerância a falhas (quer de nós, quer de ligações), etc.

   Neste exercício, o objetivo é descobrir se a infra-estrutura da rede (ou seja, o grafo) resiste bem às falhas de ligações. Para isso, pretende-se responder à seguinte pergunta, para quaisquer dois nós distintos o e d:

   > No máximo, quantas ligações podem falhar simultaneamente
   > sem comprometer a existência de um caminho entre o e d?

   Vejamos dois exemplos, com a rede esquematizada na figura.

   - Para os nós 2 e 3, a resposta é **2**.
     Repare que continua a haver caminho entre os nós 2 e 3, quaisquer que sejam as **2** ligações em baixo.
     Mas, se estiverem **3** ligações em baixo, pode não haver caminho entre os nós 2 e 3 (porque as **3** falhas podem ser, por exemplo, nas ligações $(2, 0)$, $(2, 3)$ e $(2, 6)$).
   - Para os nós 2 e 7, a resposta é **0**.
     Pode não haver caminho entre os nós 2 e 7 com **1** só falha (se essa falha for na ligação $(3, 4)$).

   Apresente uma função (em pseudo-código) que recebe:

   - uma rede $R = (V, A)$, que é um grafo não orientado e conexo, e
   - dois nós distintos, o e d.

   A função deve retornar o número máximo de ligações que podem falhar simultaneamente sem comprometer a existência de um caminho entre o e d. **O corpo da sua função deve construir um grafo** (que pode ser igual a R) **e chamar um ou vários algoritmos de grafos estudados, como se eles estivessem numa biblioteca**, mesmo que esses algoritmos retornem resultados que não interessam para resolver este problema e, consequentemente, sejam menos eficientes do que poderiam ser para este caso. Em vez de programar a construção do grafo, pode indicar claramente que grafo construiria, usando a rede do exemplo para ilustrar a sua construção, e que estruturas de dados usaria para o implementar.

[1] Na realidade, o grafo também é pesado e os pesos dos arcos indicam a latência das ligações.