Complexidade de Algoritmos, 2020/2
Paulo Roberto Albuquerque
Prova Final

1) (1,5 ponto) Verifique se as afirmações abaixo são verdadeiras ou falsas, justifique as falsas.

a) $NP\text{-}Hard \cap NP\text{-}Completo = \varnothing$.

b) Se o problema da fatoração de inteiros grandes for resolvido em tempo polinomial então $P = NP$.

c) Se $P = NP$ então qualquer problema pode ser resolvido em tempo polinomial.

d) Considerando dois problemas A e B, se $A \leq_p B$ e $A \in P$ então $B \in P$.

e) Considerando dois problemas A e B, se $A \leq_p B$ e $B \in P$ então $A \in P$.

f) $NP\text{-}Hard \subseteq NP$.

R:

a) Falso, pois a intersecção destes dois conjuntos será no mínimo, NP-Completo, já que este está contido em NP-Hard. Além disso, existem problemas NP-Hard que não estão em NP-Completo.
b) Falso, pois não se sabe se ele é um problema NP-Completo ou não.
c) Falso, pois existem problemas fora de NP, que não poderiam ser resolvidos em tempo polinomial mesmo assim.
d) Falso, pois todo problema qualquer em P pode ser reduzido em tempo polinomial para outro problema em NP.
e) Verdadeiro.
f) Falso, pois existem problemas que estão em NP-Hard que não pertencem a NP.

2) (1,5 ponto) Resolva a seguinte relação de recorrência:

$$T(n) = T(n/2) + n^2$$
$$T(1) = 1$$

R:

$T(n) = T(n/2) + n^2$

$T(n/2) = T(n/2^2) + (n/2)^2$

$T(n/2^2) = T(n/2^3) + (n/2^2)^2$

$\vdots$

$T(n) = T(n/2^L) + T(n/2^{L-1})^2$

$T(n) = 1 + \sum_{i=0}^{L-1} (n/2^i)^2$

$T(n) = 1 + n^2 \sum_{i=0}^{L-1} (1/2^i)^2$

$T(n) = 1 + n^2 \sum_{i=0}^{L-1} (1/2^2)^i$

$T(n) = 1 + n^2 \sum_{i=0}^{L-1} (1/4)^i$

$T(n) = 1 + n^2 \sum_{i=0}^{L-1} (1/4)^i$

$\sum_{i=0}^{L-1} (1/4)^i = 1\frac{1-(1/4)^L}{1-(1/4)}$

$T(n) = 1 + n^2\frac{1-(1/4)^L}{3/4}$

$T(n) = 1 + n^2\frac{4(1-(1/4)^L)}{3}$

$T(n) = 1 + n^2\frac{4-4(1/4^L)}{3}$

$T(n) = 1 + n^2\frac{4-4^{1-L}}{3}$

$T(n) = 1 + \frac{4n^2-(4^{1-L})n^2}{3}$

$T(n) = 1 + \frac{4n^2-(4^{1-log_2 n})n^2}{3}$

3) (2,0 pontos) Complete (em linguagem C) a função de inserção em um *heap* binário máximo implementando a função **heapfyLeaf** que reestabelece a propriedade de *heap* quando um elemento é inserido. Qual a complexidade de tempo e espaço da função insertHeap considerando sua implementação?

```
int parent(int i)
{
  return ((i - 1) / 2);
}
```

/* Sendo ***a*** o vetor que implementa o *heap*, ***n*** o tamanho do *heap*, ***l*** o tamanho do vetor e **e** o elemento a ser inserido */

```
void insertHeap(int *a, int n, int l, int e)
{
        if (n < l)
        {
                a[n] = e;
                heapfyLeaf(a, n);

        }
        else
        {
                printf("insertHeap: Heap overflow!");
                exit(1);
        }
}
```

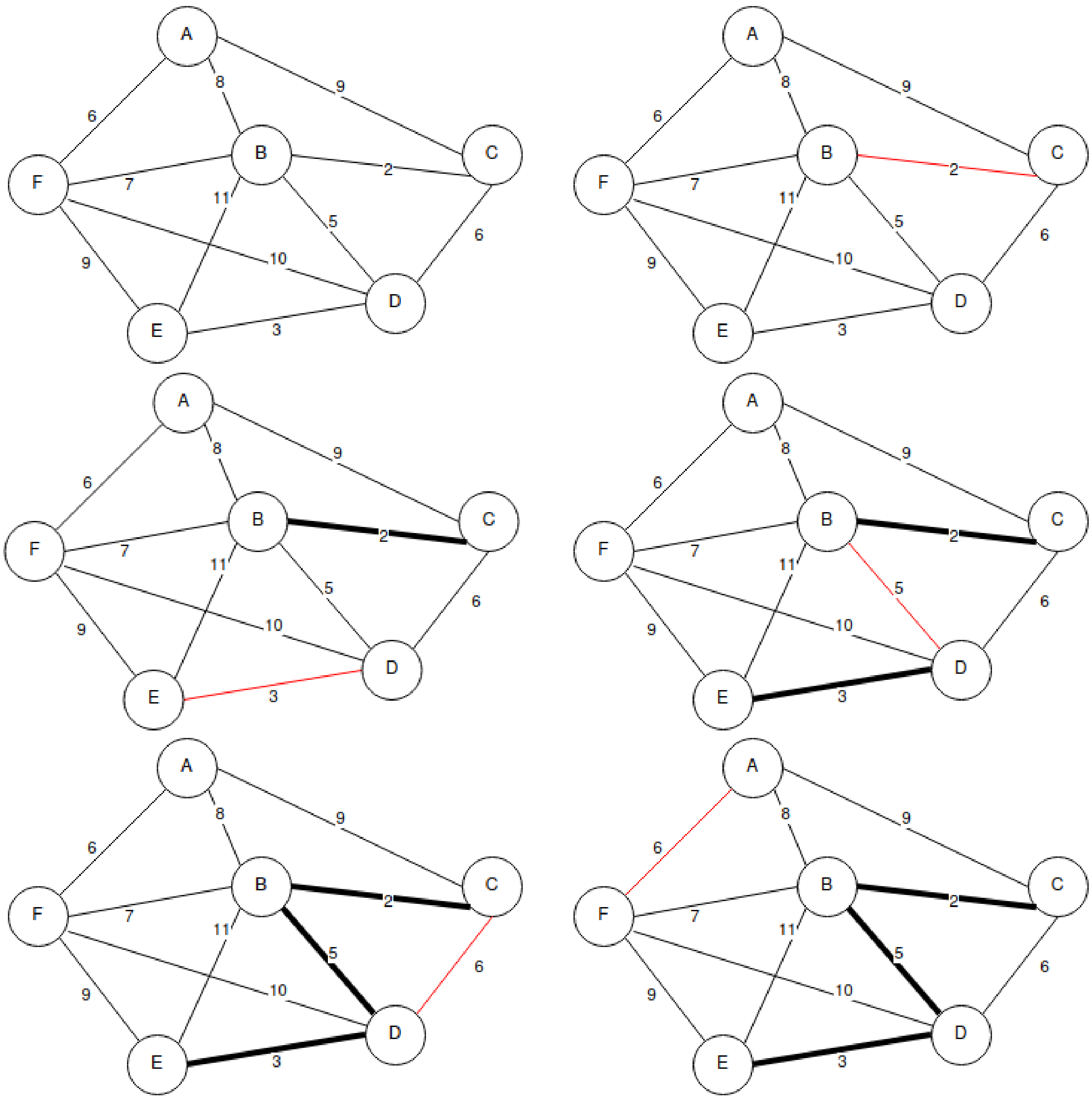
R:

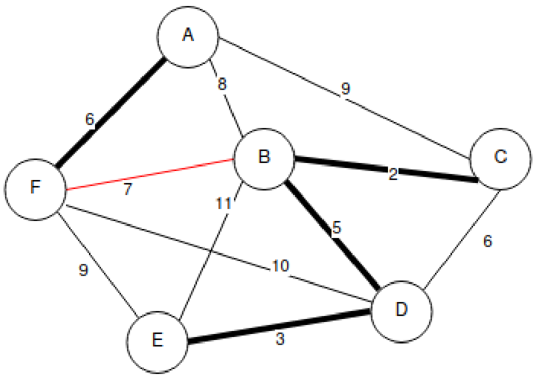
```
void heapfyLeaf(int *a,int n) {
      int parent = parent(n);
      if(a[n] > a[parent]) {
            swap(&a[n], &a[parent]);
            heapfyLeaf(a,parent);
      }
}
```

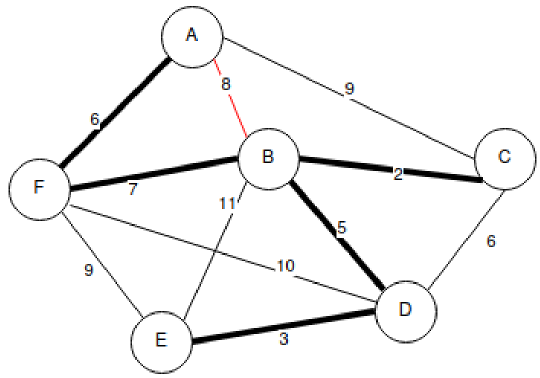
Complexidade de tempo: O(log n)
Complexidade de espaço: O(n)

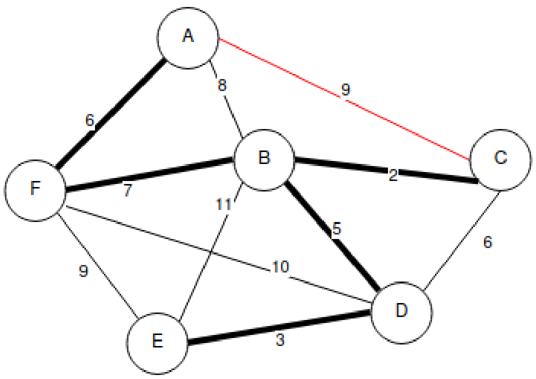
4) (2,0 pontos) Mostre cada passo da execução do algoritmo de *Kruskal* para encontrar a árvore geradora mínima do grafo abaixo. Mostre o algoritmo e a complexidade de tempo de cada *loop* e função auxiliar. Informe a complexidade de tempo do algoritmo.

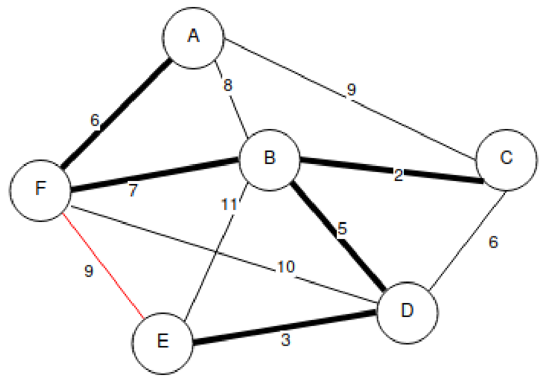
R:

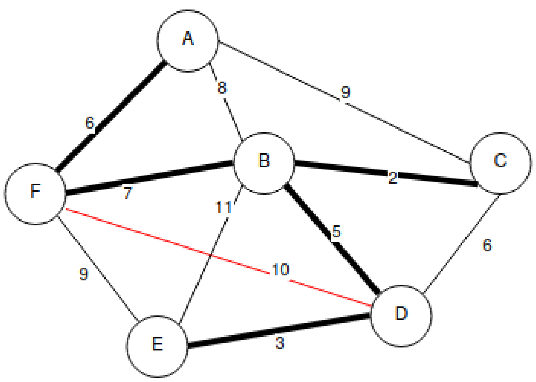


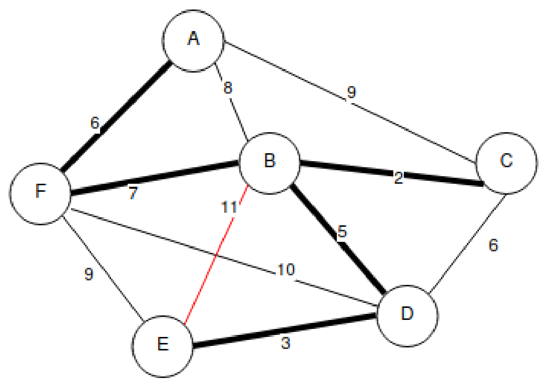

A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3

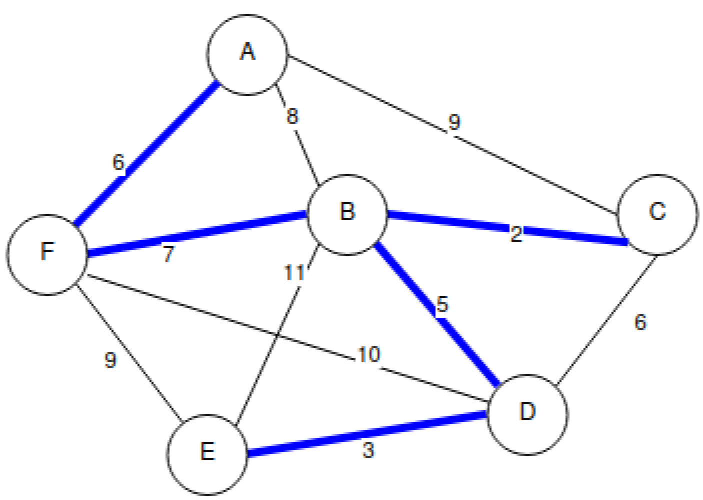

A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3


A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3


A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3


A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3


A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3


A
B
C
D
E
F
6
8
9
7
2
11
5
6
9
10
3

```
MST-KRUSKAL(G, w)
    A = ∅
    for cada vértice v ∈ G.V
        MAKE-SET(v)
    ordene as arestas de G.E em ordem não decrescente de peso w
    for cada aresta (u, v) ∈ G.E, tomada em ordem não decrescente de peso
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Usando a estrutura de dado disjoint-set, temos:

- `make-set(v) + find-set(v) + union(u,v)` → O((E + V) · α(V)) no total
- `for cada vértice v ∈ G.V` → O(V) pois `make-set` tem complexidade O(1)
- `ordene as arestas de G.E por peso w` → O(E log E) por meio de um sort de comparação
- `for cada aresta (u,v) ∈ G.E` → O(E log V), em E operações

A complexidade de tempo total do algoritmo é: O(E log E) = O(E log V)

5) (1,0 ponto) Mostre a redução do problema *2-CNF-SAT* ao *3-CNF-SAT*, usando como exemplo a instância de *2-CNF-SAT* abaixo. Essa redução demostra que *2-CNF-SAT* pertence à classe *NP-Hard*? Explique.

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

R:

6) (2,0 pontos) Um **conjunto independente de vértices** de um grafo simples $G = (V, A)$ é um subconjunto de vértices V' de G tal que não existem dois vértices adjacentes contidos em V'. Em outras palavras, se $u \in V'$ e $v \in V'$ então $(u, v) \notin A$. Prove que determinar se em um grafo existe um conjunto independente de vértices de tamanho k é um problema que pertence à classe NP.

R:

Para provar que o problema está em NP, basta provarmos que existe um algoritmo verificador em tempo polinomial que verifica uma instância do problema.

Começamos com um grafo G, e um certificado que é um conjunto de vértices.

Nosso algoritmo será dado por:

1. Verificar se todos os vértices do certificado são de fato vértices do grafo G.
2. Verificar que não existem arestas entre um par de vértices para todo par de vértices do certificado.

A complexidade do algoritmo dado é O(E + V), que se trata de uma complexidade polinomial. Então, o problema dos conjuntos de vértices independentes pertence à NP.