

---

## Estruturas Discretas - Segundo Trabalho

---

Gabriel Barbosa Diniz  
1511211

Lucas Rodrigues  
1510848

Mateus Ribeiro de Castro  
1213068

2 DE JULHO DE 2017

**Observação<sub>1</sub>:** Os códigos fontes dos algoritmos referentes aos teoremas provados seguirá em anexo em um arquivo Jupyter Notebook para melhor entendimento, compilação, execução, testes, etc.

### 1 PRIMEIRO TEOREMA

**ENUNCIADO DO TEOREMA :** Sabe-se encontrar a árvore de peso máximo de  $G = (V, E)$  que contém o vértice 1 e possui  $K$  vértices.

- Denomina-se  $A_k$  a árvore obtida com certo valor de  $k$ .
- Denomina-se  $V_k$  e  $E_k$  as listas de vértices e arestas, respectivamente, que compõem a árvore  $A_k$ .

**Caso Base:** Provando por indução simples em  $K$ , temos para o caso base  $k = 1$ , e assim haverá somente o vértice 1 e nenhuma aresta; o peso total será 0. Esta é a única árvore possível de 1 vértice e que contém  $v_1$ . Está definida por  $V_1 = \{v_1\}$  e  $E_1 = \emptyset$ .

**Hipótese Indutiva:** Pela hipótese indutiva, temos que o teorema é válido para  $k$  vértices e desejamos provar, portanto, que é válido também para  $k + 1$  vértices. Portanto, conhecemos  $V_k$  e  $E_k$ , e deseja-se determinar  $V_{k+1}$  e  $E_{k+1}$ .

**Passo Indutivo:** Considere o grafo  $B_k$  formado pelos vértices pertencentes a  $V - V_k$  e por todas as arestas formadas por vértices  $(b_1, b_2) \in (V - V_k)$ . Considere o conjunto  $R$  de arestas do tipo  $(a, b)$  em que  $a \in A_k$  e  $b \in B_k$ . Necessariamente,  $A_{k+1}$  tem seu conjunto de vértices

definido por  $V_k \cup \{b\}$  e seu conjunto de arestas definido por  $E_k \cup \{(a, b)\}$ . Determinando  $a$  e  $b$ , portanto, determinamos inteiramente  $A_{k+1}$ , onde  $a$  e  $b$  são os vértices da aresta de maior peso entre as arestas  $R$ . Com isso, está determinado  $A_{k+1}$ .

Com isso então podemos, através da prova indutiva resolvida derivar um algoritmo genérico que corresponde a prova deste teorema. Segue abaixo então o algoritmo em *python*:

```

1 import numpy as np
2
3 #####
4 #
5 # Algoritmo que recebe uma matriz de adjascencias com os pesos das arestas
6 #
7 # OBS: Ao inves de usar varios parametros, procure usar o retorno multiplo,
8 #       retornando o visited e a arvore. No caso, base tenta retornar uma matriz
9 #       vazia com o visited incluindo o primeiro vertice.
10 #
11 #####
12
13 def HeavyTree(M,k):
14
15     n = len(M)
16
17     if(k > n or k < 0):
18         print("K invalido")
19         return -1
20
21     # Caso Base
22     if(k == 1):
23         visited = [0]
24         tree = np.zeros((n,n))
25         return tree, visited
26
27     tree, visited = HeavyTree(M,k-1)
28     B = [] # Vertices vizinhos a vertices visitados
29
30     for i in range(n):
31         if(i not in visited): # Aqui temos a avaliacao de todos os vertices
32             for j in visited: # nao visitados que sao vizinhos de vertices
33                 if(M[i,j] != 0 and j not in B): # visitados.
34                     B = B + [i]
35                     break
36
37     biggest = 0
38     new_v = 0
39     old_v = 0
40
41     for i in B:
42         #new = np.max(M[i])
43         for j in visited:
44             if (M[i,j] > biggest):
45                 biggest = M[i,j]

```

```

46         new_v = i
47         old_v = j
48
49     visited = visited + [new_v]
50
51     #for i in range(n):
52     #    if (M[new_v,i] == biggest):
53     #        tree[i][new_v] = biggest
54     #        tree[new_v][i] = biggest
55     #        break
56
57     tree[new_v][old_v] = biggest
58     tree[old_v][new_v] = biggest
59
60     return tree, visited

```

Listing 1: Python algorithm

OBSERVAÇÃO SOBRE O ALGORITMO REALIZADO : Por questão prática do algoritmo em si, podemos dizer que foi realizado como passo indutivo:

- Procura-se  $\max(T(k))$ .
- Soma os pesos das arestas de maior peso de cada uma das folhas.
- Para todos os nós, adicionam-se as arestas de maior peso com os vértices restantes, ou seja,  $\max(p_{bt.qb} \in (V - V_k))$ .

Dessa forma, saberemos que adicionamos as arestas de maior peso, e à medida que se aumenta  $k$  basta incluirmos mais arestas de maior peso.

**Testes do Algoritmo:** A tabela abaixo ilustra o tempo de execução do algoritmo para diferentes instâncias e com diferentes valores do parâmetro  $k$ . O tempo foi medido executando o algoritmo por 5 segundos e contando o número de execuções.

## 1.1 TEOREMA BÔNUS

**ENUNCIADO DO TEOREMA BÔNUS** : Sabe-se encontrar a floresta de peso mínimo de  $G = (V, E)$  onde os componentes conexos possuem pelo menos  $k$  vértices.

**Caso Base:** Por indução simples em  $k$ . Para o caso base  $k = 1$ , a floresta  $F_1$  conterá todos os vértices de  $V$ , porém nenhuma aresta. Assim, haverá  $|V|$  componentes conexas e a soma dos pesos será mínima.

**Hipótese Indutiva:** Pela hipótese indutiva, temos que o teorema é válido para  $k$  vértices e desejamos provar, portanto, que é válido também para  $k + 1$  vértices. Podemos definir componente conexo como qualquer árvore  $A = (V', E')$  tal que  $V' \subset V$ ,  $E' \subset E$  e  $|E'| > 0$ . Pela hipótese indutiva, um componente conexo de  $F_k$  possuirá pelo menos  $k$  vértices.

**Passo Indutivo:** Sendo assim, o único modo de garantir que este componente passe a conter pelo menos  $k+1$  vértices é adicionando um novo vértice a este componente. Então, enquanto houverem componentes conexos em  $F_{k+1}$  com número de vértices menores que  $k + 1$ , devemos, do conjunto de arestas de  $G$  ainda não utilizadas em  $F_{k+1}$  (i.e.,  $S_k = E - E_{k+1}$ ), para um componente conexo  $A$  de  $F_{k+1}$ , escolher a aresta de menor peso  $s = (v_1, v_2) \in S_k$  tal que  $v_1 \in A$  e  $v_2 \notin A$ . Após a inclusão dessa aresta, o conjunto de componentes conexos deve ser re-avaliado. Desta maneira, todo componente conexo contará com pelo menos  $k + 1$  vértices e, assim, obteremos  $F_{k+1}$ , provando o teorema.

O algoritmo derivado desta prova indutiva é conhecido como *Algoritmo de Borůvka* e é utilizado para se obter a Árvore Geradora Mínima de grafos ponderados cujos pesos das arestas são distintos. Segue abaixo o algoritmo em *Python*:

```
1 from pygraph.classes.graph import graph
2 from pygraph.algorithms.accessibility import connected_components
3
4 def ForestTheorem(g, k):
5
6     # Salvaguarda
7     if k > len(g.nodes()):
8         raise ValueError('FORBIDDEN: K > |V|')
9
10    if k <= 0:
11        raise ValueError('FORBIDDEN: K <= 0')
12
13    # Caso base
14    if k == 1:
15        forest = graph()
16
17        for node in g.nodes():
18            forest.add_node(node)
19
20    return forest
21
```

```

22 # Hipotese indutiva
23 forest = ForestTheorem(g, k-1)
24
25 # Enquanto ainda houverem componentes conexos
26 # que nao satisfazem a condicao
27
28 while True:
29     # Atualiza a lista de componentes, pois pode ter
30     # mudado durante a adicao
31     cc = _transform_cc(connected_components(forest))
32
33     # Selecciona um que tenha comprimento < k
34     selected_component = None
35     for component in cc:
36         if len(component) < k:
37             selected_component = component
38             break
39
40     # Se nao conseguiu seleccionar, significa que todos
41     # satisfazem comprimento >= k, e podemos parar o while
42     if selected_component == None:
43         break
44
45     # Caso haja um selecionado, seleccionar a aresta de menor
46     # peso que tenha somente um dos vertices em selected_component
47     edges = g.edges()
48     used_edges = forest.edges()
49     unused_edges = [e for e in edges if e not in used_edges]
50     neighbor_edges = [e for e in unused_edges if e[0] in selected_component]
51
52     min_edge = min(neighbor_edges, key=lambda e: g.edge_weight(e))
53     forest.add_edge(min_edge)
54
55     return forest
56
57
58 def _transform_cc(cc):
59
60     inv_map = {}
61
62     for k, v in cc.iteritems():
63         inv_map[v] = inv_map.get(v, [])
64         inv_map[v].append(k)
65
66     return inv_map.values()

```

Listing 2: Python algorithm

**Testes do Algoritmo:** A tabela abaixo ilustra o tempo de execução do algoritmo, em milissegundos, para diferentes instâncias e com diferentes valores do parâmetro  $k$ .

- O tempo foi medido executando o algoritmo por 5 segundos e contando o número de execuções.

OBSERVAÇÃO SOBRE OS TESTES : No anexo enviado estão figuras que ilustram o passo-a-passo dos vértices e arestas escolhidos em cada etapa do algoritmo quando aplicado na instância ulysses16.

entrada	$k = 1$	$k = 2$	$k = 3$	$k = 5$	$k = 10$	$k = 15$	$k = 20$	$k = 30$	$k = 40$	$k = 50$
ulysses16	0.01	1.41	1.99	2.31	3.35	3.64	–	–	–	–
ulysses22	0.01	3.09	4.66	6.36	23.57	24.95	25.02	–	–	–
bays29	0.01	7.37	11.36	14.04	21.32	21.58	21.86	–	–	–
eil51	0.02	59.49	98.04	148.13	163.11	170.35	173.74	175.71	175.82	183.72
eil76	0.03	284	393	539	610	770	818	855	861	865
bier127	0.06	1861	3127	3917	6735	8142	10210	10352	10996	13077

## 2 SEGUNDO TEOREMA

ENUNCIADO DO TEOREMA  $(i, j, q)$ : Sabe-se determinar o prêmio máximo que o rei consegue coletar saindo da posição  $(i, j)$  e consumindo  $q$  unidades.

- Vamos considerar um desenvolvimento da matriz em 64 vértices distintos, com  $v_1$  correspondente a  $(1, 1)$ ,  $v_2$  a  $(1, 2)$ , assim por diante. Os conceitos de vizinhança continuam valendo:  $v_1$  tem como vizinhos  $\{v_2, v_9, v_{10}\}$ .
- Considere, também, uma tabela cujas linhas correspondem aos vértices  $v$ , e as colunas ao custo  $q$  restante a ser utilizado. As células da tabela serão preenchidas com o prêmio máximo  $P_{max}(v_{ij}, q)$ , que se consegue a partir de um trajeto que inicie no vértice  $v_{ij}$  e que consuma  $q$  unidades.

**Caso Base:** Por indução em  $q$ , temos o caso base para  $q = 0$ , preencheremos a primeira coluna da tabela. Neste caso, não existem unidades para consumir, logo não poderemos sair da origem  $(i, j)$ . Sendo assim, o prêmio máximo para ir até  $(i, j)$  será zero e para qualquer outro vértice será  $-\infty$  (que representa a impossibilidade).

**Hipótese Indutiva:** Como hipótese indutiva, temos que o teorema é válido para  $0 \leq q \leq Q$ , portanto queremos provar que o teorema também é válido para  $Q + 1$ .

**Passo Indutivo:** Neste caso, para cada um dos vértices  $v$ , devemos encontrar o prêmio máximo que pode ser obtido chegando a  $v$  consumindo  $Q + 1$  unidades. Logo, podemos observar que, para que a condição acima seja satisfeita, no instante imediatamente anterior à chegada em  $v$ , estaríamos em um vértice  $v_n$ , vizinho de  $v$ , com  $Q + 1 - q_v$  unidades consumidas, sendo  $q_v$  o custo associado ao vértice  $v$ . Visto que o prêmio  $p_v$  associado ao vértice  $v$  é constante, devemos escolher  $v_n$  de maneira que  $P_{max}(v_n, Q + 1 - q_v)$  seja máximo, garantindo, assim, que  $P_{max}(v, Q + 1) = p_v + P_{max}(v_n, Q + 1 - q_v)$  também seja máximo. Vale ressaltar que, caso  $Q + 1 - q_v < 0$ , teremos que  $P_{max}(v_n, Q + 1 - q_v) = -\infty$ , uma vez que é impossível chegar a qualquer vértice consumindo um custo total menor que zero.

E assim então, através da prova indutiva resolvida, podemos derivar um algoritmo genérico que corresponde a prova deste teorema. Segue abaixo o algoritmo em ***Python***:

### 3 ALGORITMO DE BUSCA DE UMA *String* EM TEXTOS

ENUNCIADO DO PROBLEMA : Considerando um texto definido pelo vetor de caracteres  $T[1..n]$ , deseja-se determinar todas as ocorrências da *string*  $s[1..m]$  em  $T$ .

- O algoritmo correspondente que possui este objetivo e que atenda à condição de "encontrar as ocorrências da string  $s$  exatamente como especificado" segue abaixo em **Python**:

```
1 def Transform(string):
2     n = len(string)
3     for i in range(n):
4         if(maiuscula(string[i])):
5             string[i] = chr(ord(string[i]) + 32)
6     return string
7
8 def FindWord (text, word):
9     i = 0
10    j = 0
11    f = len(text)
12    n = len(word)
13    letra = text[i]
14    instances = []
15    start = 0
16    end = 0
17    in_string = 0
18    while(i < f):
19        if(letra == word[0] and in_string == 0):
20            j = 0
21            start = i
22            end = i
23            in_string = 1
24        if(in_string == 1 and letra != word[j]):
25            in_string = 0
26            start = 0
27            end = 0
28            j = 0
29        if(in_string == 1 and letra == word[j] and j != (n-1)):
30            j = j + 1
31            end = end + 1
32        if(in_string == 1 and letra == word[j] and j == (n-1)):
33            instances = instances + [[start, end]]
34            start = 0
35            end = 0
36            in_string = 0
37        if(i == (f-1)):
38            break
39        i = i + 1
40        letra = text[i]
41    return instances
```

Listing 3: String Python Algorithm



- O algoritmo correspondente que possui este objetivo e que atenda à condição de "permitir alternativas entre elementos de  $s$ , como por exemplo, maiúsculas e minúsculas serem equivalentes" segue abaixo em **Python**:

```

1 def Transform(string):
2     n = len(string)
3     for i in range(n):
4         if(maiuscula(string[i])):
5             string[i] = chr(ord(string[i]) + 32)
6     return string
7
8 def FindWordEq (text, word):
9     text = Transform(text)
10    word = Transform(word)
11    i = 0
12    j = 0
13    f = len(text)
14    n = len(word)
15    letra = text[i]
16    instances = []
17    start = 0
18    end = 0
19    in_string = 0
20    while(i < f):
21        if(letra == word[0] and in_string == 0):
22            j = 0
23            start = i
24            end = i
25            in_string = 1
26        if(in_string == 1 and letra != word[j]):
27            in_string = 0
28            start = 0
29            end = 0
30            j = 0
31        if(in_string == 1 and letra == word[j] and j != (n-1)):
32            j = j + 1
33            end = end + 1
34        if(in_string == 1 and letra == word[j] and j == (n-1)):
35            instances = instances + [[start, end]]
36            start = 0
37            end = 0
38            in_string = 0
39        if(i == (f-1)):
40            break
41        i = i + 1
42        letra = text[i]
43    return instances

```

Listing 4: String Python Algorithm

- O algoritmo correspondente que possui este objetivo e que atenda à condição de "encontrar as ocorrências de  $s$  que deixam de verificar até um dos seus elementos" segue abaixo em **Python**:

```
1 def Transform(string):  
2     n = len(string)  
3     for i in range(n):  
4         if(maiuscula(string[i])):  
5             string[i] = chr(ord(string[i]) + 32)  
6     return string
```

Listing 5: String Python Algorithm