PERÍODO DE 2017.1
PROF. MARCUS VINICIUS S. POGGI DE ARAGÃO
PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Estruturas Discretas - Segundo Trabalho

Gabriel Barbosa Diniz 1511211 | Lucas Rodrigues 1510848 | Mateus Ribeiro de Castro 1213068

3 DE JULHO DE 2017

Observação$_1$: 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.

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. Segue abaixo então o algoritmo em ***python***:

```python
import numpy as np
from CPUtimer import *

# Algoritmo que recebe uma matriz de adjascencias com os pesos das arestas.
# OBS: Ao inves de usar varios parametros, procure usar o retorno multiplo,
#      retornando o visited e a arvore. No caso, base tenta retornar uma matriz
#      vazia com o visited incluindo o primeiro vertice.

def HeavyTree(M,k):

    n = len(M)
    if(k > n or k < 0):
        print("K invalido")
        return -1

    # Caso Base
    if(k == 1):
        visited = [0]
        tree = np.zeros((n,n))
        return tree, visited
    tree, visited = HeavyTree(M,k-1)
    B = [] # Vertices vizinhos a vertices visitados

    for i in range(n):
        if(i not in visited): # Aqui temos a avaliacao de todos os vertices
            for j in visited: # nao visitados que sao vizinhos de vertices
                if(M[i,j] != 0 and j not in B): # visitados.
                    B = B + [i]
                    break

```

```
    biggest = 0
    new_v = 0
    old_v = 0

    for i in B:
        for j in visited:
            if (M[i,j] > biggest):
                biggest = M[i,j]
                new_v = i
                old_v = j

    visited = visited + [new_v]
    tree[new_v][old_v] = biggest
    tree[old_v][new_v] = biggest

    return tree, visited
```

Listing 1: Python algorithm

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.

Caso de Teste	Valor de K	Vértices	Tempo de Execução
1	$K = 4$	[0, 1, 2, 3]	0.0001041032440411982
1	$K = 3$	[0, 1, 2]	0.000207473366629074
1	$K = 2$	[0, 1]	0.00028775016397730724
1	$K = 1$	[0]	0.00034273427178277416
-	-	-	-
2	$K = 4$	[0, 2, 1, 3]	0.00043400789064662604
2	$K = 3$	[0, 2, 1]	0.0005249149488122384
2	$K = 2$	[0, 2]	0.000622420099887222
2	$K = 1$	[0]	0.0006847354219985391
-	-	-	-
3	$K = 4$	[0, 3, 1, 2]	0.0007914045910411005
3	$K = 3$	[0, 3, 1]	0.0008848775742649195
3	$K = 2$	[0, 3]	0.001182891438304523
3	$K = 1$	[0]	0.0012448401997176006

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. Esse arquivo se chama: "***TEOREMA 1- ulysses16 - IMGs.pdf***".

1.1 TEOREMA BÔNUS

ENUNCIADO DO TEOREMA BÔNUS : Sabe-se encontrar a floresta de peso mínimo de 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 ***pseudocódigo***:

```
function ForestTheorem(V, E, K)
    se K == 1
        retorna uma floresta contendo todos os vertices, mas nenhuma aresta
    F <- ForestTheorem(V, E, K-1)

    enquanto houver componente conexa de F com |vertices| < K
        C <- uma componente conexa qualquer de F
        E <- aresta minima qualquer que nao pertence a F com um vertice em F
        adicionar E a F, inclusive seu vertice que nao estava em F
retorna F
```

A partir do enunciado foi retirado um código equivalente em ***python*** para realizar a prova do teorema e contudo realizar também os testes referentes ao mesmo, o referido codigo segue abaixo:

```python
from pygraph.classes.graph import graph
from pygraph.algorithms.accessibility import connected_components

def ForestTheorem(g, k):

  if k > len(g.nodes()):
    raise ValueError('FORBIDDEN: K > |V|')

```

```
  if k <= 0:
    raise ValueError('FORBIDDEN: K <= 0')

  # Caso base
  if k == 1:
    forest = graph()
    for node in g.nodes():
      forest.add_node(node)
    return forest

  # Hipotese indutiva
  forest = ForestTheorem(g, k-1)

  # Enquanto ainda houverem componentes conexos que nao satisfazem a condicao
  while True:

    # Atualiza a lista de componentes, pois pode ter mudado durante a adicao
    cc = _transform_cc(connected_components(forest))

    # Seleciona um que tenha comprimento < k
    selected_component = None
    for component in cc:
      if len(component) < k:
        selected_component = component
        break

    # Se nao conseguiu selecionar, significa que todos
    # satisfazem comprimento >= k, e podemos parar o while
    if selected_component == None:
      break

    # Caso haja um selecionado, selecionar a aresta de menor
    # peso que tenha somente um dos vertices em selected_component
    edges = g.edges()
    used_edges = forest.edges()
    unused_edges = [e for e in edges if e not in used_edges]
    neighbor_edges = [e for e in unused_edges if e[0] in selected_component]

    min_edge = min(neighbor_edges, key=lambda e: g.edge_weight(e))
    forest.add_edge(min_edge)

  return forest


def _transform_cc(cc):
  """
  The "connected components" structure returned
  by the function in pygraph is a dict mapping each
  node to an id.
  We'll make a new structure which is a list of lists
  of nodes that are in the same connected component.
  """
  inv_map = {}
  for k, v in cc.iteritems():
```

```
63      inv_map[v] = inv_map.get(v, [])
64      inv_map[v].append(k)
65    return inv_map.values()
```

Listing 2: Python algorithm

Testes do Algoritmo: A tabela abaixo ilustra o tempo de execução do algoritmo retirado a partir do pseudocódigo, 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.

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 desarrolamento 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 impossivel 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***:

```
import numpy as np
from CPUtimer import *

def Pos(casa):
    x = int(casa / 8)
    y = int(casa % 8)
    return x,y

def Casa(x,y):
    casa = 8 * x + y
    return casa

def Vizinhos(casa):
    vizinhos = []
    x,y = Pos(casa)
    if(x > 0):
        neighbour = Casa(x-1,y)
        vizinhos = vizinhos + [neighbour]
    if(x < 7):
        neighbour = Casa(x+1,y)
        vizinhos = vizinhos + [neighbour]
    if(y > 0):
        neighbour = Casa(x,y-1)
        vizinhos = vizinhos + [neighbour]
    if(y < 7):
        neighbour = Casa(x,y+1)
        vizinhos = vizinhos + [neighbour]
    if(x > 0 and y > 0):
        neighbour = Casa(x-1,y-1)
        vizinhos = vizinhos + [neighbour]
    if(x < 7 and y > 0):
        neighbour = Casa(x+1,y-1)
        vizinhos = vizinhos + [neighbour]
    if(x > 0 and y < 7):
        neighbour = Casa(x-1,y+1)
        vizinhos = vizinhos + [neighbour]
    if(x < 7 and y < 7):
        neighbour = Casa(x+1,y+1)
        vizinhos = vizinhos + [neighbour]
    return vizinhos

def Award(Vertex,Prize,Weight,Q):
```

```
    if (Q < 0):
        print("Combustivel invalido")
        return
    if (Q == 0 and Vertex == 0):
        path = [0]
        prize = 0
        return path, prize
    elif (Q == 0 and Vertex != 0):
        path = []
        prize = -9999999999
    vizinhos = Vizinhos(Vertex)
    vx,vy = Pos(Vertex)
    max_prize = 0
    max_path = []
    for neighbour in vizinhos:
        if(Q-Weight[vx][vy] >= 0):
            path, prize = Award(neighbour,Prize,Weight,Q-Weight[vx][vy])
            if(len(path) > 0):
                if(path[0] == 0):
                    prize = prize + Prize[vx][vy]
                    path = path + [Vertex]
                    if(prize > max_prize):
                        max_prize = prize
                        max_path = path
    return max_path, max_prize
```

Listing 3: Python algorithm

Testes do Algoritmo: A tabela abaixo ilustra o tempo de execução do algoritmo, em milissegundos, para diferentes valores. O tempo foi medido executando o algoritmo por 5 segundos e contando o número de execuções.

Caso de Teste	Valor de Q	Prêmio	Caminho	Tempo de Execução
1	$Q = 8$	16	[0,8,0,8,0,8,0,8,0]	10.543801256643519
2	$Q = 8$	16	[0,8,0,8,0,8,0,8,0]	21.630656352625465
3	$Q = 8$	16	[0,8,0,8,0,8,0,8,0]	31.6944151908624
4	$Q = 10$	16	[0,9,18,10,18,10,18,10,18,9,0]	576.7521444718427

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 ocorrêcias da *string* $s[1..m]$ em T.

- O algoritmo correspondente que possui o objetivo descrito no enunciado e que também atenda à condição de "encontrar as ocorrências da string s exatamente como especificado"segue abaixo em ***Python***:

```
def Transform(string):
    n = len(string)
    for i in range(n):
        if(maiuscula(string[i])):
            string[i] = chr(ord(string[i]) + 32)
    return string

def FindWord (text, word):
    i = 0
    j = 0
    f = len(text)
    n = len(word)
    letra = text[i]
    instances = []
    start = 0
    end = 0
    in_string = 0
    while(i<f):
        if(letra == word[0] and in_string == 0):
            j = 0
            start = i
            end = i
            in_string = 1
        if(in_string == 1 and letra != word[j]):
            if(letra == word[0]):
                j = 0
                start = i
                end = i
            else:
                in_string = 0
                start = 0
                end = 0
                j = 0
        if(in_string == 1 and letra == word[j] and j != (n-1)):
            j = j + 1
            end = end + 1
        if(in_string == 1 and letra == word[j] and j == (n-1)):
            instances = instances + [[start,end]]
            start = 0
            end = 0
            in_string = 0
        if(i == (f-1)):
            break
        i = i + 1
        letra = text[i]
    return instances
```

Listing 4: String Python Algorithm

Testes do Algoritmo: A tabela abaixo ilustra o tempo de execução do algoritmo, em milissegundos, para diferentes testes para uma ***busca exata***. Para mais informações vide arquivo ***Jupyter Notebook***!

Caso de Teste	Tempo de Execução
1	0.0001066691693267785
-	-
2	0.0005776996927124856
2	0.0007565813230030471
-	-
3	0.00222978884994518
3	0.0026381374900665833

- 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***:

```
def Transform(string):
    n = len(string)
    for i in range(n):
        if(maiuscula(string[i])):
            string[i] = chr(ord(string[i]) + 32)
    return string

def FindWordEq (text, word):
    text = Transform(text)
    word = Transform(word)
    i = 0
    j = 0
    f = len(text)
    n = len(word)
    letra = text[i]
    instances = []
    start = 0
    end = 0
    in_string = 0
    while(i<f):
        if(letra == word[0] and in_string == 0):
            j = 0
            start = i
            end = i
            in_string = 1
        if(in_string == 1 and letra != word[j]):
            if(letra == word[0]):
                j = 0
                start = i
                end = i
            else:
                in_string = 0
```

```
                start = 0
                end = 0
                j = 0
        if(in_string == 1 and letra == word[j] and j != (n-1)):
            j = j + 1
            end = end + 1
        if(in_string == 1 and letra == word[j] and j == (n-1)):
            instances = instances + [[start,end]]
            start = 0
            end = 0
            in_string = 0
        if(i == (f-1)):
            break
        i = i + 1
        letra = text[i]
    return instances
```

Listing 5: String Python Algorithm

Testes do Algoritmo: A tabela abaixo ilustra o tempo de execução do algoritmo, em milissegundos, para diferentes testes para uma ***busca com equivalência***. Para mais informações vide arquivo ***Jupyter Notebook***!

Caso de Teste	Tempo de Execução
1	0.0002371647851759917
-	-
2	0.0010501964584364032
2	0.001350043125967204
-	-
3	0.0033782235805119853
3	0.004232310054703703

- 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***:

```
def maiuscula(letter):
    if(ord(letter) >= 65 and ord(letter) <= 90):
        return 1
    return 0

def Transform(string):
    n = len(string)
    for i in range(n):
        if(maiuscula(string[i])):
            string[i] = chr(ord(string[i]) + 32)
    return string

def FindWordError (text, word):
    i = 0
```

```
    j = 0
    error = 0
    f = len(text)
    n = len(word)
    letra = text[i]
    instances = []
    start = 0
    end = 0
    in_string = 0
    while(i<f):
        if(letra == word[0] and in_string == 0):
            j = 0
            start = i
            end = i
            in_string = 1
            error = 0
        if(in_string == 1):
            if(letra != word[j] and error == 1):
                if(letra == word[0]):
                    j = 0
                    start = i
                    end = i
                    error = 0
                    in_string = 1
                else:
                    in_string = 1
                    start = i
                    end = i
                    j = 0
                    error = 1
            elif(letra == word[j] and j == (n-1)):
                instances = instances + [[start,end]]
                start = 0
                end = 0
                error = 0
                in_string = 0
                j = 0
            elif(letra == word[j] and j != (n-1)):
                j = j + 1
                end = end + 1
            elif(letra != word[j] and error == 0 and j != (n-1)):
                j = j + 1
                end = end + 1
                error = 1
        if(i == (f-1)):
            break
        i = i + 1
        letra = text[i]
    return instances
```

Listing 6: String Python Algorithm

Testes do Algoritmo: A tabela abaixo ilustra o tempo de execução do algoritmo, em milissegundos, para diferentes testes para uma ***busca com tolerância a erro***. Para mais informações vide arquivo ***Jupyter Notebook***!

Caso de Teste	Tempo de Execução
1	0.0002884832856580033
-	-
2	0.0015839088641769195
2	0.0018269386205247429
-	-
3	0.00484849862232295
3	0.005450757882954349