Estruturas Discretas - Trabalho 2

Felipe Luiz Gabriel Siqueira Guilherme Berger

12 de Junho de 2014

Teorema 1

Teorema e Prova

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

Prova Por indução simples em k.

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

Para o caso base $k = 1$, 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 $E_1 = \emptyset$.

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}.

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}.

a e b são os vértices da aresta de maior peso entre as arestas R.

Com isso, está determinado A_{k+1}.

Algoritmo e Código

Abaixo, mostramos o algoritmo em **pseudocódigo**:

```
função t1(G, K)

    se K == 1
        retorna uma árvore contendo somente o vértice 1

    A <- t1(G, K-1)
    B <- grafo formado pelos vertices de G nao presentes em A, e por todas as arestas entre esses ve
    R <- arestas do tipo (a,b) onde ( pertence(a, A) && pertence(b, B))

    new_edge <- elemento de R com maior peso
    a, b <- vertices da aresta new_edge

    adiciona vertice (b) ao grafo (A)
    adiciona aresta (a, b) ao grafo (A)

    retorna A
```

Abaixo, mostramos o algoritmo implementado em **Python**:

```python
from pygraph.classes.graph import graph

def teo_1(g, k):
  # Salvaguarda
  if k > len(g.nodes()):
    raise ValueError('FORBIDDEN: K > |V|')
  if k <= 0:
    raise ValueError('FORBIDDEN: K <= 0')

  # Caso base
  if k == 1:
    tree = graph()
    tree.add_node(1)
    return tree

  # Hipotese indutiva
  tree = teo_1(g, k-1)

  # V - V_k
  all_nodes = g.nodes()
  used_nodes = tree.nodes()
  external_nodes = [node for node in all_nodes if node not in used_nodes]

  # Conjunto F de arestas possiveis
  r = []
  for used_node in used_nodes:
    for external_node in external_nodes:
      if g.has_edge((used_node, external_node)):
        r.append((used_node, external_node))

  # Aresta de maior peso em F
  new_edge = max(r, key=lambda e: g.edge_weight(e))

  # V_{k+1} = V_k + {b}
  a, b = new_edge
  tree.add_node(b)

  # E_{k+1} = E_k + {(a, b)}
  tree.add_edge(new_edge)

  return tree
```

Testes e Resultados

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.

A leitura do arquivo e criação da árvore que o representam não foram incluídos no loop.

entrada	$k=1$	$k=2$	$k=3$	$k=5$	$k=10$	$k=15$	$k=20$	$k=30$	$k=40$	$k=50$	$k=\|V\|$
ulysses16	0.003	0.031	0.069	0.184	0.554	0.979	–	–	–	–	0.989
ulysses22	0.004	0.037	0.113	0.261	0.890	1.74	2.12	–	–	–	2.16
bays29	0.004	0.048	0.121	0.339	1.27	2.53	4.26	–	–	–	5.71
eil51	0.004	0.074	0.194	0.587	2.38	5.10	8.60	16.2	26.4	33.1	33.3
eil51	0.004	0.074	0.194	0.587	2.38	5.10	8.60	16.2	26.4	33.1	33.3
bier127	0.006	0.171	0.489	1.81	9.20	23.6	40.0	85.6	147.7	206.4	680.2
lin318	0.006	0.386	1.13	3.77	17.1	51.8	116	266	467	722	9837

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.

Teorema 2 (Bônus)

Teorema e Prova

Teorema *Sabe-se encontrar a floresta de peso mínimo de de $G = (V, E)$ onde os componentes conexos possuem pelo menos K vértices*

Prova 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.

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. 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.

Algoritmo e Código

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.

Abaixo, mostramos o algoritmo em **pseudocódigo**:

```
função t2(V, E, K)

    se K == 1
        retorna uma floresta contendo todos os vértices, mas nenhuma aresta

    F <- t2(V, E, K-1)

    enquanto houver componente conexa de F com |vertices| < K
        C <- uma componente conexa qualquer de F
        E <- aresta mínima qualquer que não pertence a F com um vértice em F
        adicionar E a F, inclusive seu vértice que não estava em F

    retorna F
```

Abaixo, mostramos o algoritmo implementado em **Python**:

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

def teo_2(g, k):
  # Salvaguarda
  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 = teo_2(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():
      inv_map[v] = inv_map.get(v, [])
      inv_map[v].append(k)
    return inv_map.values()
```

Testes e Resultados

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.

A leitura do arquivo e criação da árvore que o representam não foram incluídos no loop.

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

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.

Teorema 3

Teorema e Prova

Teorema 3 *(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.

Prova Por indução em q. Para o caso base $q = 0$, 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). Como hipótese indutiva, temos que o teorema é válido para $0 \leq q \leq Q$, e queremos provar que é válido também para $Q+1$. 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.

Algoritmo e Código

Abaixo, mostramos o algoritmo em **pseudocódigo**:

```
função caminhoDePremioMax(v, q)

    se q é zero
        se v é origem
            return caminho(v)
        caso contrario
            return "caminho impossivel"

    se q < 0
        return "caminho impossivel"

    Vn ⇐ conjunto de vizinhos de v

    caminho ⇐ maxPremio{ caminhoDePremioMax( vn ∈ Vn, q-custo(v) )

    caminho.adicionaAoFinal(v)

    return caminho
```

Abaixo, mostramos o algoritmo em **Python**:

```python
def teo_3(pos, costs, rewards, energy):
  # Salvaguarda
  if pos < 0 or pos >= 64 or energy < 0:
    raise ValueError("Invalid position/energy!!")

  # Dict cujas chaves sao tuplas (posicao, energia) e cujos valores
  # sao tuplas contendo o premio maximo que o rei consegue coletar
  # comecando da posicao dada, com dada energia disponivel, e parando na
  # posicao 0 com 0 energia, e o caminho para tal
  memo = {}

  # Caso base -- q=0
  memo[(0, 0)] = (0, [0])
  for v in range(1, 64):
    memo[(v, 0)] = None

  # Hipotese indutiva e passo indutivo -- preencher a tabela
  # Para cada coluna de energia
  for q in range(1, energy+1):
    # Para cada vertice nessa coluna
    for v in range(64):
      # custo_vizinho e a coluna em que vamos olhar
      custo_vizinho = q - costs[v]
      vizinhos = find_neighbors(v)
      if custo_vizinho < 0:
        memo[(v, q)] = None
      else:
        # Filtra as celulas -- somente se nao for impossivel (not None)
        # e pega a tupla (premio, caminho) delas
        tuplas = [memo[(vizinho, custo_vizinho)] for vizinho in vizinhos if not memo[(vi
        if len(tuplas) > 0:
```

```
            # O melhor_vizinho e o que tem maior premio
            melhor_vizinho = max(tuplas, key=lambda x: x[0])
            # O novo_premio e o premio do melhor vizinho somado ao do vertice em questao
            novo_premio = melhor_vizinho[0] + rewards[v]
            # O novo_caminho e o caminho do melhor vizinho acrescido do vertice em questao
            novo_caminho = melhor_vizinho[1][:] + [v]
            memo[(v, q)] = (novo_premio, novo_caminho)
          else:
            memo[(v, q)] = None

    # Com a tabela em maos, vamos encontrar o caminho comecando em 0
    # que obtenha o maior premio possivel, utilizando qualquer quantidade
    # de energia menor que a fornecida
    maior = 0
    for q in range(energy, -1, -1):
      x = memo[(0, q)]
      if x != None and x[0] > maior:
        maior = x[0]
        inst = x + (q,)
    return inst


def find_neighbors(pos):
    x = pos/8
    y = pos%8

    naive = [(x-1, y-1), (x-1, y), (x-1, y+1),
             (x,   y-1),           (x,   y+1),
             (x+1, y-1), (x+1, y), (x+1, y+1)]
    filtered = [n for n in naive if n[0]>=0 and n[1]>=0 and n[0]<8 and n[1]<8]

    return map(lambda pos: pos[0]*8+pos[1], filtered)
```

Testes e Resultados

Testamos o algoritmo nas instâncias fornecidas. Os resultados encontram-se abaixo, e também em anexo no arquivo walk.out no formato pedido.

instância	tempo	prêmio obtido	energia utilizada	energia disponível	caminho encontrado
1ª	3.70ms	16	8	8	0 1 0 1 0 1 0 1 0
2ª	3.52ms	16	8	8	0 1 0 1 0 1 0 1 0
3ª	3.97ms	16	8	8	0 1 0 1 0 1 0 1 0
4ª	14.22ms	284	22	22	0 9 18 27 36 44 52 60 52 60 52 60 52 60 52 60 52 44 36 27 18 9 0
5ª	9.28ms	57	18	18	0 8 16 25 16 25 16 25 16 25 16 25 16 25 16 8 0