6º Lista de PAA - Fabio Alves

1º)

A matriz de incidência de u grafo dirigido $G = (V, E)$, sem auto laço, é uma matriz $B = (b_{ij})$, $|V| \times |E|$, tal que

$$b_{ij} = \begin{cases} -1, & \text{Se a aresta } j \text{ sai do vértice } i \\ 1, & \text{Se a aresta } j \text{ entra no vértice } i \\ 0, & \text{Se demais casos} \end{cases}$$

Descreva o que as entradas do produto BB^T representa, onde B^T é a transposta de B.

Resposta:

Digamos que temos o seguinte grafo:

$$A \rightarrow_{ab} B \rightarrow_{bc} C$$

A matriz de incidência ficaria da seguinte forma:

$$B = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$$

O produto pela matriz tranposta seria:

$$BB^T = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

Podemos observar a matriz resultante BB^T relaciona o o comportamento das arestas num vértice i com o comportamento das arestas num vértice j. Neste contexto há dois cenário a serem analiados, quando relacionamos o vértice com ele mesmo ($i = j$) e quando relacionamos dois vértices diferentes ($i \neq j$).

- Se $i = j$, então estaremos operando duas linhas equivalentes, multiplicando entradas de mesmo valor, ou seja, sempre teremos 1^2, -1^2 ou 0^2. O 1 ou -1 aparecerão quando houver uma aresta saindo ou entrando no vértice atual, e o 0 quando não houverem arestas. Ao fim da operação teremos que o valor resultante representa o número de arestas entrando ou saindo do vértice.
- Se $i \neq j$, então estaremos operando duas linhas distintas. Se uma dada aresta estiver entrando no vértice i (1) e saindo do vértice j (-1), então o resultado da operação será -1. Se uma dada aresta estiver entrando ou saindo no vértice i (1 ou -1) mas não entrar

nem sair vértice j (0), então a operação será 0. Desta forma, quando $i \neq j$, a operação resultará em -1 multiplicado pelo número de arestas conectadas a i e j simultaneamente.

2º)

O diâmetro de uma árvore $T = (V, E)$ é definido como $max_{u,v \in V} \delta(u, v)$, que é, o maior valor de todos os percusos mais curtos da árvore. Dê um algoritmo eficiente para computar o diâmetro de uma árvore e analise o seu custo em tempo.

Resposta:

A partir da busca em largura:

É possível calcular o diametro da árvore por meio do algoritmo de busca em profundidade (BFS). Primeiramente consideramos um nodo x como a raiz e executamos o BFS em busca do nodo mais distante dele (y). Executamos novamente a BFS, considerando y como raiz, em busca do nodo mais distante dele (z). Calculamos a distância do percurso entre y e z e este será o diâmetro da árvore. O custo do BFS para analisar toda a árvore é $O(|V| + |E|)$. O custo para calcular o percurso entre dois nodos numa árvore é $O(|V|)$, logo temos que o custo desse algoritmo é dado por:

$$2(|V| + |E|) + |V|$$
$$2|V| + |V| + 2|E|$$
$$3|V| + 2|E|$$
$$O(|V| + |E|)$$

Alternativa Recursiva:

Também é possível calcular o diâmetro da árvire recursivamente. Há três casos possíveis: o diâmetro está contido na subárvore esquerda; o diâmetro está contido na subárvore direita; ou o diâmetro é soma das alturas das subárvores esquerda e direita mais 1.

O algoritmo recebe o nodo raiz da árvore (R) e sua altura, que consideramos inicialmente ser 0. Checamos o caso base, onde caso o nodo atual for NIL, retornamos seu diâmetro e altura como 0. Caso contrário, fazemos duas chamadas recursosvas passando como parâmetro as raizes das subárvores direita $(R.right)$ e esquerda $(R.left)$, também considerando que a altura delas inicialmente é 0. Após o retorno das chamadas recursivas, recebemos o diâmetro e a altura das respectivas subárvores. Sejam ld e lh o diâmetro e altura da subárvore esquerda, respectivamente, e rd e rh o diâmetro e altura da subárvore direita, respectivamente.

Calculamos altura da árvore de raiz R pela equação abaixo:

$$rootHeight = max(lh, rh) + 1$$

Calculamos o diâmetro da árvore de raiz R pela equação abaixo:

$$rootDiameter = max(lh + rh + 1, ld, rd)$$

Para finalizar o algoritmo, retornamos $rootDiameter$ e $rootHeight$ como os valores finais do diâmetro e altura da árvore inicial. O custo deste algorito é da ordem de $O(|V|)$, uma vez que visitamos cada nodo da árvore uma única vez.

Abaixo segue a implementação da solução recursiva

```
class Node:
  def __init__(self, data):
    self.data = data
    self.left = self.right = None

def treeDiameter(root, height = 0):
  if root is None:
    return 0, 0 # diametro atual e altura da árvore atual

  rdiameter, rheight = treeDiameter(root.right)
  ldiameter, lheight = treeDiameter(root.left)
  height = max(lheight, rheight) + 1
  diameter = max(lheight + rheight + 1, ldiameter, rdiameter)
  return diameter, height

"""
            1
          /   \
        2      3
      /  \
    4     5
"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print(treeDiameter(root)[0])

"""
            1
          /   \
        2      3
      /  \
     4    5
    /      \
   6        7
  /          \
 8            9

"""
```

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.right.right = Node(7)
root.left.right.right.right = Node(9)
root.left.left.left = Node(6)
root.left.left.left.left = Node(8)


print(treeDiameter(root)[0])
```

```
    4
    7
```

3º)

Um grado direcionado $G = (V, E)$ é simplismente conectado se $u \rightsquigarrow v$ implicar que G contem no máximo um único percursos simples de u até v para todo vértice $u, v \in V$. Dê um algoritmo eficiente para determinar se um grafo é ou não simplismente conectado.

Resposta:

Isso pode ser feito na ordem $O(|V||E|)$. Para fazer isso, primeiro execute uma classificação topológica dos vértices. Então, iremos conter para cada vértice uma lista de seus ancestrais com grau 0. Calculamos essas listas para cada vértice na ordem, começando pelos anteriores topologicamente. Então, se alguma vez tivermos um vértice com o mesmo grau 0 vértice aparecendo nas listas de dois de seus pais imediatos, sabemos que o gráfico não está conectado individualmente. entretanto, se em cada etapa temos que em cada etapa todos os pais têm conjuntos disjuntos de vértices de grau 0 como ancestrais, o gráfico é conectado individualmente. Visto que, para cada vértice, a quantidade de tempo necessária é limitada pelo número de vértices vezes o grau do vértice particular, o tempo de execução total é limitado por $O(|V||E|)$.

4º)

Suponha que os pesos das arestas são uniformementes distribuídos sobre o intervalo $[0, 1)$. Qual algoritmo, de Kruskal ou Prim, roda mais rápido? Suponha que ambos os algoritmos rodem na mesma máquina sobre as mesmas condições físicas.

Resposta:

Para dados extraídos de uma distribuição uniforme, eu usaria o bucket-sort com o algoritmo de Kruskal, para classificação linear esperada de arestas por peso. Isso alcançaria o tempo de execução esperado $O(E\alpha(V))$.

5º)

Mostre que o algoritmo de Dijkstra é um algoritmo ótimo para a determinação de percursos mais curtos em grafos dirigidos com pesos não negativos.

Resposta:

De acordo com o teorema 24.6 do capitulo 24 do Coromen, o algorítmo de Dijkstra é correto, por tanto *sempre* encontrará o menor caminho possível para qualquer nó em um Grafo G = (V,E). Assim sendo, sempre correto e sempre encontrado a solução ótima (menor caminho *possível*. Uma vez que não exista nenhum outro caminho C' que seja melhor que o caminho C encontrado pelo algorítmo de Dijkstra, averia uma contradição, uma vez que C' seria o melhor caminho então. Isso se trata de um absurdo e C' não pode ser o melhor caminho possível, a menos que C' = C.

Quanto ao custo do algorítmo Dijkstra, apenas seria possível melhora-lo (otimizá-lo) em relação ao tempo de execução se existisse alguma maneira de encontrar o nó menos distânte sem precisar percorrer todos os nós adjacentes de u no for (linha 8, acima). Tal terefa no momento seria apenas possível, talvez, utilizando superposição (computação quântica), onde talvez seria possível criar um algoritmo quântico (sort) que encontrasse de imediato, o nó adjacente a u com menor peso, sem percorrer cada nó adjacente um à um. No que compete a máquina de Turing, o algorítmo de Dijkstra é ótimizado e correto, sempre encontrado a melhor resposta, no melhor tempo de execução possível para uma máquina de Turing.

✓ 0s conclusão: 22:09