CPS740 - Algoritmos e Grafos - Lista 2
Thiago Guimarães Rebello Mendonca de Alcantara - DRE: 118053123
Tudo esta no repositorio https://github.com/guim4dev/CPS740

Questão 1)

a) *Complexidade: O(2^n)*

```
# P = limite
# n = lista de objetos
# i = iterador (algoritmo eh recursivo)
# objeto = [p, v], sendo p = peso e v = valor
def greedy_knapsack(P, n, i = -1, final_array = []):
        if i == -1: # setar iterador na primeira chamada conforme tamanho do array de itens
                i = max((len(n) - 1), 0)

        if i == 0 or P == 0: # caso base
                return [0, final_array]

 # máximo entre dois casos:
        if n[i][0] > P: # peso do item > capacidade do knapsack - nao podemos incluir
                return greedy_knapsack(P, n, i-1)
        else: # máximo entre dois casos:
                # item atual incluído  e item atual nao incluido
                included_call = greedy_knapsack(P-n[i][0], n, i-1, final_array + [n[i]])
                included = n[i][1] + included_call[0] # valor máximo quando incluido este item
                not_included_call = greedy_knapsack(P, n, i-1, final_array)
                not_included = not_included_call[0] # valor máximo quando não incluído

                if included > not_included:
                        return [included, included_call[1]]
                else:
                        return not_included_call

coisas = [[10, 60], [20, 100], [30, 120], [80, 1000], [20, 100]]
P = 80
print(greedy_knapsack(P, coisas)) # retorna array com valor dentro do knapsack e array dos objetos dentro do knapsack
```

b) *Complexidade Tempo: O(n*P) - n: número de elementos, P: capacidade desejada; Complexidade Espaço: O(n*P) - matriz auxiliar criada*

```
dp = []
def dp_knapsack(P, n, i = -1):
 global dp
 if i == -1:
  i = len(n)

 if dp == []: # montar tabela de Capacidades X Itens
  dp = [[0 for x in range(P+1)] for y in range(len(n)+1)] # montar tabela de Capacidades X
Itens

 if i == 0 or P == 0: # caso base
  result = 0
 elif dp[i-1][P] != 0: # aproveitar a memoizacao de operacoes
  return dp[i-1][P]
 elif n[i-1][0] > P:
  result = dp_knapsack(P, n, i-1)
 else:
  included = n[i-1][1] + dp_knapsack(P - n[i-1][0], n, i-1)
  not_included = dp_knapsack(P, n, i-1)

  result = max(included, not_included)
 dp[i][P] = result
 return(result)

def get_dp_knapsack_array(P, n):
 result = dp_knapsack(P, n)
 res = result
 items = []
 p = P
 for i in range(len(n), 0, -1):
  if res <= 0:
   break
  # resultado vem de cima dp[i-1][p]
  # ou de (n[i-1][1] + dp[i-1][p] na tabela Knapsack.
  # Se vem do segundo, o item foi incluído.
  if res == dp[i - 1][p]:
   continue
  else:
   # Item incluído
   items.append(n[i-1])
   res = res - n[i - 1][1]
   p = p - n[i - 1][0]
 return(result, items)
```

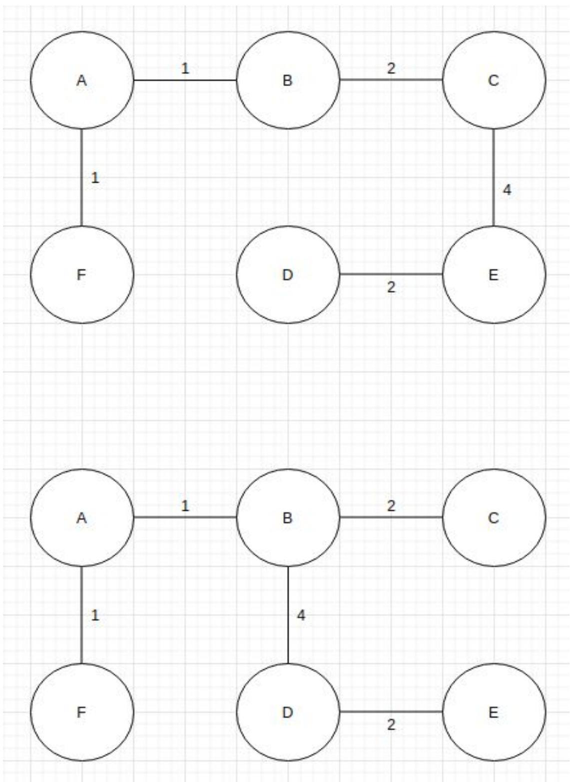
A ideia do algoritmo é a mesma da letra A, porém, não estamos realizando as mesmas chamadas repetidamente, armazenando-as na matriz dp.

Questão 2)

```
def maxTasks(tasks):
  if len(tasks) == 0: return 0, tasks
  tasks.sort(key = task_end) # ordenar por quem termina primeiro

  selected_tasks = tasks[0:1] # iniciar com o primeiro item da lista
  count = 1
  for task in tasks[1:]:
    if task[0] >= selected_tasks[-1][1]: # comparar com última tarefa selecionada
      selected_tasks.append(task)
      count += 1
  return count, selected_tasks

def task_end(task):
  return task[1]

T = [(2, 5), (11, 15), (4, 9), (7, 10)]
print(maxTasks(T))
```

Questão 3)

a) 2 árvores geradoras mínimas:



b) A aresta FD possui peso igual ao peso total das duas árvores geradoras mínimas possíveis.

c) O número cromático de G é 2. $\chi(G) = 2$

Questão 4)

a) Lista de Adjacências: A ideia é buscar na linked list (O(n)) a aresta em questão e arrumar os ponteiros, de forma a apontar o ponteiro do "pai", ou seja, vindo do adjacente anterior ao procurado, para apontar para o sucessor do procurado (sendo vazio ou não). Algoritmo O(n).

Dado: u.adjacentes e v.adjacentes = LinkedList
def **existe_deleta_aresta_estrutura**(u, v): # O(n)
  exists = False
  for adjacente in u.adjacentes:
    if adjacente == v:
      exists = True
      aponta_ponteiro_para_proximo_adjacente(adjacente) # apontar ponteiro apontado para si(v na lista encadeada) para o filho de v em questão
  if exists: # se forem adjacentes, apagar também na lista linkada do vertice v
    for adjacente v.adjacentes:
      if adjacente == u:
        aponta_ponteiro_para_proximo_adjacente(adjacente) # apontar ponteiro apontado para si(u na lista encadeada) para o filho de u em questão
    return 'Nós eram adjacentes. Aresta deletada.'
  else:
    return 'Nós não são adjacentes.'

Matriz de adjacências: Por ser uma matriz, a operação perde totalmente a complexidade temporal. Trata-se apenas de mudar os valores na matriz se for necessário. Algoritmo O(1)

def **existe_deleta_aresta_matriz**(u, v, matriz_adjacencias): # O(1)
  if matriz_adjacencias[u][v] >= 1:
    matriz_adjacencias[u][v] = 0
    matriz_adjacencias[v][u] = 0

b) Sendo um vetor, a parte de achar a aresta poderia ser feita via busca binária, já que estaria ordenado, sendo mais otimizado, já que é O(logn). Logo, a busca da adjacência seria mais eficiente. Já na destruição da aresta, fica pior. Porque deletando um, todos os itens do vetor terão que ser realocados na memória para manter a contiguidade.

A mudança do algoritmo seria algo assim:

def **existe_deleta_aresta_estrutura_b**(u, v): # O(logn)

search_response = binary_search(u.adjacentes, v) # Retorna None se nao encontrar. Se encontrar, retorna o indice no vetor do item procurado.

if search_response == None: return 'Nós não são adjacentes.' # guard clause para caso não sejam adjacentes

del u.adjacentes[search_response] # apagar item da memoria e o vetor se realoca na memoria pela contiguidade

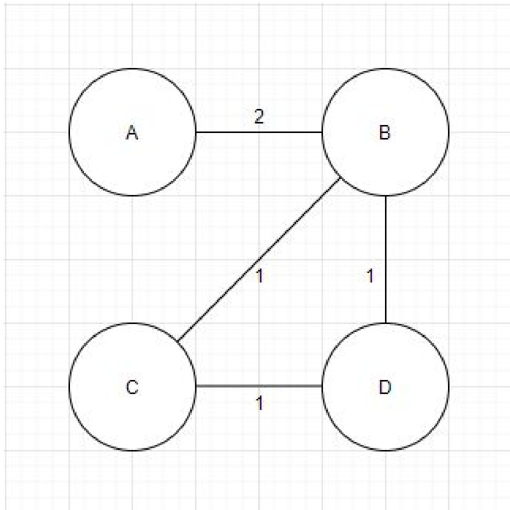
second_search = binary_search(v.adjacentes, u)
del v.adjacentes[second_search] # apagar item da memoria e o vetor se realoca na memoria pela contiguidade

return 'Nós eram adjacentes. Aresta deletada.'

Questão 5)

a)

Nesse grafo a direita, o algoritmo Mistério não retornaria uma árvore geradora mínima, pois incluiria o ciclo BCD.



b)

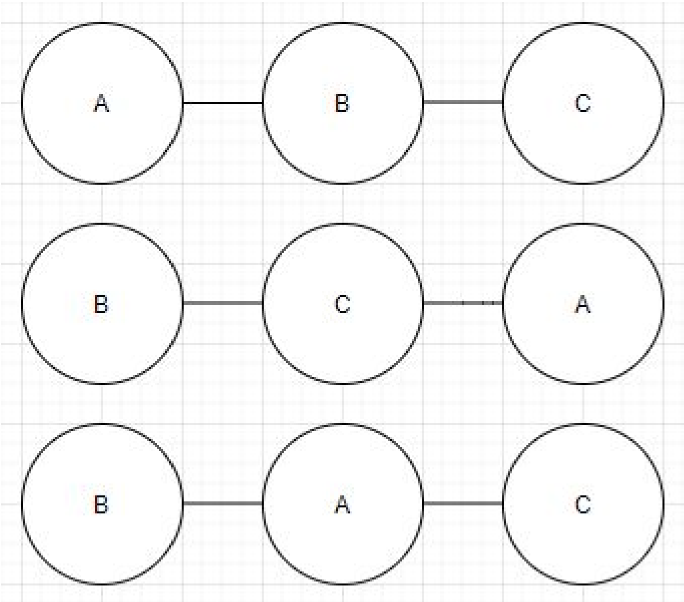
Algoritmo: Mistério (G)
Entrada: Um grafo G = (V, E)
M ← ∅
E ← sort(E)
enquanto M não for uma árvore geradora **faça**
    e ← min(E)
    se (M.acrescenta(e)) não formar ciclo **faça** # M.acrescenta é análogo a União
        M.acrescenta!(e) # alterar na memória valor de M diretamente
    Remova e de E
retorne M

c) Como toda árvore com n vértices têm n-1 arestas, M só seria árvore geradora de V quando |E(M)| = |V(G)| - 1. Por causa disso, desta equivalência, pode-se dizer que as duas formas são equivalentes e portanto o algoritmo continuaria correto com esta alteração.

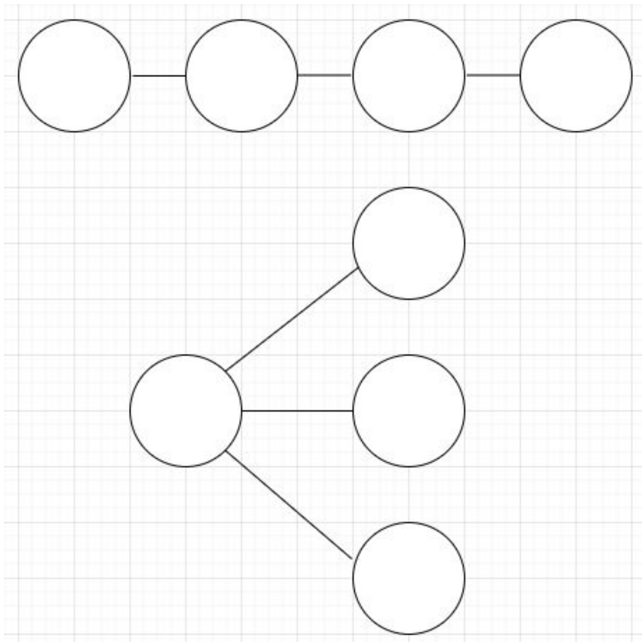
Questão 6)

a) T2 = K2, portanto, T2 = 1;

Com K3, temos 3 possibilidades:



Em K4, temos 2 tipos de estrutura:



Na primeira, as escolhas dos vértices podem ser permutadas, tendo então 4! combinações, porém, precisando excluir ordens idênticas, porém somente ao "contrário", portanto 4!/2, que é igual a 12 árvores geradoras mínimas possíveis.

Na segunda, a definição do nó "Pai" (nó mais a esquerda) , define o grafo, logo, possuímos 4 possíveis árvores geradoras dessa forma.

T2 = 1,
T3 = 3,
T4 = 12 + 4 = 16
Tn = n^(n-2)

Justificativa: como estamos montando árvores mínimas, faz sentido que a regra seja a mesma que a dada por Cayley. Podemos tomar essas árvores mínimas como "árvores com diferentes rótulos" e estruturas, dependendo do caso, gerando as suas respectivas sequências de Prufer.

b) n^(n-2), pois é uma sequência de n-2 números, sendo que cada "slot" tem n possibilidades, ficando n*n*n*n…*n (n-2 vezes)

c) **Algoritmo: PruferSeq(G)**

Entrada: Grafo G = (V,E)

T ← {} # conjunto vazio

folhas ← G.folhas

enquanto folhas não for vazio **faça**

    folha ← min(folhas)

    T.acrescenta!(folha.vizinho)

    Remove folha de G

se size(V) = 2 **então faça** # se tenho 2 vértices, retorne T

    retorne T

retorne T U PruferSeq(G) # recursividade, chamar a mesma função, para ver a próxima camada de folhas.

d) **Algoritmo: PruferTree(Seq)**

Entrada: Sequência de Prufer Seq = {}

N ← Length(Seq)

G ← Grafo com N+2 nós, sem arestas, rotulados de 1 a N+2

graus ← Array de Inteiros

para cada nó em G **faça**

    graus[nó] ← 1

para cada valor em Seq **faça**

    graus[valor] ← graus[valor] + 1

para cada valor em Seq **faça**

    para cada nó em G **faça**

        se graus[nó] = 1 **então faça**

            Crie Aresta(nó, valor) em G

            graus[nó] ← graus[nó] - 1

            graus[valor] ← graus[valor] - 1

            break

a ← b ← 0

para cada nó em G **faça**

    se graus[nó] = 1 **então faça**

        se a = 0 **então faça**

            a ← nó

        senão faça

            b ← nó

            break

graus[a] ← graus[a] - 1

graus[b] ← graus[b] - 1

retorne G