GRAFOS

Nombre: Enzo Palau

Ejercicio 1

```python
"""-----------Graphs--------------"""
#Part 1------->

"""Ejercicio 1"""

def createGraph(listV, listA):
    #operación crear grafo
    #ListV-listaNumeros, ListA-ListaTuplas
    graph=[]
    #creo la lista de adyacencia con los vertices
    for i in range(0,len(listV)):
        L=[]
        graph.append(L)
        graph[i].append(listV[i])
    #comparo y relleno
    for i in range(0,len(graph)):
        aux=graph[i][0]
        for j in range(0,len(listA)):
            #como no es dirijido inserto en ambos sentidos
            if listA[j][0]==aux:
                graph[i].append(listA[j][1])
            if listA[j][1]==aux:
                graph[i].append(listA[j][0])

    return graph
```

Ejercicio 2

```python
def existPath(Grafo, v1, v2):
    #un camino entre los vértices v1 y v2
    Lis_bfs=BFS(Grafo,v1)

    #como la funcion de bfs asume que el grafo es conexo
    # si no esta el v2 en la lista significa que no hay camino

    for current in Lis_bfs:
        if current is v2:
            return True
    return False
```

Ejercicio 3

```python
def isConnected(Grafo):
    #Implementa la operación es conexo
    Lis_bfs=BFS(Grafo,Grafo[0][0])
    verts=[]
    for current in Grafo:
        #los primeros elementos de la lista de ady son los vertices
        verts.append(current[0])

    #la funcion de bfs asume que el grafo es conexo asi que solo comparo
    for vert in verts:
        find=False
        for each in Lis_bfs:
            if each == vert:
                find=True
        if find is False:
            return False
    return True
```

```python
def BFS (graph,V):
    #return a list bfs Of Graph
    #coloco en la queue el vertice
    queue=[]
    queue.append(V)
    #creo las listas de recorridos y pendientes
    grayList=[]
    blackList=[]
    #marco el primer nodo como gris
    grayList.append(V)

    while len(queue)>0:
        #saco el elemento para luego ponerlo en la lista negra
        posc=SearchVert(queue[0],graph)
        aux=queue.pop(0)
        #me ubico en la lista de el elemto que estoy visitando
        L=graph[posc]
        #si no estan sus elementos en la lista gris los agrega
        for current in L:
            if search(grayList,current)==None:
                grayList.append(current)
                queue.append(current)
        blackList.append(aux)

    return blackList
```

Ejercicio 4

```python
def isTree(graph):
    #Implementa la operación es árbol
    #que sea conexo
    if  isConnected(graph) is False:
        return False
    #que no tenga ciclos
    if Aristas(graph)==len(graph)-1: return True
    else: return False

```

```python
def Aristas(Graph):
    #cuenta las aristas
    A_count=0
    for each in Graph:
        A_count+=len(each)-1
    return round(A_count/2)
```

Ejercicio 5

```python
"""Ejercicio 5"""

def isComplete(Grafo):
    #Implementa la operación es completo
    index=0
    L_verts=[]
    for each in Grafo:
        L_verts.append(Grafo[0][0])
    for i in range(0,len(Grafo)):
        #comparo las longuitudes de las listas de adyacencia
        if len(Grafo[i])!=len(L_verts):
            return False
        else:
            #chequeo que todos los elementos sean los de la lista de vertices
            cond=False
            for j in range(0,len(L_verts)):
                if Grafo[i][j]==L_verts[i]:
                    cond=True
            if cond is False:
                return True
    return True
```

Ejercicio 6

```python
"""Ejercicio 6"""
def convertTree(Grafo):
    #dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol
    Aux=[]#cuento las aristas
    for each in Grafo:
        for i in range(0,len(each)):
            if i != 0:
                if (each[0],each[i]) not in Aux and (each[i],each[0]) not in Aux:
                    Aux.append((each[0],each[i]))
    visted=[]
    elimA=[]
    cont=0 #si ambas estan entonces esa arista tiene que eliminarse
    while cont<len(Aux):
        flag=True
        if Aux[cont][0] not in visted or Aux[cont][1] not in visted:
            if Aux[cont][0] not in visted: visted.append(Aux[cont][0])
            if Aux[cont][1] not in visted: visted.append(Aux[cont][1])
            cont=cont+1
        else:
            elimA.append(Aux[cont])
            Aux.remove(Aux[cont])
    print (Aux)
    return elimA

```

Ejercicio 7

```python
#Part 2------->
"""Ejercicio 7"""

def countConnections(Grafo):
    #cantidad de componentes conexas
    arist=[] #lista de aristas
    for each in Grafo:
        for i in range(0,len(each)):
            if i != 0:
                if (each[0],each[i]) not in arist and (each[i],each[0]) not in arist:
                    arist.append((each[0],each[i]))
    L=make_set(Grafo)#coloco cada vertice en una lista propia
```

```python
def make_set(graph):
    L=[]
    for i in range(0,len(graph)):
        L.append([])
        L[i].append(graph[i][0])
    return L
```

```
    L=make_set(Grafo)#coloco cada vertice en una lista propia
    for i in range(0,len(arist)):#comparo y realizo la coleccion de conjuntos
        n1=arist[i][0]
        n2=arist[i][1]
        flag=False
        for x in range(0,len(L)):
            for j in range(0,len(L[x])):
                if n1==L[x][j]:
                    index1=x
                    flag=True
                if n2==L[x][j]:
                    index2=x
                    aux=L[index2]
                    flag=True
        if index1!=index2 and flag is True:
            L[index1].append(aux)
            if aux in L:
                L.remove(aux)
    print(L)
    return len(L)
```

Ejercicio 8

```
def convertToBFSTree(Grafo, v):
    #Convierte un grafo en un árbol BFS
    Aristas=[]
    if isConnected(Grafo) is True:
        #la op de bfs asume que es conexo
        #coloco en la queue el vertice
        queue=[]
        queue.append(v)
        #creo las listas de recorridos y pendientes
        grayList=[]
        blackList=[]
        #marco el primer nodo como gris
        grayList.append(v)

        while len(queue)>0:
            #saco el elemento para luego ponerlo en la lista negra
            posc=SearchVert(queue[0],Grafo)
            aux=queue.pop(0)
            #me ubico en la lista de el elemto que estoy visitando
            L=Grafo[posc]
            #si no estan sus elementos en la lista gris los agrega
            for current in L:
                if search(grayList,current)==None:
                    grayList.append(current)
                    queue.append(current)
                    Aristas.append((aux,current))
            blackList.append(aux)
    print(blackList)
    print(Aristas)
    return createGraph(blackList,Aristas)
```

Ejercicio 9

```
300 def convertToDFSTree(Grafo):
301     #convierte un grafo en un árbol DFS
302     grayList=[]
303     blackList=[]
304     whitelist=[]
305     Aristas=[]
306     #coloco todos en blanco
307     for each in Grafo:
308         whitelist.append(each[0])
309     for each in Grafo:
310         vertex=each[0]
311         #busco los nodos que son blancos
312         if search(whitelist,vertex)!=None:
313             DFS_Visit_TREE(Grafo,vertex,grayList,whitelist,blackList,Aristas)
314     blackList.reverse()
315     print(blackList)
316     print(Aristas)
317     return createGraph(blackList,Aristas)
318
```

```
376 def DFS_Visit_TREE(G,V,grayList,whitelist,blackList,Aristas):
377     #saco de los blancos y los pongo en gris
378     whitelist.remove(V)
379     grayList.append(V)
380     posc=SearchVert(V,G)
381     #busco en cada sublista blancos
382     for each in G[posc]:
383         if search(whitelist,each)!=None:
384             Aristas.append((each,V))
385             DFS_Visit_TREE(G,each,grayList,whitelist,blackList,Aristas)
386     blackList.append(V)
387
```

Ejercicio 10

```
321 def bestRoad(graph, start, goal):
322     explored = []
323
324     # Queue for traversing the
325     # graph in the BFS
326     queue = [[start]]
327
328     # If the desired node is
329     # reached
330     if start == goal:
331         print("Same Node")
332         return
333
334     # Loop to traverse the graph
335     # with the help of the queue
336     while queue:
337         path = queue.pop(0)
338         node = path[-1]
339
340         # Condition to check if the
341         # current node is not visited
342         if node not in explored:
343             neighbours = graph[node]
345             # Loop to iterate over the
346             # neighbours of the node
347             for neighbour in neighbours:
348                 new_path = list(path)
349                 new_path.append(neighbour)
350                 queue.append(new_path)
351
352                 # Condition to check if the
353                 # neighbour node is the goal
354                 if neighbour == goal:
355                     print("Shortest path = ", *new_path)
356                     return
357             explored.append(node)
358
359     # Condition when the nodes
360     # are not connected
361     print("So sorry, but a connecting"\
362                 "path doesn't exist :(")
363     return
364
```

Ejercicio 12

Por definición un grafo de N vértices es un árbol si es conexo y ademas la cantidad de aristas es igual a la cantidad de vértices menos 1, por lo que si partimos de un grafo el cual es conexo y es un árbol, al agregarle un vértice mas perdería la propiedad antes mencionada y se formarían ciclos por lo que dejaría de ser un árbol.

Ejercicio 13

Por hipótesis la arista no pertenece al árbol BFS ya que si perteneciese estaríamos en el caso de un árbol con aristas de retroceso por lo cual ya los niveles de u y v no difieren a lo sumo en 1.

Ejercicio 14:

```
444  def PRIM(G):
445      #algoritmo de prim para AACM
446      # Numero de vertices
447      N = len(G)
448      selected_node = [0]*N
449      U = 0
450      selected_node[0] = True
451
452      ListV=[]
453      ListA=[]
454
455      while (U < N - 1): #(O|V|)
456          vert=search_minimun_edge(N,selected_node,G,ListA)
457          selected_node[vert] = True
458          U += 1
459          ListV.append(vert)
460
461      return createGraph(ListV,ListA)
```

```
3  def search_minimun_edge(N,selected_node,G,ListA):
4      minimum = 999999
5      a = 0
6      b = 0
7      for i in range(N):
8              if selected_node[i] is True:
9                  for j in range(N):
0                      if selected_node[j] is not True and G[i][j]!=0:
1                          # not in selected and there is an edge
2                          if minimum > G[i][j]:
3                              minimum = G[i][j]
4                              a = i
5                              b = j
6      print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
7      aux=(a,b,G[a][b])
8      ListA.append(aux)
9      return b
```

Ejercicio 15

```
483 def KRUSKAL(Grafo):
484     #algoritmo de Kruskal para AACM
485     Arists=[]
486     verts=[]
487     Makeset=[]
488     #veo las aristas, los vertices y la implementacion de make set
489     for i in range(0,len(Grafo)):
490         if i!=0:
491             verts.append(Grafo[0][i])
492             Makeset.append([Grafo[0][i]])
493             for j in range(0,len(Grafo)):
494                 if j!=0:
495                     a=Grafo[0][i]
496                     b=Grafo[j][0]
497                     c=Grafo[a+1][b+1]
498                     if c!=0 and (((a,b,c) not in Arists) and ((b,a,c) not in Arists)) :
499                         Arists.append((a,b,c))
```

```
500         #ordeno los pesos
501         AS=sort_by_weight(Arists)
502         New_arist=[]
503         for each in AS:
504             a=find_set(each[0],Makeset)
505             b=find_set(each[1],Makeset)
506             if a is not b:
507                 New_arist.append(each)
508                 Union(a,b,Makeset)
509         print(New_arist)
510         return createGraph(verts,New_arist)
```

```
512 def sort_by_weight(L):
513     #ordena por el peso de la arista
514     weights=[]
515     ArSort=[]
516     for each in L:
517         weights.append(each[2])
518     weights.sort()
519     for i in range(0,len(weights)):
520         for j in range(0,len(L)):
521             if weights[i]==L[j][2]:
522                 if L[j] not in ArSort:
523                     ArSort.append(L[j])
524     return ArSort
```

```
526  def find_set(V,Makeset):
527      #busca el conjunto conexo de un vertice dado
528      for each in Makeset:
529          if V in each:
530              return each
531
532  def Union(l1,l2,L):
533      #une los conjuntos conexos
534      i=0
535      for each in L:
536          if l1==each:
537              aux=l1
538          elif l2==each:
539              index=i
540          i+=1
541      for each in aux:
542          L[index].append(each)
543      L.remove(l1)
```

Ejercicio 21:

```
class Vertex:
    distance=None
    parent=None
    key=None
```

```
def relax(u,v,G):
    #relaja el vertice actualizando su parent y su distancia
    if v.distance > (u.distance + calculeweight(u.key,v.key,G)):
        v.distance = u.distance + calculeweight(u.key,v.key,G)
        v.parent= u
```

```
577  def shortestPath(Grafo, s, v):
578      #determina el algoritmo de
579      verts=initRelax(Grafo,s)
580      visited=[]
581      queue=minqueue(verts)#ordeno por dist
582      while len(queue)>0:
583          u=queue.pop(0)
584          visited.append(u)
585          for each in adjunt(u.key,Grafo,verts):# lista de adjuntos
586              if each not in visited:
587                  relax(u,each,Grafo)
588          queue=minqueue(queue)
589      #bloque para ver S
590      a=None
591      b=None
592      for each in verts:
593          if each.key==s:
594              a=each
595          elif each.key==v:
596              b=each
597      S_Path=parent_path(b,a)
598      return  S_Path
```

```
def initRelax(G,s):
    #veo los vertices
    verts=[]
    Nodes=[]
    for i in range(0,len(G)):
        if i!=0:
            verts.append(G[0][i])
    #Relax inicial
    for ve in verts:
        if ve==s:
            newNode=Vertex()
            newNode.key=ve
            newNode.parent=None
            newNode.distance=0
            Nodes.append(newNode)
        else:
            newNode=Vertex()
            newNode.key=ve
            newNode.parent=None
            newNode.distance=9999#inf
            Nodes.append(newNode)
    return Nodes
```

```
626 def minqueue(V):
627     #devulve una queue con los nodos ordenados por distancia
628     q=[0]*len(V)
629     d=[]
630     for each in V:
631         d.append(each.distance)
632     d.sort()
633     for i in range(0,len(V)):
634         for each in V:
635             if d[i]==each.distance and each not in q:
636                 q[i]=each
637     return q
```

```
639 def adjunt(v,G,verts):
640     #da una lista de vertices adjuntos a el
641     adj=[]
642     aux=[]
643     for i in range(len(G)):
644         if i!=0:
645             if G[i][0]==v:
646                 for j in range(0,len(G[i])):
647                     if G[i][j]!=0 and j!=0:
648                         aux.append(G[0][j])
649     for each in verts:
650         if each.key in aux:
651             adj.append(each)
652     return adj
```

```
661 def calculeweight(u,v,G):
662     #calcula el peso de una arista
663     a=G[0].index(v)
664     a=a
665     for i in range(0,len(G)):
666         if i!=0 and G[i][0]==u:
667             return G[i][a]
668
669 def parent_path(v,s):
670     #calcula el camino mas corto mirando los parent
671     L=[]
672     L.insert(0,v.key)
673     aux=True
674     while aux!=False:
675         if v.parent is not None:
676             v=v.parent
677             if v.key==s.key:
678                 aux=False
679         else:
680             return None
681         L.append(v.key)
682         #L.reverse()
683     return L
684
```