

GRAFOS

Nombre: Enzo Palau

Ejercicio 1

```
1  """-----Graphs-----"""
2  #Part 1----->
3
4  """Ejercicio 1"""
5
6  def createGraph(listV, listA):
7      #operación crear grafo
8      #ListV-listaNumeros, ListA-ListaTuplas
9      graph=[]
10     #creo la lista de adyacencia con los vertices
11     for i in range(0,len(listV)):
12         L=[]
13         graph.append(L)
14         graph[i].append(listV[i])
15     #comparo y relleno
16     for i in range(0,len(graph)):
17         aux=graph[i][0]
18         for j in range(0,len(listA)):
19             #como no es dirigido inserto en ambos sentidos
20             if listA[j][0]==aux:
21                 graph[i].append(listA[j][1])
22             if listA[j][1]==aux:
23                 graph[i].append(listA[j][0])
24
25     return graph
```

Ejercicio 2

```
29 def existPath(Grafo, v1, v2):
30     #un camino entre los vértices v1 y v2
31     Lis_bfs=BFS(Grafo,v1)
32
33     #como la funcion de bfs asume que el grafo es conexo
34     # si no esta el v2 en la lista significa que no hay camino
35
36     for current in Lis_bfs:
37         if current is v2:
38             return True
39     return False
40
```

Ejercicio 3

```
43 def isConnected(Grafo):
44     #Implementa la operación es conexo
45     Lis_bfs=BFS(Grafo,Grafo[0][0])
46     verts=[]
47     for current in Grafo:
48         #los primeros elementos de la lista de ady son los vertices
49         verts.append(current[0])
50
51     #la funcion de bfs asume que el grafo es conexo asi que solo comparo
52     for vert in verts:
53         find=False
54         for each in Lis_bfs:
55             if each == vert:
56                 find=True
57         if find is False:
58             return False
59     return True
```

```
141 def BFS (graph,V):
142     #return a list bfs Of Graph
143     #coloco en la queue el vertice
144     queue=[]
145     queue.append(V)
146     #creo las listas de recorridos y pendientes
147     grayList=[]
148     blackList=[]
149     #marco el primer nodo como gris
150     grayList.append(V)
151
152     while len(queue)>0:
153         #saco el elemento para luego ponerlo en la lista negra
154         posc=SearchVert(queue[0],graph)
155         aux=queue.pop(0)
156         #me ubico en la lista de el elemto que estoy visitando
157         L=graph[posc]
158         #si no estan sus elementos en la lista gris los agrega
159         for current in L:
160             if search(grayList,current)==None:
161                 grayList.append(current)
162                 queue.append(current)
163         blackList.append(aux)
164
165     return blackList
```

Ejercicio 4

```

63 def isTree(graph):
64     #Implementa la operación es árbol
65     #que sea conexo
66     if isConnected(graph) is False:
67         return False
68     #que no tenga ciclos
69     if Aristas(graph)==len(graph)-1: return True
70     else: return False
71

```

```

194 def Aristas(Graph):
195     #cuenta las aristas
196     A_count=0
197     for each in Graph:
198         A_count+=len(each)-1
199     return round(A_count/2)
200

```

Ejercicio 5

```

72 """Ejercicio 5"""
73
74 def isComplete(Grafo):
75     #Implementa la operación es completo
76     index=0
77     L_verts=[]
78     for each in Grafo:
79         L_verts.append(Grafo[0][0])
80     for i in range(0,len(Grafo)):
81         #comparo las longitudes de las listas de adyacencia
82         if len(Grafo[i])!=len(L_verts):
83             return False
84         else:
85             #chequeo que todos los elementos sean los de la lista de vertices
86             cond=False
87             for j in range(0,len(L_verts)):
88                 if Grafo[i][j]==L_verts[i]:
89                     cond=True
90             if cond is False:
91                 return True
92     return True

```

Ejercicio 6

```

4  """Ejercicio 6"""
5  def convertTree(Grafo):
6      #dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol
7      Aux=[]#cuento las aristas
8      for each in Grafo:
9          for i in range(0,len(each)):
10             if i != 0:
11                 if (each[0],each[i]) not in Aux and (each[i],each[0]) not in Aux:
12                     Aux.append((each[0],each[i]))
13
14      visted=[]
15      elimA=[]
16      cont=0 #si ambas estan entonces esa arista tiene que eliminarse
17      while cont<len(Aux):
18          flag=True
19          if Aux[cont][0] not in visted or Aux[cont][1] not in visted:
20              if Aux[cont][0] not in visted: visted.append(Aux[cont][0])
21              if Aux[cont][1] not in visted: visted.append(Aux[cont][1])
22              cont=cont+1
23          else:
24              elimA.append(Aux[cont])
25              Aux.remove(Aux[cont])
26      print (Aux)
27      return elimA

```

Ejercicio 7

```

233  #Part 2----->
234  """Ejercicio 7"""
235
236  def countConnections(Grafo):
237      #cantidad de componentes conexas
238      arist=[] #lista de aristas
239      for each in Grafo:
240          for i in range(0,len(each)):
241              if i != 0:
242                  if (each[0],each[i]) not in arist and (each[i],each[0]) not in arist:
243                      arist.append((each[0],each[i]))
244      L=make_set(Grafo)#coloco cada vertice en una lista propia

```

```

334  def make_set(graph):
335      L=[]
336      for i in range(0,len(graph)):
337          L.append([])
338          L[i].append(graph[i][0])
339      return L

```



```

244     L=make_set(Grafo)#coloco cada vertice en una lista propia
245     for i in range(0,len(arist)):#comparo y realizo la coleccion de conjuntos
246         n1=arist[i][0]
247         n2=arist[i][1]
248         flag=False
249         for x in range(0,len(L)):
250             for j in range(0,len(L[x])):
251                 if n1==L[x][j]:
252                     index1=x
253                     flag=True
254                 if n2==L[x][j]:
255                     index2=x
256                     aux=L[index2]
257                     flag=True
258             if index1!=index2 and flag is True:
259                 L[index1].append(aux)
260                 if aux in L:
261                     L.remove(aux)
262     print(L)
263     return len(L)

```

Ejercicio 8

```

267 def convertToBFSTree(Grafo, v):
268     #Convierte un grafo en un árbol BFS
269     Aristas=[]
270     if isConnected(Grafo) is True:
271         #la op de bfs asume que es conexo
272         #coloco en la queue el vertice
273         queue=[]
274         queue.append(v)
275         #creo las listas de recorridos y pendientes
276         grayList=[]
277         blackList=[]
278         #marco el primer nodo como gris
279         grayList.append(v)
280
281         while len(queue)>0:
282             #saco el elemento para luego ponerlo en la lista negra
283             posc=SearchVert(queue[0],Grafo)
284             aux=queue.pop(0)
285             #me ubico en la lista de el elemto que estoy visitando
286             L=Grafo[posc]
287             #si no estan sus elementos en la lista gris los agrega
288             for current in L:
289                 if search(grayList,current)==None:
290                     grayList.append(current)
291                     queue.append(current)
292                     Aristas.append((aux,current))
293             blackList.append(aux)
294     print(blackList)
295     print(Aristas)
296     return createGraph(blackList,Aristas)
297

```

Ejercicio 9

```
299
300 def convertToDFS_Tree(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]
344
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 además 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 más 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.

