UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Algoritmos y estructuras de datos II

Grafos

Ejercicio 1

Implementar la función crear grafo que dada una lista de vértices y una lista de aristas cree un grafo con la representación por Lista de Adyacencia.

def createGraph(List, List)
Descripción: Implementa la operación crear grafo
Entrada: **LinkedList** con la lista de vértices y **LinkedList** con la lista de aristas donde por cada par de elementos representa una conexión entre dos vértices.
Salida: retorna el nuevo grafo

```
4   class Vertex:
5       def __init__(self,key):
6           self.key = key
7       color = None
8       parent = None
9       distance = None
10      f = None
11
12  class Graph:
13      # vertices_list = [v1,v2,v3,...,vn]
14      # edges_list = [(v1,v2),(v2,v3),...,(vi,vj)]
15      def __init__(self,vertices_list,edges_list):
16          self.vertices_list = vertices_list
17          self.edges_list = edges_list
18
19          self.adj_list = [[] for _ in range(len(self.vertices_list))]
20
21          for i in range(len(edges_list)):
22              self.adj_list[edges_list[i][0].key - 1].append(edges_list[i][1])
23              self.adj_list[edges_list[i][1].key - 1].append(edges_list[i][0])
```

Ejercicio 2

Implementar la función que responde a la siguiente especificación.

def existPath(Grafo, v1, v2):
Descripción: Implementa la operación existe camino que busca si existe un camino entre los vértices v1 y v2
Entrada: **Grafo** con la representación de Lista de Adyacencia, **v1** y **v2** vértices en el grafo.
Salida: retorna **True** si existe camino entre v1 y v2, **False** en caso contrario.

```
def exist_path(self,v1,v2):
    if v1 not in self.vertices_list or v2 not in self.vertices_list:
        return 'Error: v1 or v2 not belong to graph'
    if v1 in self.adj_list[v2-1] and v2 in self.adj_list[v1-1]:
        return True
    return False
```

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Ejercicio 3

Implementar la función que responde a la siguiente especificación.

def isConnected(Grafo):
Descripción: Implementa la operación es conexo
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si existe camino entre todo par de vértices, **False** en caso contrario.

```
34      def is_connected(self):
35          newG = copy.deepcopy(self)
36          BFS(newG,v1)
37          for vertex in newG.vertices_list:
38              if vertex.color == 'white':
39                  return False
40          return True
```

```
135 def BFS(G,s):
136     for vertex in G.vertices_list:
137         if vertex.key != s.key:
138             vertex.color = 'white'
139             vertex.distance = math.inf
140     s.color = 'gray'
141     s.distance = 0
142     Q = []
143     Q.insert(0,s)
144     while Q != []:
145         u = Q.pop()
146         for vertex in G.adj_list[u.key-1]:
147             if vertex.color == 'white':
148                 vertex.color = 'gray'
149                 vertex.distance = u.distance + 1
150                 vertex.parent = u
151                 Q.insert(0,vertex)
152         u.color = 'black'
```

Ejercicio 4

Implementar la función que responde a la siguiente especificación.

def isTree(Grafo):
Descripción: Implementa la operación es árbol
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si el grafo es un árbol.

```
42      def is_tree(self):
43          # Un arbol es un grafo de n vertices, aciclico con n-1 aristas
44          # Verifico cantidad de aristas
45          if len(self.vertices_list) - 1 != len(self.edges_list):
46              return False
```

```
47
48          # Detecto si existen ciclos en el grafo
49          visited = set()
50          start_vertex = self.vertices_list[0]
51          stack = [(start_vertex,None)]
```

```
52          while stack != []:
53              current_vertex, parent = stack.p   (variable) current_vertex: Any
54              visited.add(current_vertex)
55              for adj_vertex in self.adj_list[current_vertex.key-1]:
56                  if adj_vertex != parent:
57                      if adj_vertex not in visited:
58                          stack.append((adj_vertex,current_vertex)) # Ciclo detectado, el grafo no es un arbol
59                      else:
60                          return False
61
62          # Reviso si hay vertices no visitados (componentes desconectadas)
63          if len(visited) != len(self.vertices_list):
64              return False
65
66          # Si se cumplen con todas las condiciones, el grafo es un arbol
67          return True
```

Ejercicio 5

Implementar la función que responde a la siguiente especificación.

```
def isComplete(Grafo):
      Descripción: Implementa la operación es completo
      Entrada: Grafo con la representación de Lista de Adyacencia.
      Salida: retorna True si el grafo es completo.
```

Nota: Tener en cuenta que un grafo es completo cuando existe una arista entre todo par de vértices.

```
69      def is_complete(self):
70          vertices_count = len(self.vertices_list)
71          if len(self.edges_list) != vertices_count * ((vertices_count - 1) / 2):
72              return False
73          return True
```

Ejercicio 6

Implementar una función que dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol. Respetar la siguiente especificación.

```
def convertTree(Grafo)
      Descripción: Implementa la operación es convertir a árbol
      Entrada: Grafo con la representación de Lista de Adyacencia.
      Salida: LinkedList de las aristas que se pueden eliminar y el grafo
      resultante se convierte en un árbol.
```

```
75      def convert_tree(self):
76          removal_list = []
77          visited = set()
78          start_vertex = self.vertices_list[0]
79
80          self.dfs_convert_tree(start_vertex,visited,None,removal_list)
81
82          return removal_list
```

```
84      def dfs_convert_tree(self, vertex, visited, parent, removal_list):
85              visited.add(vertex)
86              for neighbor in self.adj_list[vertex.key - 1]:
87                  print('vertex:',vertex.key,'neighbor',neighbor.key)
88                  if neighbor not in visited:
89                      self.dfs_convert_tree(neighbor, visited, vertex, removal_list)
90                  elif neighbor != parent:
91                      removal_list.append([vertex.key, neighbor.key])
```