NOMBRE: ALVARO BLANCO
LEGAJO: 10622
TRABAJO PRACTICO Grafos parte I y II

Ejercicio 1

```
def createGraph(vertices,edges):
  Graph = {}
  for vertex in vertices:
    Graph[vertex] = []
  for vertex1, vertex2 in edges:
    Graph[vertex1].append(vertex2)
    Graph[vertex2].append(vertex1)
  return Graph
```

Ejercicio 2

```
def existPath(Grafo, v1, v2):
  visited = set()
  stack = [v1]
  while stack:
    vertex = stack.pop()
    if vertex == v2:
      return True
    if vertex not in visited:
      visited.add(vertex)
      stack.extend(Grafo[vertex])
  return False
```

Ejercicio 3

```
def isConnected(Grafo):
  vertices = list(Grafo.keys())
  for i in range(len(vertices)):
    v1 = vertices[i]
    for j in range(i + 1, len(vertices)):
      v2 = vertices[j]
      flag = existPath(Grafo, v1, v2)
      if flag == False:
        return False

  return True
```

Ejercicio 4

```
def isTree(Grafo):
  n = len(Grafo)
  edges = countEdges(Grafo)
  if isConnected(Grafo) and hasCycleDFS(Grafo) == False and edges == n-1:
    return True
  else:
    return False
```

```python
def countEdges(Grafo):
  contEdges = 0
  for vertex in Grafo:
    contEdges += len(Grafo[vertex])
  return contEdges // 2
```

```python
def hasCycleDFS(Grafo):
  visited = set()
  stack = []
  for vertex in Grafo:
    if vertex not in visited:
      stack.append((vertex, None))
      while stack:
        curr, prev = stack.pop()
        visited.add(curr)
        for neighbor in Grafo[curr]:
          if neighbor not in visited:
            stack.append((neighbor, curr))
          elif neighbor != prev:
            return True
  return False
```

Ejercicio 5

```python
def isComplete(Grafo):
  n = len(Grafo)
  for vertex in Grafo:
    if len(Grafo[vertex]) != n - 1:
      return False
  return True
```

Ejercicio 7

```python
def countConnections(Grafo):
  visited = set()
  count = 0
  for vertex in Grafo:
    if vertex not in visited:
      DFS_2(Grafo, vertex, visited)
      count += 1
  return count
```

Ejercicio 8

```python
def convertToBFSTree(Grafo, v):
  visited = set()
  q = Queue()
  visited.add(v)
  q.put(v)
  bfs = {v: []}
  while not q.empty():
    vertex = q.get()
    for neighbor in Grafo[vertex]:
      if neighbor not in visited:
        visited.add(neighbor)
        q.put(neighbor)
        bfs[vertex].append(neighbor)
        bfs[neighbor] = []
  return bfs
```

Ejercicio 9

```
def convertToDFSTree(Grafo, v):
  DFS_Tree = {v: []}
  visited = set()
  DFS(Grafo, v, v, DFS_Tree, visited)
  for vertex in Grafo:
    if vertex not in visited:
      DFS_Tree[vertex] = []
      DFS(Grafo, vertex, vertex, DFS_Tree, visited)
  return DFS_Tree

def DFS(Grafo, v, root, DFS_Tree, visited):
  visited.add(v)
  for adj_vertex in Grafo[v]:
    if adj_vertex not in visited:
      DFS_Tree[root].append(adj_vertex)
      DFS_Tree[adj_vertex] = []
      DFS(Grafo, adj_vertex, root, DFS_Tree, visited)
```

Ejercicio 10

```
def bestRoad(Grafo, v1, v2):
  visited = set()
  q = Queue()
  visited.add(v1)
  q.put((v1, []))

  while not q.empty():
    vertex, ruta = q.get()
    if vertex == v2:
      return ruta + [vertex]
    for neighbor in Grafo[vertex]:
      if neighbor not in visited:
        visited.add(neighbor)
        q.put((neighbor, ruta + [vertex]))
  return []
```

Ejercicio 12

Si estamos en presencia de un arbol eso significa que hay solo un camino de la raiz a cada nodo del arbol, si yo agrego una arista entre cualquier par de nodos del arbol esto nos lleva a tener 2 caminos distintos para llegar al mismo nodo, lo que es en efecto un ciclo.

Ejercicio 13

Si la arista (u,v) no pertenece al arbol BFS. Esto significa que la unica forma en que se puede llegar a v desde la raiz es a traves de otro vertice, digamos w, que ya ha sido descubierto y explorando la arista (w,v). Por lo tanto, v no puede estar en un nivel superior al de w en el arbol BFS, ya que de lo contrario se habria descubierto antes que w y no se habria necesitado una arista adicional para alcanzarlo.

PARTE III

Ejercicio 14

```
def PRIM(Grafo):
  n = len(Grafo)
  visitados = [False] * n
  padre = [None] * n
  costo = [float('inf')] * n
  costo[0] = 0

  for _ in range(n):
    u = None
    for i in range(n):
      if not visitados[i] and (u is None or costo[i] < costo[u]):
        u = i

    visitados[u] = True
    for v in range(n):
      if Grafo[u][v] != 0 and not visitados[v] and Grafo[u][v] < costo[v]:
        costo[v] = Grafo[u][v]
        padre[v] = u

  arbol = [[] for _ in range(n)]
  for v in range(1, n):
    arbol[padre[v]].append(v)
    arbol[v].append(padre[v])

  return arbol
```

Ejercicio 15

```
def get_peso(arista):
  return arista[2]

def KRUSKAL(Grafo):
  n = len(Grafo)
  aristas = []
  for i in range(n):
    for j in range(i + 1, n):
      if Grafo[i][j] != 0:
        aristas.append((i, j, Grafo[i][j]))
  aristas = sorted(aristas, key=get_peso)
  componentes_conexas = [[i] for i in range(n)]

  arbol = []
  for arista in aristas:
    u, v, peso = arista
    componente_u = None
    componente_v = None
    for componente in componentes_conexas:
      if u in componente:
        componente_u = componente
      if v in componente:
        componente_v = componente
    if componente_u != componente_v:
      arbol.append((u, v))
      componente_u.extend(componente_v)
```

```
    componentes_conexas.remove(componente_v)
return arbol
```

Ejercicio 16

Supongamos que el grafo G es un grafo no dirigido y conexo, y que se divide en dos conjuntos disjuntos U y V - U. Sea (u, v) la arista de menor costo que conecta un nodo en U con uno en V - U.

Ahora, consideremos un árbol abarcador de costo mínimo T de G que no contiene la arista (u, v). Ya que T es un árbol abarcador de costo mínimo, su costo debe ser menor o igual al costo de cualquier otro árbol abarcador de G.

Podemos considerar dos casos:

1. Si T no contiene ningún nodo de V - U, entonces (u, v) es la única arista que conecta U y V - U. Si eliminamos la arista (u, v) del grafo G, entonces este se divide en dos componentes: uno que contiene el nodo u y otro que contiene el nodo v. Como T no contiene ningún nodo en V - U, debe estar contenido en el componente que contiene el nodo u. Pero entonces, no es posible que T sea un árbol abarcador de costo mínimo, ya que se requiere la arista (u, v) para conectar los dos componentes y formar un árbol abarcador.
2. Si T contiene al menos un nodo en V - U, entonces podemos encontrar un camino en T que conecte un nodo en U con un nodo en V - U. Si eliminamos la arista (u, v) del grafo G, el camino en T todavía conectará los mismos dos nodos, pero a través de una ruta más larga. Esto significa que hay al menos una arista en el camino que conecta un nodo en U con un nodo en V - U. La arista de menor costo en este camino es la arista (u, v), por lo que si eliminamos cualquier otra arista en el camino y agregamos la arista (u, v), obtendremos un nuevo árbol abarcador de costo menor que T, lo que contradice el supuesto de que T es un árbol abarcador de costo mínimo.

Por lo tanto, podemos concluir que la arista (u, v) pertenece a cualquier árbol abarcador de costo mínimo del grafo G.