UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

FACULTAD
DE INGENIERÍA

UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

ALGORITMOS Y ESTRUCTURAS DE DATOS II

Grafos

Yacante Daniel

Leg: 10341

Estructura principal de la clase grafo() y sus métodos.

```python
"""
La estructura grafo() consta de una hashTable que va a contener a los vertices, la estructura contenida es un grafNodo()
el cual contiene al valor del vértice y la lista de adyacencia con los nodos vecinos.
Cada vez que se instancia la estructura grafo() se deberá pasar como parámetro a La lista de vertices, con esto se creara la hashTable
con los vertices y se inicializaran algunos parámetros de la estructura que tendrán uso para varias de las funciones.
"""
class grafo():
    def fhash(self,key,i):
        m=self.m
        h1=lambda key: key
        h2=lambda key: 1+(key%(m-1))
        return (h1(key)+i*h2(key))%m

    def __init__(self,nVertices:list) -> None:
        self.nodos=[None for n in nVertices]
        self.m=len(nVertices)
        self.vistos=[]
        self.noVistos=[]
        self.aristas=0
        self.componentes=0
        for v in nVertices:                         #En bucle se van insertando los valores de los vertices, utilizando una
            IN=False                                #funcion de doble hash
            for i in range(self.m):
                pos=self.fhash(v,i)
                if self.nodos[pos]==None:
                    self.nodos[pos]=grafNodo(v)
                    self.noVistos.append(v)
                    IN=True
                    break
            if not IN:
                print(f"Vértice {v} sin insertar")
    #metodo para imprimir el grafo en su forma de lista de adyacencia
    def __str__(self) -> str:
        cad=""
        for n in self.nodos:
            cad+=str(n.value)+"->"
            if n.vecinos!=None:
                cad+=", ".join(str(vec) for vec in n.vecinos)
            cad+="\n"
        return cad
    #funcion que devuelve la posicion de un elemento en la hash, si no existe regresa none
    def _getPos(self,key):
        enc=False
        for i in range(self.m):
            pos=self.fhash(key,i)
            if self.nodos[pos].value==key:
                enc=True
                break
        if not enc:
            # print(f"Vértice {x} no encontrado")
            return None
        else:
            return pos
    #getters del numero de aristas y de numero de nodos/vertices
    def getNumAristas(self):
        return self.aristas
    def getNumNodos(self):
        return self.m
    #función que añade una arista, al ser un grafo no direccional en la lista de adyacencia aparecerá tanto u->v como v->u
    def addArista(self,arista:tuple):
        x,y=arista
        posx=self._getPos(x)
        posy=self._getPos(y)
        if posx!=None and posy!=None:
            self.aristas+=1
            if self.nodos[posx].vecinos != None:
                self.nodos[posx].vecinos.append(y)
            else:
                self.nodos[posx].vecinos=[y]
            if self.nodos[posy].vecinos != None:
                self.nodos[posy].vecinos.append(x)
            else:
                self.nodos[posy].vecinos=[x]
            return True
        else:
            return False
"""
Estructura del vértice para la hashTable
"""
class grafNodo():
    def __init__(self,value) -> None:
        self.value=value
        self.vecinos=None
```

Ejercicio 1

```python
"""
La función createGraph() instancia una estructura grafo() con la lista de vertices proporcionada y luego en un bucle va añadiendo la
lista de aristas pasada
"""
def createGraph(vertices:list, aristas:list):
    graph=grafo(vertices)
    for ar in aristas:
        if not graph.addArista(ar):
            print(f"Arista {ar} no valida")
    return graph
```

Al instanciar la clase grafo(vértices) se usa una hashtable para insertar cada uno de los vértices como estructura de grafNodo().

El método addArista((u,v)) lo que hace es obtener las posiciones tanto de u como v en la hashtable interna, y para cada vertice añade a la lista de vecinos al otro vertice. Si el grafo fuera direccional solamente haría falta insertar el vertice en un único sentido y no en ambos como se hace aquí.

Ejercicio 2

```python
"""
Para ver si existe un camino entre v1 y v2 utilizo un algoritmo que primero se fija si no lo tiene de vecino al vértice buscado
Y si no lo tiene hace una búsqueda en profundidad con los nodos vecinos, es decir que principalmente realiza una búsqueda en profundidad
pero a su vez por cada nivel se fija si el nodo v2 no es vecino del vértice en el que esta parado actualmente.
"""
def existPath(graf:grafo,v1, v2):
    posV1=graf._getPos(v1)
    if posV1!=None and not (graf.nodos[posV1].value in graf.vistos):
        vec=graf.nodos[posV1].vecinos
        graf.vistos.append(graf.nodos[posV1].value)
        if vec!=None:
            for v in vec:
                if v==v2:
                    graf.vistos.clear()
                    return True
            for v in vec:
                resp=existPath(graf,v,v2)
                if resp:
                    return resp
        return False
```

Ejercicio 3

```python
"""
Para saber si un grafo es conexo para cada par de vertices tiene que existir un camino, por lo que en 2 loops anidados se pasa la función
existPath para ver si hay un camino entre cada par de vertices, y si en alguno no hay camino, regresa False, sino si se prueba cada par de vertices
y no se encontró que falte algún camino regresa True
"""
def isConnected(graf:grafo):
    nodos=graf.nodos
    for i in range(len(nodos)):
        for j in range(len(nodos)):
            if i!=j:
                if not existPath(graf,graf.nodos[i].value,graf.nodos[j].value):
                    graf.vistos.clear()
                    return False
    return True
```

Ejercicio 4

```python
"""
Para que un grafo sea un árbol deben ocurrir 3 cosas, que sea conexo, que el numero de aristas sea el numero de vertices menos 1
y que no hayan ciclos.
"""
def isTree(graf:grafo):
    return len(detectCycles(graf))>0 and isConnected(graf) and (graf.getNumAristas()==graf.getNumNodos()-1)
```

La funcion isConnected ya esta mostrada, luego ver si nA=(nV-1) es una operación matemática, ya que estos valores se calculan al momento de generar el grafo. Por lo tanto solamente queda como encontrar los ciclos dentro del grafo.

Implementación de detectCycles()

```python
"""
Para encontrar ciclos en un grafo realizo una búsqueda en profundidad viendo si el vértice vecino que trato de acceder no esta ya
en la lista de vertices que se han recorrido, a su vez aprovecho para guardar entre que vertices seria que se cierra el ciclo
ya que sera de utilidad para algunas de las funciones pedidas. Mientras recorro vertices, voy sacándolos de la lista de los vertices
no vistos, ya que puede suceder que el grafo no sea conexo y de esta forma siempre tengo registro de por cuales vertices me queda
pasar y de esta forma puedo contar las partes conexas del grafo.
"""
def _cycles(graf:grafo,v,va,ciclos:set):
    pos=graf._getPos(v)
    if pos!=None:
        if not (graf.nodos[pos].value in graf.vistos):
            vec=graf.nodos[pos].vecinos
            graf.vistos.append(graf.nodos[pos].value)
            graf.noVistos.remove(graf.nodos[pos].value)
            if vec!=None:
                for vn in vec:
                    if vn!=va:
                        resp=_cycles(graf,vn,v,ciclos)
                        if resp:
                            # ciclos.append((v,vn))
                            ciclos.add(tuple(sorted((vn,v))))
                return False
        else:
            return True
"""
La función detectCycles hace uso de la parte recursiva _cycles() para ver si hay ciclos, pero a su vez cuenta las partes conexas del
grafo, ya que mientras haya elementos sin ver, se eligir como punto de partida de la búsqueda en profundidad a uno de los vertices
sin ver.
"""
def detectCycles(graf:grafo):
    nodes=graf.noVistos.copy()
    loop=set()
    cont=0
    while graf.getNumNodos()>len(graf.vistos):
        posi=graf._getPos(graf.noVistos[randint(0,len(graf.noVistos)-1)])
        _cycles(graf,graf.nodos[posi].value,None,loop)
        cont+=1
    graf.componentes=cont
    graf.vistos.clear()
    graf.noVistos=nodes.copy()
    return loop
```

Ejercicio 5

```python
"""
Para que un grafo sea completo cada vértice tiene que tener de vecinos al resto de vertices
"""
def isComplete(graf:grafo):
    nNodes=len(graf.nodos)
    complete=False
    for nodo in graf.nodos:
        vecinos=nodo.vecinos
        if len(vecinos)==(nNodes-1) and not (nodo.value in vecinos):
            complete= True
        else:
            complete=False
            break
    return complete
```

Ejercicio 6

```python
"""
Ya que la función convertTree() debe retornar una lista de aristas a eliminar para que el grafo sea un árbol, simplemente es la ejecución
de la función detectCycles que ya calcula las aristas que de añadirse forman ciclos al árbol
"""
def convertTree(graf:grafo):
    return detectCycles(graf)
```

Ejercicio 7

```python
"""
Para encontrar las partes conexas primero me fijo si no se ha ejecutado nunca la función detectCycles() ya que en ella se calculan las
partes conexas como medio de búsqueda de ciclos. Sino simplemente regreso el valor ya previamente calculado.
"""
def countConnections(graf:grafo):
    if graf.componentes==0:
        detectCycles(graf)
    return graf.componentes
```

Ejercicio 8

```
"""
Parte recursiva de búsqueda en anchura que va viendo cuales de los vecinos del nodo obtenido de la queue (graf.vistos es una lista
pero se puede usar como queue) puede pasar como arista al nuevo grafo que solo tendrá las aristas que no  cierren ciclos.
"""
def _bfsTree(graf:grafo,tree:grafo):
    v=graf.vistos.pop(0)
    graf.noVistos.remove(v)
    posG=graf._getPos(v)
    if posG!=None:
        vec=graf.nodos[posG].vecinos
        if vec!=None:
            for vn in vec:
                if not (vn in graf.vistos) and (vn in graf.noVistos):
                    tree.addArista((vn,v))
                    graf.vistos.append(vn)
            if len(graf.vistos)!=0:
                _bfsTree(graf,tree)
            else:
                return
"""
Lo primero que hago es hacer una copia de los elementos no vistos del grafo y con ellos generar un nuevo grafo que sera el equivalente
al grafo principal pero solo con las aristas que no generen ciclos. Ambos grafos se pasan a la función recursiva _bfsTree()
"""
def convertToBFSTree(graf:grafo, v):
    if isConnected(graf):
        nodes=graf.noVistos.copy()
        grafTree=grafo(graf.noVistos)
        graf.vistos.append(v)
        _bfsTree(graf,grafTree)
        graf.vistos.clear()
        graf.noVistos=nodes.copy()
        return grafTree
    else:
        return None
```

Ejercicio 9

```
"""
Parte recursiva de la búsqueda en profundidad, nada mas que no se utiliza la queue como medio de ver que vértice tomar para la proxima
iteración, sino que como medio de chequeo que no se tome un valor ya visto anteriormente
"""
def _dfsTree(graf:grafo,tree:grafo,v):
    pos=graf._getPos(v)
    if pos!=None:
        if not (graf.nodos[pos].value in graf.vistos):
            vec=graf.nodos[pos].vecinos
            graf.vistos.append(graf.nodos[pos].value)
            graf.noVistos.remove(graf.nodos[pos].value)
            if vec!=None:
                for vn in vec:
                    if not vn in graf.vistos:
                        resp=_dfsTree(graf,tree,vn)
                        tree.addArista((vn,v))
                return False
        else:
            return True
    return
"""
Al igual que en convertToBFSTree aquí se hace un nuevo grafo para ir añadiendo las aristas que no generen ciclos.
"""
def convertToDFSTree(graf:grafo, v):
    if isConnected(graf):
        nodes=graf.noVistos.copy()
        grafTree=grafo(graf.noVistos)
        _dfsTree(graf,grafTree,v)
        graf.vistos.clear()
        graf.noVistos=nodes.copy()
        return grafTree
    else:
        return None
```

Ejercicio 10

```
"""
Para encontrar la mejor ruta o la ruta mas corta entre 2 vertices se debe realizar una búsqueda en anchura, por lo que el algoritmo
no es muy diferente a los anteriores, aquí no añado aristas sino que voy viendo si el nodo que se busca es vecino de en el que estoy
parado, y si lo es regreso una lista con el vértice actual y el que se buscaba. Dicha lista se ira agrandando cada vez que suba un nivel
"""
def _bestRoute(graf:grafo,v2):
    v=graf.vistos.pop(0)
    graf.noVistos.remove(v)
    posG=graf._getPos(v)
    if posG!=None:
        vec=graf.nodos[posG].vecinos
        if vec!=None:
            for vn in vec:
                if not (vn in graf.vistos) and (vn in graf.noVistos):
                    if vn==v2:
                        return [vn,v]
                    else:
                        graf.vistos.append(vn)
            if len(graf.vistos)!=0:
                resp=_bestRoute(graf,v2)
                if resp!=None:
                    resp.append(v)
                    return resp
            else:
                return  None
def bestRoute(graf:grafo,v1,v2):
    nodes=graf.noVistos.copy()
    graf.vistos.append(v1)
    path=_bestRoute(graf,v2)
    graf.vistos.clear()
    graf.noVistos=nodes.copy()
    return path
```

Ejercicio 12

Dado que el grafo G es un árbol, cumple las siguientes condiciones:

- No tiene ciclos.
- Es conexo, es decir que existe un camino entre para cualquier par de vértices.
- El numero de aristas es igual al número de vértices menos 1.

Si nosotros agregamos una arista más con los vértices que ya se tenían, ocurren 2 cosas, que el numero de aristas ya no va a ser igual al número de vértices menos 1, ya que el número de vértices se mantiene constante. Y segundo que se generaría un ciclo entre dichos vértices, ya que, si los dos vértices antes no estaban unidos, por propiedades de árboles, existe un ancestro en común entre dichos vértices. Por lo cual al añadir un nuevo vértice nuestro grafo ya no es un árbol.

Ejercicio 13

Partiendo de la suposición de que la arista (u,v) pertenece a un grafo G, pero no al árbol BFS de dicho grafo podemos suponer que el nivel de u y el de v diferirá a lo sumo en 1, es decir estarán en el mismo nivel o uno tendrá un nivel mayor que el otro.

Para demostrarlo partimos que u y v son vecinos, ya que existe la arista (u,v) en G. Si armamos el árbol BFS pueden ocurrir 2 cosas, que ambos vértices u y v se descubran al mismo tiempo porque tienen a un 3er vecino en común, o que no tengan a un vecino inmediato en común pero sí que pertenezcan a una misma rama del árbol (es decir que tienen un ancestro en común).

Para el primer caso la arista (u,v) no se incluirá en el árbol BFS ya que ambos vértices estarán encolados para ser visitados luego, y al que primero le toque ser visitado no podrá añadir la arista (u,v) porque el otro vértice ya está “marcado” para visitar. En dicho caso tanto u y v tienen el mismo nivel y la arista (u,v) no aparece en el árbol.

En el segundo caso puede ocurrir que al estar en la misma rama, alguno de los 2 nodos sea marcado primero para ser visitado ya que la “distancia” al nodo que genera la rama en común es mas corta para alguno de los vértices, o que la distancia sea la misma. Si la distancia es la misma el caso es similar a que si tuvieran a un 3er vértice en común, ambos serian encolados al mismo nivel, y la arista (u,v) no se incluiría. Si la distancia es

distinta para cada nodo pero es mayor a 1, haría que el primer vértice encolado pueda marcar al otro sin problemas ya que por el otro camino aun no se ha visto a ese nodo y la arista (u,v) aparecerá en el BFS, pero como hemos dicho que no tiene que estar lo que nos queda es que la distancia difiera como mucho en 1. Ahora uno de los vértices será visitado primero que el otro que aun está en la cola.

Por lo tanto para que (u,v) que pertenece al grafo NO aparezca en el BFS u y v tienen una diferencia de nivel como mucho de 1, ya que sino de otra forma (u,v) estará en el árbol BFS.