

| ***Alumno(s)*** | | ***Nota*** |
| --- | --- | --- |
| ***Rodriguez Ordoñez Juan Daniel*** | |  |
| ***Grupo*** | ***B*** | |
| ***Ciclo*** | ***III*** | |
| ***Fecha de entrega*** | ***03/12/2024*** | |

**I.- OBJETIVOS:**

* Definir las reglas básicas a seguir para la construcción y la correcta interpretación de los Diagramas de Flujo, resaltando las situaciones en que pueden, o deben, ser utilizados.
* Elaborar y Diseñar algoritmos con arreglos de una sola dimensión(unidimensional) denominada vectores

**II.- SEGURIDAD:**

|  | **Advertencia:**  **En este laboratorio está prohibida la manipulación del hardware, conexiones eléctricas o de red; así como la ingestión de alimentos o bebidas.** |
| --- | --- |

**III.- FUNDAMENTO TEÓRICO:**

* Revisar el texto guía que está en el campus Virtual.

**IV.- NORMAS EMPLEADAS:**

* No aplica

**V.- RECURSOS:**

* En este laboratorio cada alumno trabajará con un equipo con Windows 10.

**VI.- METODOLOGÍA PARA EL DESARROLLO DE LA TAREA:**

* El desarrollo del laboratorio es individual.

**VII.- PROCEDIMIENTO:**

**EJERCICIO DE APLICACIÓN**

1.- Graph – Implementación con matrix adyacente

'''

Represent Vertex in Graph

'''

class Vertex:

def \_\_init\_\_(self, node):

self.id = node

def getId(self):

return self.id

def setId(self, id):

self.id = id

'''

Represent a Graph

'''

class Graph:

def \_\_init\_\_(self, numVertices, cost=-1):

self.adjMatrix \

= [[cost for u in range(numVertices)]

for v in range(numVertices)]

self.numVertices = numVertices

self.vertices = []

for i in range(0, numVertices):

newVertex = Vertex(i)

self.vertices.append(newVertex)

def setVertex(self, vtx , id ):

if 0 <= vtx < self.numVertices:

self.vertices[vtx].setId(id)

def getVertexIndex(self, n):

for vertxin in range(0, self.numVertices):

if n == self.vertices[vertxin].getId():

return vertxin

return (-1)

def getVertex(self, index):

return self.vertices[index]

def addEdge(self, frm, to, cost = 0):

if self.getVertexIndex(frm) != -1 \

and self.getVertexIndex(to) != -1:

idx\_frm = self.getVertexIndex(frm)

idx\_to = self.getVertexIndex(to)

self.adjMatrix[idx\_frm][idx\_to] = cost

''' For directed graph do not add this'''

# self.adjMatrix[idx\_to][idx\_frm] = cost

def getVertices(self):

vertices = []

for vertxin in range(0, self.numVertices):

id\_vert = self.vertices[vertxin].getId()

vertices.append(id\_vert)

return vertices

def printMatrix(self):

for u in range(0, self.numVertices):

row = []

for v in range(0, self.numVertices):

row.append(self.adjMatrix[u][v])

print(row)

def getEdges(self):

edges = []

for v in range(0, self.numVertices):

for u in range(0, self.numVertices):

if self.adjMatrix[u][v] != -1:

vid = self.vertices[v].getId()

uid = self.vertices[u].getId()

edges.append((vid, uid, self.adjMatrix[u][v]))

return edges

Ejecutar el siguiente código

if \_\_name\_\_ == '\_\_main\_\_':

G = Graph(6)

G.setVertex(0, 'a')

G.setVertex(1, 'b')

G.setVertex(2, 'c')

G.setVertex(3, 'd')

G.setVertex(4, 'e')

G.setVertex(5, 'f')

print('Graph data:')

G.addEdge('a', 'e', 10)

G.addEdge('a', 'c', 20)

G.addEdge('c', 'b', 30)

G.addEdge('b', 'e', 40)

G.addEdge('e', 'd', 50)

G.addEdge('f', 'e', 60)

G.printMatrix()

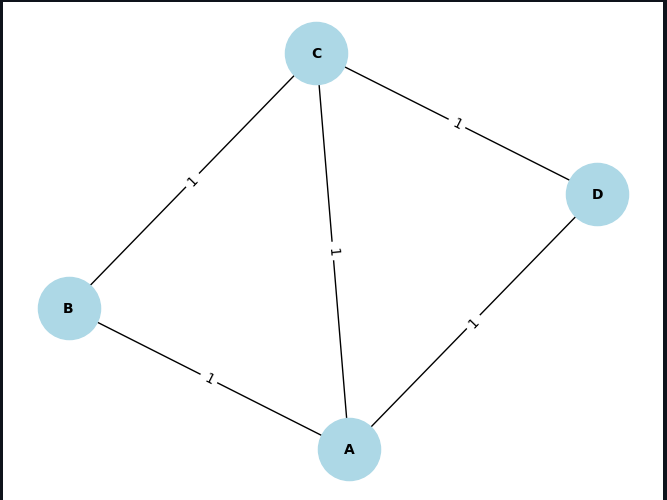
print(G.getEdges())

* Dibujar la matriz adyacente del ejercicio anterior, se muestra un ejemplo

Imagen que contiene cuarto, oscuro, reloj, agua

Descripción generada automáticamente

* Dibuje el grafo



2.- Graph – Implementación con lista adyacente

'''

Represent Vertex in Graph

'''

class Vertex:

def \_\_init\_\_(self,key):

self.id = key

self.adjList = {}

def addNeighbor(self,nbr,weight=0):

self.adjList[nbr] = weight

def getAdjLists(self):

return self.adjList.keys()

def getId(self):

return self.id

def getWeight(self,nbr):

return self.adjList[nbr]

'''

Represent a Graph

'''

class Graph:

def \_\_init\_\_(self):

self.vertices = {}

self.numVertices = 0

def addVertex(self,key):

self.numVertices += 1

newVertex = Vertex(key)

self.vertices[key] = newVertex

return newVertex

def getVertex(self,n):

if n in self.vertices:

return self.vertices[n]

else:

return None

def addEdge(self, f, t, weight = 0):

if f not in self.vertices:

nv = self.addVertex(f)

if t not in self.vertices:

nv = self.addVertex(t)

self.vertices[f] \

.addNeighbor(self.vertices[t],

weight)

def getVertices(self):

return self.vertices.keys()

def getEdges(self):

edges = []

for v in self.vertices.values():

for w in v.getAdjLists():

vid = v.getId()

wid = w.getId()

edges.append((vid, wid,

v.getWeight(w)))

return edges

Ejecutar el siguiente código

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph()

for i in range(6):

g.addVertex(i)

#print(g.vertList)

g.addEdge(0,1,5)

g.addEdge(0,5,2)

g.addEdge(1,2,4)

g.addEdge(2,3,9)

g.addEdge(3,4,7)

g.addEdge(3,5,3)

g.addEdge(4,0,1)

g.addEdge(5,4,8)

g.addEdge(5,2,1)

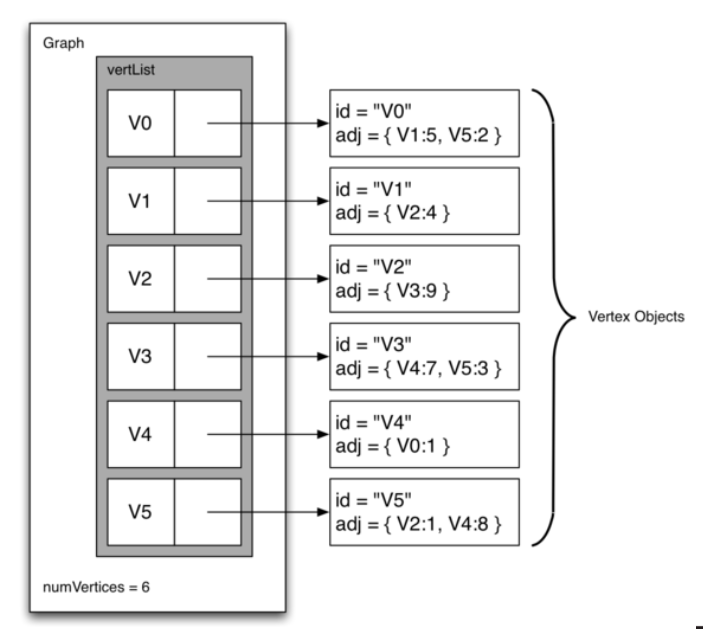
for v in g.vertices.values():

print(f"vertice = {v.getId()}")

for w in v.getAdjLists():

print(f" {v.getId()} -> {w.getId()}:{v.getWeight(w)}")

* Dibujar la lista adyacente del ejercicio anterior.



**CONCLUSIONES:**

1. **Los grafos con matrices y listas de adyacencia permite modelar y resolver problemas relacionados con conexiones.**
2. **Los grafos poseen vértices y aristas que serán utilizados para la creación de redes y conexiones.**
3. **Se creó un grafo con matriz de adyacencia, utilizando pesos en aristas y otra con lista adyacente que ofrece una representación más dinámica y flexible de las conexiones entre vértices.**