	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jesús Cruz Navarro
Asignatura:	Estructuras de Datos y Algoritmos 2
Grupo:	1
No de Práctica(s):	6
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	32
Semestre:	2021-1
Fecha de entrega:	15 de noviembre de 2020
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA 6: GRAFOS 1

Objetivo: El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para entender el algoritmo de búsqueda por expansión.

a) Diseñar e implementar las clases **Vértice** y **Grafo**, con los métodos **AgregarVertice** y **AgregarArista**, como los vistos en clase (usando como parámetros los nombres de los vértices, en lugar de pasar un objeto de tipo **Vértice**, como la práctica de la coordinación).

Se debe validar que:

- En la clase **Vértice**, al agregar vecino, no exista ya un vecino con ese nombre de vértice.
- En la clase **Grafo**, al agregar vértice, no exista ya un vértice con ese nombre.
- En la clase **Grafo**, al agregar arista, que existan los vértices entre la arista.

En caso de error, se debe imprimir un mensaje de error en consola.

Los vértices se deben guardar en un diccionario y acceder a ellos utilizando su llave, no iterando sobre estos.

Además, ambas clases deben sobrescribir los métodos **__str__** y **__repr__** para poder imprimir el grafo desde la función **print()**. Para el caso de la clase grafo al imprimir se debe imprimir algo similar a una lista de adyacencia.

```python
class Graph:
    def __init__(self):
        self.vertices = {}

    def getVertex(self, nodeName):
        return self.vertices[nodeName]

    def addVertex(self, nodeName):
        if nodeName in self.vertices:
            raise KeyError("Repeated key")
        newVertex = Node(nodeName)
        self.vertices[nodeName] = newVertex

    def addEdge(self, nodeName1, nodeName2):
        if nodeName1 not in self.vertices or nodeName2 not in self.vertices:
            raise KeyError("Cannot add edge to non-existing node")
        node1 = self.getVertex(nodeName1)
        node2 = self.getVertex(nodeName2)
        node1.addNeighbor(node2)
```

Clase Graph

Se implementó la clase **Graph** con los requerimientos planteados de la misma forma en que ya fue analizado en clase. Nótese que, en caso de que las condiciones de validación requeridas se decidió generar una excepción **KeyError**.

```python
def __str__(self):
    base = ""
    for node in self.vertices.values():
        base += str(node) + "\n"
    return base

def __repr__(self):
    base = ""
    for node in self.vertices.values():
        base += repr(node) + "\n"
    return base
```

Métodos __str__ y __repr__ de la clase Graph

Además, se implementaron los métodos **__str__** y **__repr__**, utilizando los métodos respectivos de la clase **Node** para cumplir con el objetivo.

```python
class Node:
    def __init__(self, name):
        self.name = name
        self.neighbors = []
        self.color = Color.WHITE
        self.distance = math.inf
        self.parent = None

    def addNeighbor(self, newNeighbor):
        for neighbor in self.neighbors:
            if neighbor.name == newNeighbor.name:
                raise KeyError("Repeated key")

        self.neighbors.append(newNeighbor)
```

Clase Node

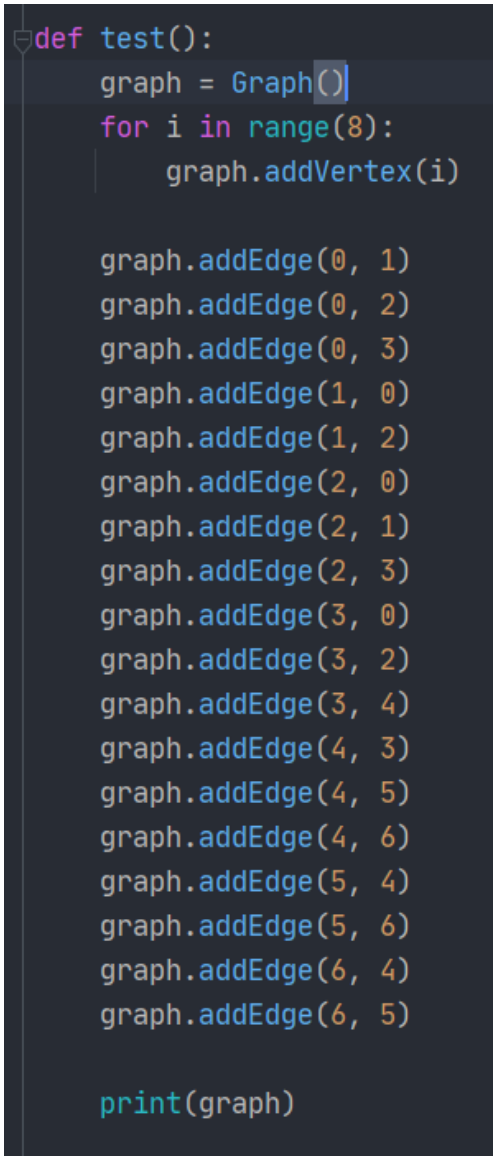
Se implementó la clase **Node** con los requerimientos solicitados. De igual forma, se utilizó la excepción **KeyError** para validar que el nodo a insertar como vecino no se encontrara en la lista.

```python
def __neighborsToString(self):
    base = "[ "
    numNeighbors = len(self.neighbors)

    for i in range(numNeighbors - 1):
        base += str(self.neighbors[i].name) + ", "

    last = self.neighbors[numNeighbors - 1].name if numNeighbors > 0 else ""
    return base + str(last) + " ]"

def __str__(self):
    return str(self.name) + " → " + self.__neighborsToString()

def __repr__(self):
    base = "Name: {0}\nColor: {1}\nDistance: {2}\n{3}\nNeighbors: {4}\n"
    hasParent = "Does not have a parent node" if self.parent is None else ("Parent name: " + str(self.parent.name))
    return base.format(self.name, self.color.value, self.distance, hasParent, self.__neighborsToString())
```

Métodos __str__ y __repr__ de la clase Node

Finalmente, para obtener las representaciones en forma de cadena se implementaron los métodos mostrados. En el caso de **__str__**, se utilizó el formato propuesto en la práctica, es decir, parecido al de una representación con listas de adyacencia. En cambio, en el método **__repr__** se decidió retornar una representación un poco más detallada con las demás propiedades del nodo.

b) Desarrollar un programa que utilice las clases generadas y genere un grafo como el visto en clase *(Ejemplo Básico)* e imprima su lista de adyacencia. Además, pruebe los casos de error y muestre los mensajes en pantalla.



```
def test():
    graph = Graph()
    for i in range(8):
        graph.addVertex(i)

    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(0, 3)
    graph.addEdge(1, 0)
    graph.addEdge(1, 2)
    graph.addEdge(2, 0)
    graph.addEdge(2, 1)
    graph.addEdge(2, 3)
    graph.addEdge(3, 0)
    graph.addEdge(3, 2)
    graph.addEdge(3, 4)
    graph.addEdge(4, 3)
    graph.addEdge(4, 5)
    graph.addEdge(4, 6)
    graph.addEdge(5, 4)
    graph.addEdge(5, 6)
    graph.addEdge(6, 4)
    graph.addEdge(6, 5)

    print(graph)
```

Inicialización del grafo e impresión

Como se logra observar en la captura de pantalla anterior, primero se inicializó el grafo, añadiendo los vértices correspondientes y las aristas entre ellos. Después, se imprimió la representación simple de éste.

```python
# Agregar vértice repetido a grafo
try:
    graph.addVertex(0)
except KeyError:
    print("No es posible agregar un nodo repetido al grafo")

# Agregar vecino repetido a nodo
try:
    node0 = graph.getVertex(0)
    node1 = graph.getVertex(1)
    node0.addNeighbor(node1)
except KeyError:
    print("No es posible agregar un vecino repetido al nodo")

# Agregar arista a nodo inexistente
try:
    graph.addEdge(-1, 0)
except KeyError:
    print("No es posible agregar una arista a un nodo inexistente")
```

Validación de casos límite

Finalmente, se comprobó el funcionamiento de la validación para los casos límite requeridos: agregar un nodo repetido al grafo, agregar un nodo repetido a los vecinos de un nodo, y agregar una arista a un nodo inexistente. Se imprimió una breve descripción de la situación presentada en cada caso.

A continuación, se muestra una captura de pantalla de la salida del programa:

```
0 → [ 1, 2, 3 ]
1 → [ 0, 2 ]
2 → [ 0, 1, 3 ]
3 → [ 0, 2, 4 ]
4 → [ 3, 5, 6 ]
5 → [ 4, 6 ]
6 → [ 4, 5 ]
7 → [  ]

No es posible agregar un nodo repetido al grafo
No es posible agregar un vecino repetido al nodo
No es posible agregar una arista a un nodo inexistente
```

Salida del programa

CONCLUSIONES

Los **grafos** son una de las estructuras matemáticas más importantes en la actualidad: nos permiten representar cierta información y fenómenos en una forma simple y apropiada para muchos problemas. Debido a lo anterior, es importante representarlas de forma eficiente en dispositivos de cómputo, para así lograr resolver problemas que las involucren de forma satisfactoria. Existen diferentes formas de realizar lo anterior, siendo las más comunes las **listas y matrices de adyacencia**, por ello, debemos identificar sus ventajas y desventajas para determinar si es pertinente su utilización en determinada aplicación.

Además, es importante comprender y analizar los algoritmos que operan sobre ellas. Dos de los más importantes son **Breadth First Search** y **Depth First Search**, que nos permiten realizar búsquedas en el grafo respectivo, y con ello, resolver los problemas planteados en cada caso.