Guía práctica de estudio 6

Algoritmos de Grafos. Parte 1.

Elaborado por:

M.I. Elba Karen Sáenz García

Revisión:

Ing. Laura Sandoval Montaño

Facultad de Ingeniería

U.N.A.M.

Guía Práctica 6

Estructura de datos y Algoritmos II

Algoritmos de Grafos. Parte 1.

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

Actividades

Implementar la búsqueda por expansión en un grafo representado por una lista de adyacencia en algún lenguaje de programación.

Antecedentes

- Análisis previo del concepto de grafo, su representación y algoritmo visto en clase teórica.
- Manejo de listas, diccionarios, estructuras de control, funciones y clases en Python 3.
- Conocimientos básicos de la programación orientada a objetos.

Introducción

Un grafo es una entidad matemática introducida por Leonhard Euler en 1736 con el trabajo de los 7 puentes de Konigsberg, donde se formula el problema de cómo recorrer 7 puentes del centro de la ciudad de manera que se pasará solo una vez por ellos y se pudiera regresar al punto de partida.

Euler planteó el problema representando cada parte de tierra como un punto (llamado vértice) y cada puente como una línea (conocidas como aristas). Fig. 6.1.

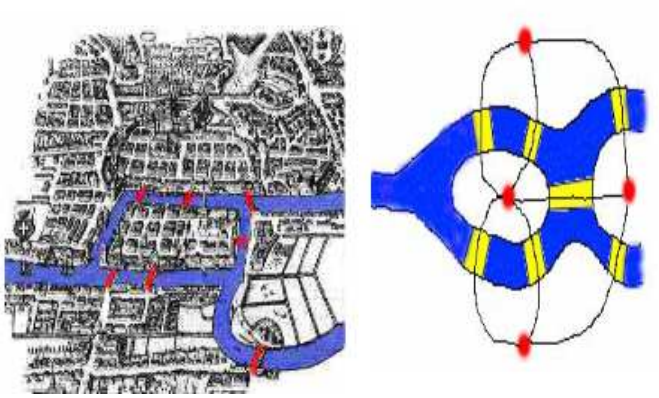


Figura 6.1.

Entonces un grafo permite representar entidades (vértices) y sus relaciones entre sí (aristas), por lo que permiten modelar problemas que se definen mediante las conexiones entre objetos o entidades; por ejemplo, en ingeniería

para la representación de una red de cualquier tipo como de transporte (tren, carretera, avión), de servicios (eléctrica, gas, agua, comunicaciones), redes de internet, etc. Otros ejemplos pueden ser para representar estrategias de pase de un balón en un equipo de futbol, conexiones en circuitos lógicos entre otros.

Al modelar un problema mediante un grafo, las relaciones derivadas de las conexiones entre las entidades u objetos se pueden utilizar para responder a preguntas importantes para la solución de un problema como: ¿cuál es la menor distancia entre un objeto y otro?, ¿existe un camino para ir de un objeto a otro siguiendo las conexiones?, ¿cuántos objetos se pueden alcanzar a partir de uno determinado?

Definiciones

Grafo: Un grafo es un par ordenado $G = (V, A)$ donde V es el conjunto finito no vacío de elementos llamados vértices (o nodos) y A es el conjunto de pares no ordenados (i, j) donde i, j pertenecen a V, a este conjunto se le llama aristas(o arcos).

Un grafo $G = (V, A)$ contiene $|V|$ vértices y $|A|$ aristas, y el nombre dado a los vértices serán valores comprendidos entre 0 y $|V|$

Dependiendo de la importancia del orden de los vértices se tienen:

Grafos Dirigidos: Es un grafo donde A es un conjunto de aristas con una relación binaria en V, es decir el orden importa, la arista $(i, j) \neq (j, i)$. El que el vértice i esté conectado con el vértice j no implica que el vértice j esté conectado con el vértice i. En otras palabras cuando las aristas tienen asignadas direcciones.

En la figura 6.2 se observan un grafo dirigido con un conjunto de vértices $V = \{0,1,2,3,4,5\}$ y un conjunto de aristas $A = \{(0,1), (0,3), (1,2), (1,3), (2,2), (2,3), (3,0), (5,4)\}$ [1]

En grafos dirigidos pueden existir aristas de un vértice a sí mismo, denominadas aristas cíclicas o self-loops.

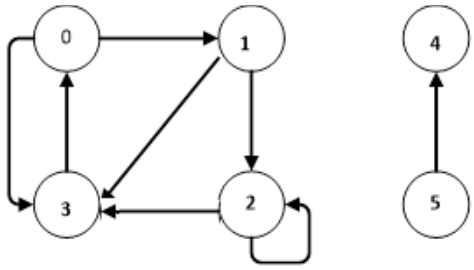


Figura 6.2.

Grafo no dirigido: Es un grafo donde el conjunto de las aristas es un conjunto de pares no ordenados, la arista (i, j) y (j, i) se consideran una única y misma arista. En un grafo no dirigido no se permiten las aristas cíclicas.

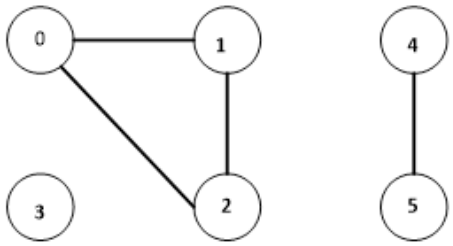


Figura 6.3

En la figura 6.3 se muestra un grafo no dirigido sobre un conjunto de vértices $V = \{0,1,2,3,4,5\}$ y el conjunto de aristas $A = \{(0,1), (0,2), (1,2), (4,5)\}$

Subgrafo: Un subgrafo de $G = (V, A)$ es un grafo $G' = (V', A')$ tal que $V' \subseteq V$ y $A \subseteq A$

Orden: Es el número de vértices del grafo, el cardinal del conjunto V de vértices: $|V|$

Aristas incidentes a un vértice

En un grafo dirigido se dice que una arista a es **incidente** en un vértice v hacia el **exterior**, si v es el extremo inicial de a y a no es un ciclo. En la figura 6.2 las aristas que salen del vértice 1 son $(1,2)$ y $(1,3)$.

La arista a es **incidente** a un vértice v hacia el **interior** si v es el extremo final de a y a no es un ciclo. En la figura 6.2 la arista que inciden sobre el vértice 1 es $(0,1)$.

En un grafo no dirigido se dice que una arista a es incidente a un vértice v, si v es uno de los extremos de a, y a no es un ciclo.

Adyacencia:

Dos vértices son adyacentes si son distintos y existe una arista que va de uno a otro. En la figura 6.2 y 6.3 el vértice 1 es adyacente al vértice 0 ya que la arista $(0,1)$ pertenece a los dos grafos. Pero en la figura 6.2 el vértice 0 no es adyacente al 1 ya que la arista $(1,0)$ no pertenece al grafo.

Dos aristas son adyacentes, si siendo distintas, comparten un extremo común.

Grado: El grado de un vértice es el número de aristas que inciden en él. Si todos los vértices tienen el mismo grado, se conoce como grafo regular.

En un grafo dirigido se distingue entre semigrado interior o entrante de un vértice y es el número de aristas que llegan a él (inciden hacia el interior) y semigrado exterior o saliente de un vértice y es el número de aristas que salen de él (inciden hacia el exterior).

Grafo etiquetado. Un grafo se dice que está etiquetado, si cada arista tiene asociada una etiqueta o valor de cierto tipo.

Grafo ponderado o con pesos: Es un grafo etiquetado con valores numéricos.

Camino: En un grafo dirigido un camino es una secuencia finita de aristas tal que el extremo final de cada arista coincide con el extremo inicial del siguiente. Al camino se le llama simple cuando no utiliza dos veces la misma arista, en caso contrario se llama camino compuesto.

Longitud del camino: Es el número de aristas del camino y está dado por el número de vértices menos uno.

Circuito o ciclo: Es un camino en el que el vértice final coincide con el inicial.

Representación de los grafos

Un grafo se puede representar mediante una matriz cuadrada $B = [b_{ij}]$ con $|V|x|V|$ elementos y es llamada matriz de adyacencia. En esta matriz cada renglón y cada columna representa un vértice del grafo y la posición (i, j) representa una arista o su ausencia, el extremo inicial de la arista se representa con el renglón i y el extremo final con la columna j. Se puede colocar un 1 en la posición (i, j) si existe un enlace que va de i a j y un 0 en caso contrario.

$$b_{ij}=\begin{cases} 1 & si\ (i,j) \in A \\ 0 & en\ otro\ caso \end{cases}$$

En el caso de un grafo no dirigido la matriz es simétrica lo que no ocurre en grafos dirigidos y con esta característica se puede almacenar solo la mitad de la matriz y ahorrar memoria. En la figura 6.4 se muestra la representación de un grafo no dirigido como una matriz de adyacencia.

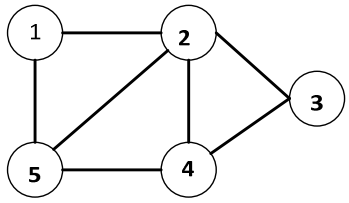
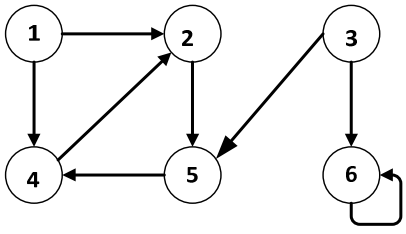


Figura 6.4 [1]

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

En la figura 6.5 se muestra un grafo dirigido representado en una matriz de adyacencia



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figura 6.5[1]

Para grafos ponderados se puede colocar en lugar de 1, el valor del peso de la arista (i, j) en la misma posición y así hacer referencia a su existencia y un valor 0 o NULL para indicar la ausencia de la misma.

Es preferible usar una representación con Matriz de Adyacencia cuando el grafo es denso, es decir, el número de aristas $|A|$ es cercano al número de vértices al cuadro ($|V|^2$) (el grafo es pequeño) y cuando se quiere saber rápidamente si hay un arco o arista conectando dos vértices.

La otra forma de representar un grafo es mediante una lista de adyacencia, Figura 6.6. En este caso el Grafo $G = (V, A)$ consiste de un arreglo Adj que almacena $|V|$ listas, una para cada vértice o nodo en V. Para cada $u \in V$, la lista de adyacencia $Adj[u]$ contiene (la referencia a) todos los vértices v tal que hay una arista $(u, v) \in A$. Esto es $Adj[u]$ se conforma de todos vértices adyacentes a v en el grafo G [2].

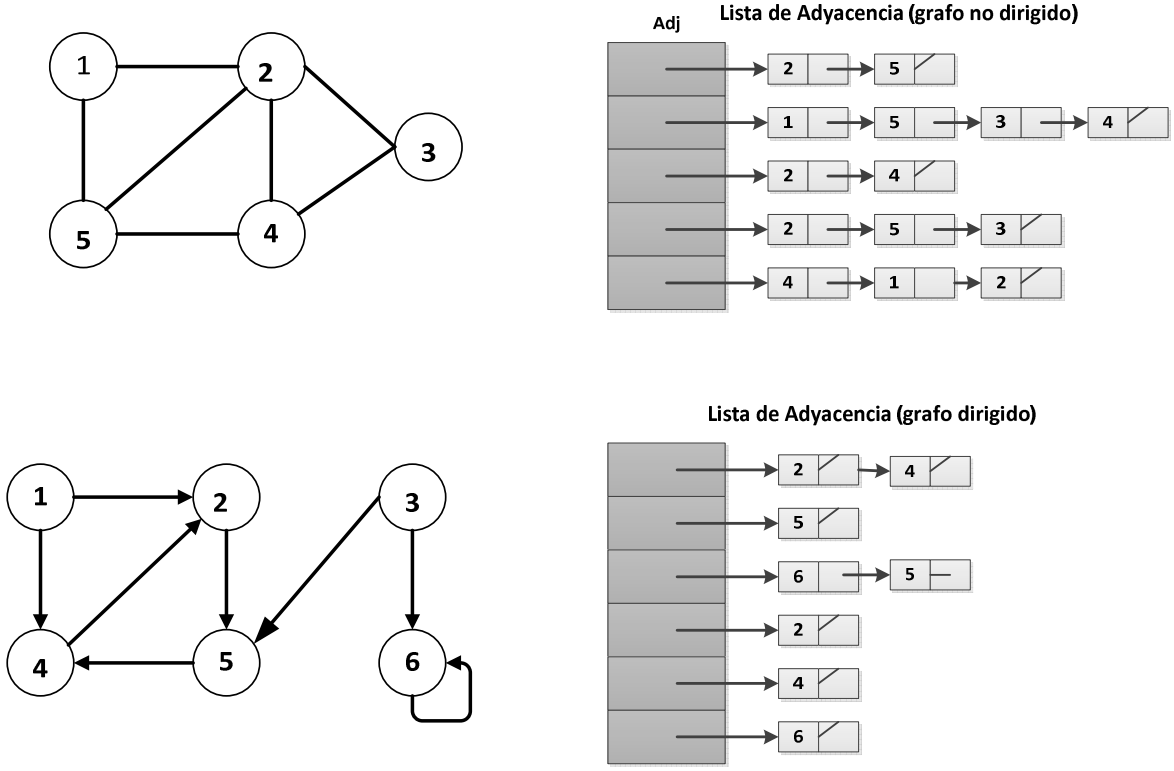


Figura 6.6 [1]

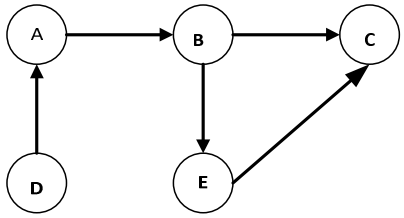
Si el grafo es dirigido, se cumple que la suma de los largos de las listas de adyacencia es $|A|$.

Si el grafo no es dirigido, se cumple que la suma de los largos de las listas de adyacencia es 2*|A|. Dado que cada arco aparece dos veces.

Las listas de adyacencia pueden ser fácilmente adaptadas para representar grafos con peso. En estos un peso es asociado a cada arista a través de una función de peso w: $A \rightarrow R$. Así el peso de la arista (u, v) es puesto en el nodo v de la lista u.

Otra representación frecuente con grafos no orientados es la llamada matriz de incidencia Figura 6.7, que es una matriz $B = [b_{ij}]$ de $|V|x|A|$ tal que:

$$b_{i,j} = \begin{cases} -1 \ si\ la\ arista\ j\ sale\ del\ vertice\ i \\ 1\ si\ la\ arista\ j\ incide\ en\ el\ vertice\ j \\ 0\ En\ otro\ caso \end{cases}$$



A	B	C	D	D
0	1	0	-1	0
-1	0	1	0	1
0	-1	0	0	-1
1	0	0	0	0
0	-1	1	0	0

Figura 6.7

Algoritmos de exploración de grafos

Existen dos algoritmos importantes para la exploración de grafos que realizan un recorrido para visitar de forma eficiente cada vértice y arista, estos son:

- Búsqueda por expansión, también llamado búsqueda primero en amplitud o anchura
- Búsqueda por profundidad, también llamado búsqueda primero en profundidad.

Ambos algoritmos se ejecutan en un tiempo que crece linealmente al crecer el tamaño del grafo. [3]

Búsqueda primero en anchura (Breadth-first search- BFS)

Este algoritmo es uno de los más simples para explorar un grafo y es prototipo para otros algoritmos importantes como el algoritmo de Prim, que obtiene el árbol generador mínimo o árbol de recubrimiento mínimo y el algoritmo de Dijkstra que obtiene el camino más corto de un vértice a los otros vértices. [2]

Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto a través de la anchura de la frontera, es decir, llega a los nodos de distancia k, sólo luego de haber llegado a todos los nodos a distancia k-1.

En el algoritmo BFS, dado un grafo $G = (V, A)$ y un vértice s , se exploran sistemáticamente las aristas de G para descubrir cada vértice que es alcanzable desde s , además calcula la menor distancia (menor número de arcos) de s a cada vértice alcanzable.

Una característica del algoritmo es que para cualquier vértice v alcanzable desde s , en la búsqueda, se va formando un árbol con raíz en s que contiene la ruta más corta y simple de s a v en G.

Para visualizar el progreso del algoritmo se colorea cada vértice de blanco, gris o negro. Todos los vértices inician en blanco y posteriormente se pueden colorear en gris y finalmente en negro, considerando lo siguiente [1]:

- Cuando se descubre un vértice por primera vez en el proceso de búsqueda, se colorea de gris.
- Si un vértice es gris o negro significa que ya ha sido descubierto, pero hay una distinción entre ellos para asegurar que la búsqueda se realice realmente en anchura.
- Si $(u, v) \in A$ y u es negro, entonces el vértice v es gris o negro, lo que significa que todos los vértices adyacentes a los vértices negros ya han sido descubiertos.
- Los vértices grises pueden tener algunos vértices adyacentes blancos que representan la frontera entre los vértices descubiertos y no descubiertos.

A continuación, se muestra el algoritmo en pseudocódigo de una función para realizar la búsqueda primero en anchura (BFS) en un grafo $G = (V, A)$ y partiendo de un vértice s [1]. Es importante mencionar que en el algoritmo que se describe se considera lo siguiente:

- El grafo está representado por una lista de adyacencia.
- A cada vértice se le agregan los atributos de color, distancia y predecesor, donde; el atributo color de un vértice u $(u.color)$ puede ser blanco, gris o negro; el atributo distancia $(u.d)$ mantiene la distancia de s al vértice u y el atributo predecesor $(u.p)$ contiene información del predecesor del vértice u. Si no tiene se le asigna NULL.
- Se utiliza una cola Q (First in- First out), para la gestión de los vértices grises, o los que han sido descubiertos.

```
BFS(G,s)
Inicio
    Para cada vértice u ∈ V − {s}
        u.color=White
        u.d = ∞
        u. p=NULL
    Fin Para
    s.color=Gris
    s.d=0
    s.p=NULL
    Q = ∅
    Encolar(Q,s)
    Mientras Q ≠ ∅
        u=Desencolar(Q)
        Para cada vértice v que es adyacente al vértice u
            Si v.color== Blanco
                v.color=Gris
                v.d=u.d+1
                u.p=u
                Encolar (Q,v)
            Fin Si
        Fin Para
        u.color=Negro
    Fin Mientras
Fin
```

En la función BFS() primero se inician todos los vértices, a excepción de s, en blanco, se les coloca el atributo distancia en $u.d = \infty$ y el predecesor $u.p = NULL$.

Después se dan valores iniciales al vértice s para después encolarlo en la cola Q.

El ciclo *Mientras* funciona si se tienen vértices grises en la cola, los cuales forman el conjunto de vértices descubiertos cuyos vértices adyacentes (listas de adyacentes) todavía no se han revisado por completo. Antes de iniciar el ciclo *Mientras*, el único vértice que se encuentra en la cola es el vértice origen s.

La primera instrucción del ciclo Mientras es sacar de la cola al vértice u que se encuentre al inicio, para después con la estructura de repetición *Para* revisar todos los vértices v adyacentes a u . Si algún vértice v de los revisados, está coloreados en blanco, significa que apenas se descubrió y el algoritmo lo descubre colocando en sus atributos $v.color = Gris$, $v.d = u.d + 1$, $v.p = u$, una vez que se descubre el vértice v, se encola en Q, donde están los vértices descubiertos. Cuando ya se han revisado todos los vértices v adyacentes a u, u se coloca en negro.

El algoritmo trabaja con grafos dirigidos y no dirigidos.

Como ejemplo se va a realizar el recorrido en el siguiente grafo, Figura 6.8. Donde el vértice origen es s, se asignan valores iniciales a los demás y en la cola se agrega a s.

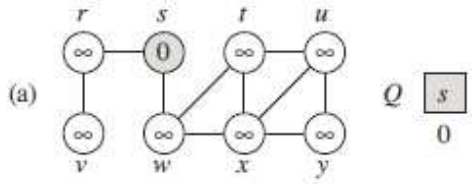


Figura 6.8[1]

Se desencola s para revisar sus vértices adyacentes (w y r), que apenas se descubrieron y están en blanco. Al descubrirse se colocan en gris y se agregan a la cola.

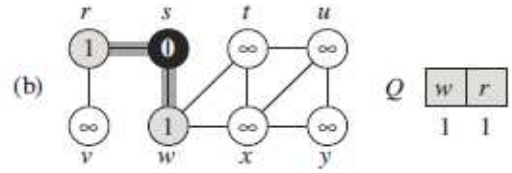


Figura 6.9[1]

Se desencola w y se revisan sus adyacentes, que son s, t, x, y los que están en blanco (t y x) se descubren colocándose en gris, se les aumenta el atributo distancia y se le añaden a la cola.

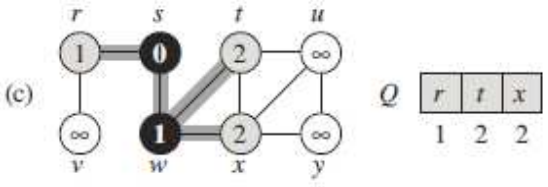


Figura 6.10[1]

Ahora se desencola r y se revisan sus adyacentes que son s y v pero solo v se descubre, se colorea de gris, se le modifica la distancia, y se agrega a la cola.

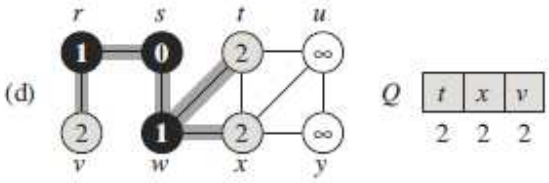


Figura 6.11[1]

El procedimiento continúa como se muestra en las siguientes secuencias, Figura 6.12. Hasta que todos los vértices ya están en negro y se obtiene la distancia d del vértice s a cada vértice alcanzable. Notar que la distancia d se observa dentro del nodo o vértice.

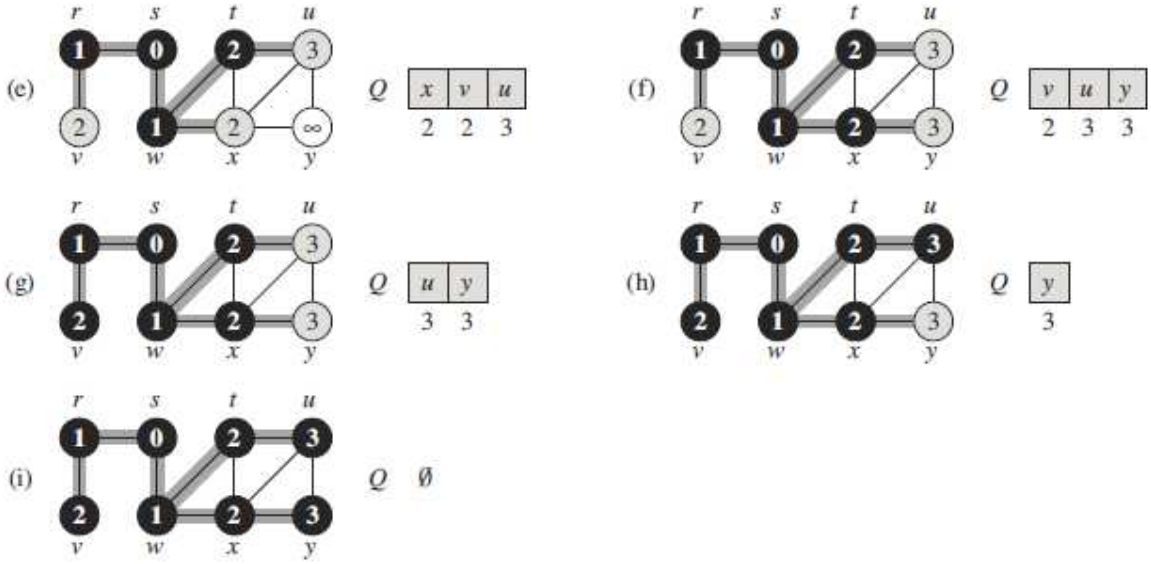


Figura 6.12[1]

Desarrollo

En esta práctica se involucra el uso de la programación orientada a objetos en Python, se introduce el uso de clases con sus atributos y métodos; la creación de un objeto, y su inicialización con el uso del método constructor al instanciar el objeto.

Antes de explicar la implementación del algoritmo BSF en Python se dan las definiciones básicas para recordar conceptos que se han visto en la asignatura de POO y como se usa en Python.

Clases: Las clases son los modelos o plantillas sobre los cuales se construirán los objetos, están formadas por atributos y métodos. En Python, una clase se define con la instrucción *class* seguida de su nombre y dos puntos, por ejemplo:

```
class nomClase:
```

La clase más elemental, es una clase vacía

```
class Clase:
  pass
```

donde la declaración pass indica que no se ejecutará ningún código, pero si se pueden instanciar objetos de la clase, por ejemplo:

```
objeto1 = Clase()  # Crea objeto1 de la clase Clase
objeto2 = Clase()  # Crea objeto2 de la clase Clase
```

Atributos. Las propiedades o atributos, son las características propias del objeto y modifican su estado. Estas se representan a modo de variables, los dos tipos de atributos o de variables existentes son variables de clase y variables de instancia (objetos). Ejemplo:

```
class Llanta():
  tipo = "variable de clase"
  longitud = ""

class Automovil():
  color = ""
  tamaño = ""
  modelo = ""
  llantas = Llanta() # atributo compuesto por un objeto de la clase Llanta
```

Métodos. Los métodos son *funciones*, y representan acciones propias que puede realizar el objeto:

```
class  Automovil():
    color = "azul"      #atributos
    tamaño = "grande"
    modelo = "VW"
    llantas = Llanta()
    def frenar(self): #metodo     #métodos
        pass
```

Importante: Notar que el primer parámetro de un método, siempre debe ser *self*

Método Constructor

El método __init__ es el encargado de fungir como método constructor, es decir que este va a inicializar una serie de atributos y ejecutar el código que le definamos al momento de que se cree un objeto de la clase.

```
class Clase:
  def __init__(self):
        self.variable=42
```

Al crear el objeto obj de la forma, $obj = Clase()$, éste tiene su atributo *variable* inicializado con un valor de 42.

Si se quiere que los atributos se inicialicen de forma dinámica, se puede hacer lo siguiente:

```
class Clase:
  def __init__(self, valor=42):
  self.variable = valor
```

Actividad 1

Realizar un programa que represente un grafo mediante una lista de adyacencia y el paradigma orientado a objetos. A continuación, se describe brevemente un diseño e implementación en Python que ayudará al desarrollo del programa.

Considerar dos clases, la clase Vértice y la clase Grafo con sus atributos y métodos, como se muestra en el diagrama de clases de la figura 6.13.

La clase Vértice, tiene como atributos un nombre y una lista de vértices adyacentes o vecinos y como método, agregar un vértice adyacente o vecino a su lista.

La clase grafo tiene como atributo un conjunto de vértices (representados con un diccionario) y como métodos, agregar vértices, agregar arista e imprimir grafo.

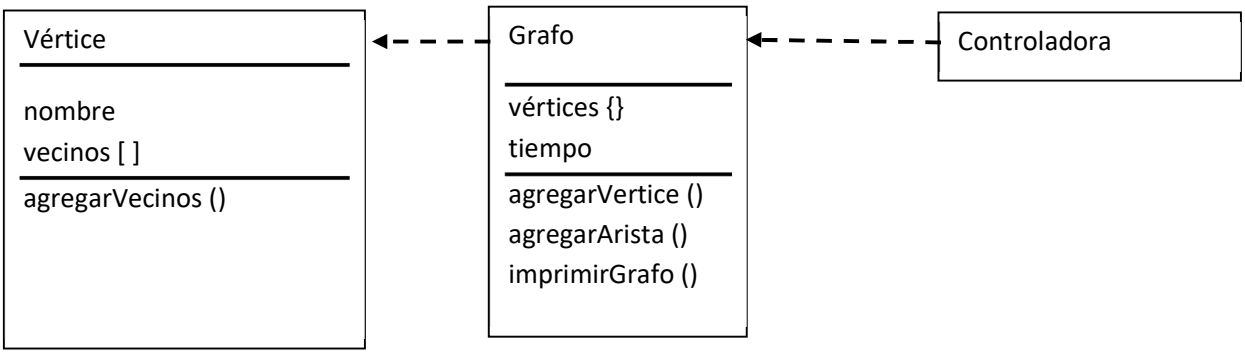


Figura 6.13

Abajo se muestra la implementación en Python de la clase Vértice y Grafo:

```
class Vertice:
    def __init__(self, n):
        self.nombre = n
        self.vecinos = list()


    def agregarVecino(self, v):
        if v not in self.vecinos:
            self.vecinos.append(v)
            self.vecinos.sort()
```

```
class Grafo:
    vertices = {}

    def agregarVertice(self, vertice):
        if isinstance(vertice, Vertice) and vertice.nombre not in self.vertices:
            self.vertices[vertice.nombre] = vertice
            return True
        else:
            return False

    def agregarArista(self, u, v):
        if u in self.vertices and v in self.vertices:
            for key, value in self.vertices.items():
                if key == u:
                    value.agregarVecino(v)
                if key == v:
                    value.agregarVecino(u)
            return True
        else:
            return False

    def imprimeGrafo(self):
        for key in sorted(list(self.vertices.keys())):
            print("Vertice "+key +" Sus vecinos son"+ str(self.vertices[key].vecinos))
```

La siguiente clase, es la controladora, donde se coloca la secuencia de código que permite formar un grafo utilizando las clase *Vértice* y *Grafo* y cuyos vértices se representan con letras mayúsculas del alfabeto. Primero se crea un objeto 'g' de la clase *Grafo* y después se crean vértices que se van agregando al grafo.

```
#Clase controladora
class Controladora:
    def main(self):
        #Se crea un objeto 'g' de la clase Grafo, el grafo
        g = Grafo()
        #Se crea un objeto 'a' de la clase Vertice,un vertice
        a = Vertice('A')
        # se agrega el vertice a al grafo
        g.agregarVertice(a)

        # Esta estructura de repetición es para agragar
        #todos los vertices, y no hacerlo uno a uno
        for i in range(ord('A'), ord('K')):
            g.agregarVertice(Vertice(chr(i)))

        # Se declara una lista que contiene las aristas del grafo
        edges = ['AB', 'AE', 'BF', 'CG', 'DE', 'DH', 'EH', 'FG', 'FI', 'FJ', 'GJ']

        # Se agregan las aristas al grafo
        for edge in edges:
            g.agregarArista(edge[:1], edge[1:])
        # Se imprime el grafo, como lista de adyacencia
        g.imprimeGrafo()
```

Para la ejecución del programa, se crea un objeto de la clase controladora y después se llama a la función main():

```
obj = Controladora()
obj.main()
```

Una vez terminado el programa probarlo con tres grafos no dirigidos propuestos.

Actividad 2

Realizar las modificaciones que se describen abajo a las clases *Vértice* y *Grafo* para que en el programa realizado se implemente el algoritmo en pseudocódigo de BSF explicado anteriormente.

Lo primero es agregar los atributos de color, distancia y predecesor a la clase Vértice como sigue:

```
class Vertice:
    def __init__(self, n):
        self.nombre = n
        self.vecinos = list()
        self.distancia = 9999
        self.color = 'white'
        self.pred = -1


    def agregarVecino(self, v):
        if v not in self.vecinos:
            self.vecinos.append(v)
            self.vecinos.sort()
```

Ahora adicionar el método *bsf()* a la clase Grafo. El método realiza la **Búsqueda primero en anchura** y es el siguiente:

```
def bfs(self, vert):
    vert.distancia = 0
    vert.color = 'gris'
    vert.pred=-1
    q=list()

    q.append(vert.nombre)

    while len(q) > 0:

        u = q.pop()
        node_u = self.vertices[u]
        for v in node_u.vecinos:
            node_v = self.vertices[v]
            if node_v.color == 'white':
                node_v.color='gris'
                node_v.distancia=node_u.distancia + 1
                node_v.pred=node_u.nombre
                q.append(v)
        self.vertices[u].color='black'
```

Además también se modifica el método *imprimeGrafo()* para mostrar las distancias obtenidas con la búsqueda del vértice A a cualquier otro vértice.

```
def imprimeGrafo(self):
    for key in sorted(list(self.vertices.keys())):
        print("Vertice "+key +" Sus vecinos son "+ str(self.vertices[key].vecinos))
        print("La Distancia de A a " + key + " es: "+ str(self.vertices[key].distancia))
```

Una vez terminadas las modificaciones, probar el programa con diferentes grafos propuestos y verificar resultados

Actividad 3

Ejercicios propuestos por el profesor.

Referencias

[1] CORMEN, Thomas, LEISERSON, Charles, et al.
Introduction to Algorithms
3rd edition
MA, USA
The MIT Press, 2009

[2] Ziviani, Nivio
Diseño de algoritmos con implementaciones en Pascal y C
Ediciones paraninfo /Thomson Learning
2007

[3] Baase-Val Gelder
Algoritmos computacionales. Introducción al análisis y diseño
Tercera edición
Addison Wesley

[4]Hernández Figueroa, Rodríguez, del Pino,González Domínguez,Díaz Roca, Pérez Aguilar, Rodríguez Rodríguez
Fundamentos de estructuras de Datos, Soluciones en ADA,JAVA y C++

[5] http://librosweb.es/libro/python/capitulo_5/programacion_orientada_a_objetos.html