Ficha 30

Estructuras No Lineales: Grafos

1.] Introducción y necesidades de uso.

Hasta aquí las estructuras de datos estudiadas permitían almacenar elementos pero no nos decían mucho respecto de las relaciones entre ellos, más allá de la relación de "sucesor" o "antecesor" que surge de la propia disposición de los objetos en cada estructura.

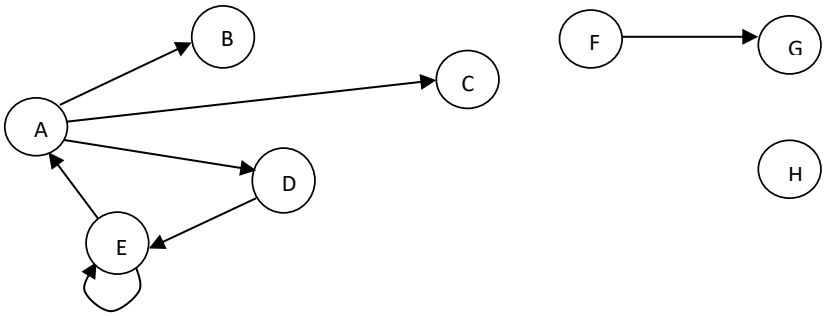
Sin embargo, muchos problemas se centran en colecciones de objetos u elementos y *también* en las relaciones existentes entre ellos. Por ejemplo, un sociólogo querrá almacenar la información sobre los individuos que integran los grupos humanos que estudia, pero también (y con la misma importancia) querrá almacenar la información sobre las relaciones que se dan entre esos individuos (¿habla el individuo ***a*** con el ***b***? ¿comparten momentos fuera del trabajo los individuos ***c*** y ***d***?, etc.) Por otra parte, los directivos de una empresa de servicios viales querrán almacenar información acerca de cada ciudad existente en la región en que prestan servicios, pero también querrán almacenar información acerca de qué caminos (o sea, relaciones) existen entre qué ciudades, para luego poder analizar la forma de optimizar los procesos de transporte entre ellas. Y sólo hemos citado dos ejemplos conocidos del mundo real... que pueden representarse muy bien mediante *grafos*.

Los *grafos* son estructuras de datos que permiten modelar con relativa sencillez las *relaciones* que existen entre los diversos *objetos* del dominio de un problema. Muchos de los problemas referidos a grafos se han estudiado muy bien, existiendo soluciones eficientes muy conocidas y aplicadas. Otros problemas están bien estudiados, pero no se conocen soluciones eficientes (o sea, soluciones que no consistan en analizar por *fuerza bruta* todas las combinaciones de datos posibles, lo cual suele llevar a algoritmos que resultan inaplicables cuando el número de datos es grande). Mucha de la investigación que se realiza en el mundo de las ciencias de la computación tiene que ver con problemas referidos a grafos.

Un ***grafo*** *es una estructura de datos no lineal, compuesta por un conjunto de n nodos (también llamados vértices) y por un conjunto de m arcos (también llamados aristas), de tal forma que un arco une dos vértices cualesquiera*. Notar que en la definición no se imponen reglas en cuanto a qué vértices deben unirse por cuáles arcos, ni cuántos arcos pueden salir de o llegar a un vértice. Esos elementos dependen del problema que se está modelando.

En la *Figura 1* siguiente, mostramos un ejemplo de un *grafo* (en particular, y como veremos, un *grafo dirigido*), compuesto por un conjunto de ocho vértices $V = \{A, B, C, D, E, F, G, H\}$ y por un conjunto de siete arcos $W = \{(A,B), (A,C), (A,D), (D,E), (E,E), (E,A), (F,G)\}$. No importa por el momento exactamente qué representan esos ocho vértices, ni a qué relación se refiere cada uno de los siete arcos, ni tampoco qué significa el sentido u orientación de cada flecha con la que se representa a un arco:

Figura 1: Un grafo dirigido.



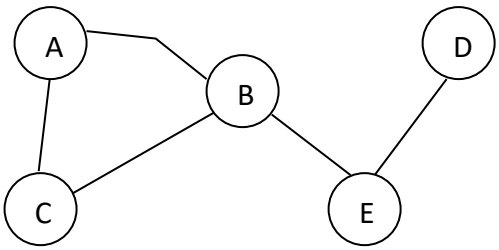
2.] Tipos de grafos.

En la *Figura 1* anterior se muestra la representación gráfica de un *grafo* compuesto por ocho vértices y siete arcos. Por otra parte, dependiendo del problema, pueden darse distintas modalidades de grafos. Si en un grafo importa indicar el *sentido* de cada arco (es decir que cada arco tiene un *vértice de partida* o *inicio* y un *vértice de llegada*), entonces el grafo se dice *dirigido* y se dibuja a cada arco terminado en punta de flecha. Se entiende que el vértice desde el cual parte la flecha es el vértice del partida del arco, y el vértice al cual apunta la flecha es el vértice de llegada [1]:

El grafo de la *Figura 1* es un *grafo dirigido*: los arcos indican relaciones entre nodos, y el sentido de cada arco indica cual es el nodo de partida y cual el nodo de llegada de la relación. Por ejemplo, un *grafo dirigido* como el de la figura citada, puede usarse para modelar un *sociograma*: cada vértice puede representar a un individuo de un grupo, y cada arco puede representar algún tipo de relación entre esos individuos que esté siendo estudiada por un sociólogo o un psicólogo. Así, por ejemplo, si un arco parte del vértice que representa al individuo *A* y llega a un individuo *B*, podría significar que *A* ha elegido a *B* para algún tipo de tarea que se hubiese propuesto, y así modelar una red de preferencias entre los individuos del grupo.

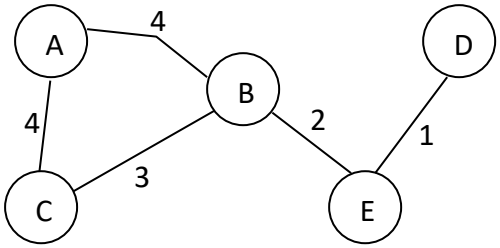
Si el grafo no se requiere que los arcos sean dirigidos (es decir, solo importa indicar que el arco existe pero no cual de sus vértices es el origen y cual el de llegada), entonces el grafo se dice *no dirigido* y sus arcos se representan simplemente como una línea que une dos vértices, pero sin punta de flecha en ninguno de sus extremos: el arco que conecta dos vértices puede entenderse como que puede recorrerse indistintamente en un sentido como en el otro. El grafo de la *Figura 2* es un *grafo no dirigido*. Los grafos no dirigidos pueden usarse, por ejemplo, para modelar una red de caminos que vinculan a distintas ciudades o pueblos de una zona: cada vértice será una ciudad o pueblo, y cada arco indicará la existencia de una ruta directa entre ambas. Es claro que si hay una ruta que va de la ciudad *A* hacia la *B*, entonces la misma ruta existe saliendo de *B* y viajando hacia *A*.

Figura 2: Un grafo no dirigido.



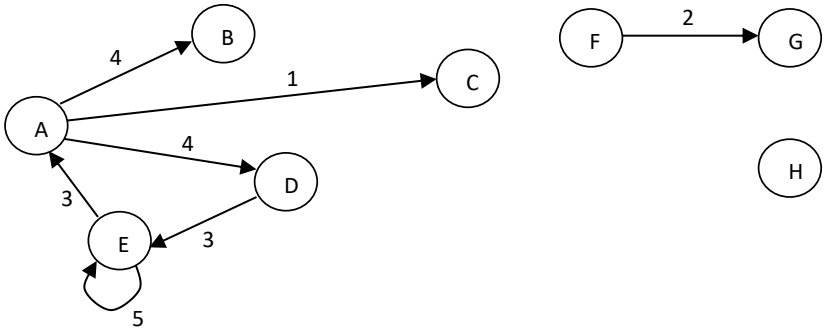
También puede ocurrir que en algunos problemas se requiera que cada arco tenga asociado un *valor*, *peso* o *ponderación*. El grafo puede ser en este caso dirigido o no dirigido, pero se debe asociar a cada arco el *peso* de este. En estos casos, el grafo se dice *ponderado* o con *factores de peso*. El grafo de la *Figura 3* es un *grafo ponderado no dirigido*. Un grafo ponderado no dirigido podría usarse en el mismo caso anterior (un modelo de ciudades y las rutas que las conectan) pero agregando en cada arco la *distancia* entre las ciudades unidas por ese arco (o algún otro valor como podría ser el monto a pagar en concepto de peaje si se circula por esa zona, etc.)

Figura 3: Un grafo ponderado no dirigido.



Por cierto, un grafo ponderado también puede ser dirigido de acuerdo a los requerimientos del dominio del problema. El mismo grafo dirigido de la *Figura 1* podría tener factores de peso como se muestra en la *Figura 4*:

Figura 4: Un grafo ponderado dirigido.



3.] Definiciones y terminología básica.

Más allá de la representación gráfica, un grafo puede ser descripto usando notación de conjuntos. Para el conjunto de vértices basta con una *definición por enumeración* de los valores contenidos en cada vértice. En cuanto al conjunto de arcos, la notación depende del tipo de grafo. Si el grafo es dirigido se enumeran los arcos usando la noción de *par ordenado* para cada uno. Así, el arco que parte del vértice *A* y llega al *B* puede representarse con el par ordenado *(A, B)* en el que el primer elemento del par indica el vértice de partida, y el segundo indica el de llegada. De esta forma, el par *(A, B) no representa* al mismo arco que el par *(B, A)* pues en este último el vértice de partida sería *B*, y el de llegada el *A*. Con estos criterios, el grafo *G = {V, W}* de la *figura 1* (donde *V* es el conjunto de vértices y *W* el de arcos) puede describirse así:

$$G \begin{cases} \text{conjunto de vértices} \quad V = \{A, B, C, D, E, F, G, H\} \\ \text{conjunto de arcos} \quad W = \{(A, B), (A, D), (A, C), (E, E), (D, E), (E, A), (F, G)\} \end{cases}$$

Y está claro que en un grafo no dirigido, el par que represente a un arco será un *par libre* en el cual no tendrá importancia cual sea el vértice de partida y cual el de llegada. En ese sentido, en un grafo no dirigido el par *(A, B)* hará referencia indistintamente al mismo arco que el par *(B, A)*.

Las figuras que hemos mostrado en las secciones anteriores ilustran algunos conceptos interesantes, que enumeramos [1]:

a.) Un grafo puede constar de varios *subgrafos* no unidos entre sí. La *Figura 1* representa un *solo grafo*, el cual contiene tres partes o subgrafos no unidos por arcos a los otros subgrafos. Cada uno de estos subgrafos separados de un grafo, se suele designar también como *componente conexa*. En ese grafo hay tres *componentes conexas*: la primera está formada por los vértices *A*, *B*, *C*, *D*, y *E*, que están relacionados entre sí de alguna manera. La segunda está formada por los vértices *F* y *G*, que se relacionan entre sí pero no con los otros vértices. Y la última, está conformada solo por el vértice *H*, que no se relaciona con ningún otro.

b.) Si todos los vértices de un grafo están vinculados entre sí formando una sola gran componente conexa, el grafo es un *grafo conexo*. La *Figura 2* y la *Figura 3* representan grafos conexos.

c.) El caso del vértice *H* en la *Figura 1*, muestra claramente que no es obligatorio que un vértice sea punto de partida o de llegada de algún arco. Un vértice puede estar completamente aislado del resto.

d.) En general, dos vértices se dicen *adyacentes* si existe un arco que los une, sin importar, en principio, el sentido de ese arco. En la *Figura 1*, por ejemplo, los vértices *A* y *C* son adyacentes. Observar que el vértice *H* no es adyacente a ningún otro vértice del grafo. Además, todos los arcos que tienen a un nodo *x* como uno de sus extremos (sin importar si ese extremo se toma como de entrada o de llegada), se dicen *arcos incidentes* al nodo *x*.

e.) Se dice que entre dos vértices *A* y *B* hay un *camino de longitud k*, si existe una secuencia de *k* arcos que permitan llegar desde *A* hasta *B*. Es obvio que si dos vértices son adyacentes, entonces existe un camino de longitud 1 (uno) entre ellos. En la *Figura 2*, por ejemplo, existe al menos un camino de longitud 3 entre los vértices *C* y *D*: el camino incluye los arcos *(C, B)*, *(B, E)* y *(E, D)*. Nada impide que haya más de un camino (de la misma longitud o de longitudes diferentes) para unir los mismos dos vértices.

f.) Si existe un camino de cualquier longitud desde un vértice que lleve de retorno al mismo vértice, entonces el grafo tiene un *ciclo*, y se denomina *grafo cíclico*. En caso contrario, el grafo es *acíclico*. En la *Figura 1* hay un ciclo que une los vértices *A*, *D* y *E*. Un arco que parte de un nodo y llega al mismo nodo, suele designarse como un *auto ciclo* o como un *bucle*. En la *Figura 1*, el vértice *E* tiene un *auto ciclo*.

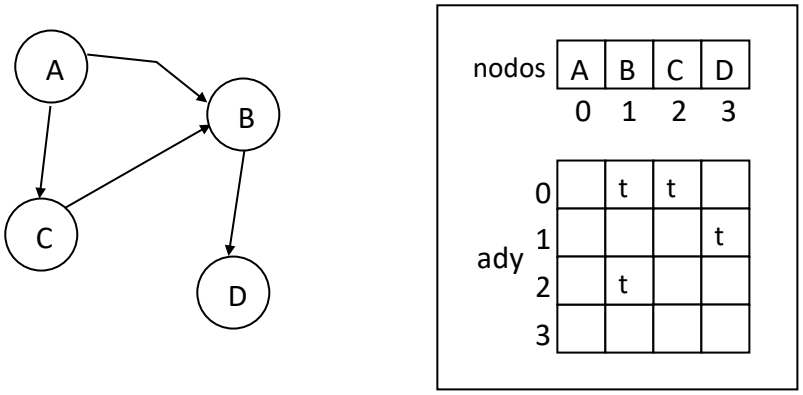
4.] Implementación matricial de grafos.

Existen diversas maneras de implementar un grafo en un lenguaje de programación y la decisión de usar una u otra alternativa dependerá de algunos factores típicos. La forma clásica (y más elemental, que es la aplicaremos en este curso) de representación de un grafo consiste en usar un *arreglo bidimensional* para representar los arcos. Asumiendo que el grafo tendrá un número de vértices no mayor a un cierto número *n* conocido, la información asociada a cada vértice podría almacenarse en un vector de *n* componentes (uno por cada vértice), y las relaciones o arcos podrían estar mantenidas en una matriz *ady*, cuadrada de orden *n*, que *en*

principio podría ser booleana. El uso de esta matriz es la que justifica que la técnica se designe como *implementación matricial* [1].

Si el grafo es *dirigido*, la idea es que las filas de la matriz representarán a los vértices de partida de los arcos del grafo y las columnas representarán a los vértices de llegada. Si existe un arco entre los vértices *i* y *j*, entonces habrá un valor *True* en el componente *ady[i][j]* y si no existe tal arco entonces el valor del componente será *False*. En el siguiente ejemplo mostramos un grafo dirigido simple y su representación matricial (el valor *True* es representado con una *t*, y los casilleros en blanco se suponen valiendo *False*):

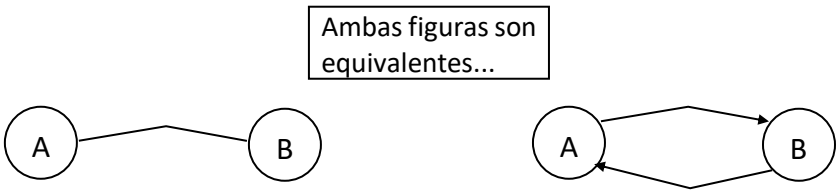
Figura 5: Implementación matricial de un grafo dirigido.



La matriz usada para representar los arcos del grafo se designa como *matriz de adyacencias* del grafo, ya que la misma efectivamente está representando todas las relaciones de adyacencia entre los vértices o nodos del grafo. El vector usado para contener los vértices nos indica también el número de fila o columna que le corresponde a cada nodo en la matriz. Así, el vértice *A* está asignado en el casillero cero del vector, y esto nos dice que en la matriz el nodo *A* será representado por la *fila 0* (cuando se lo tome como vértice de partida de un arco) o por la *columna 0* (cuando se lo tome como nodo de llegada de un arco).

Para implementar un *grafo no dirigido* en forma matricial, debemos hacer una observación simple: cada arco de un grafo no dirigido puede entenderse en realidad como *dos arcos dirigidos*: el que va del primer nodo al segundo y el que vuelve del segundo al primero:

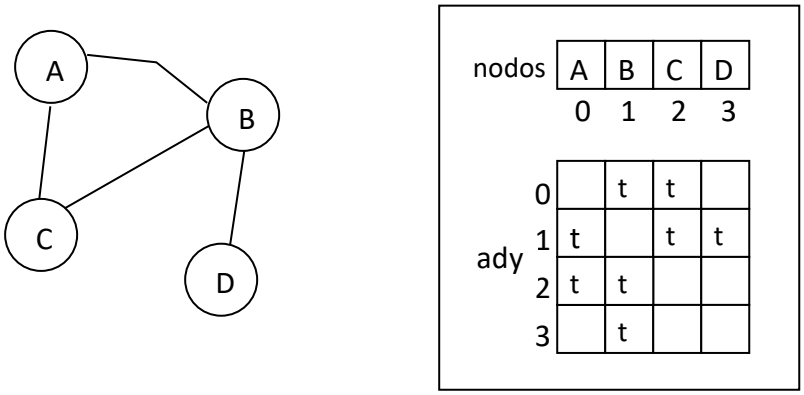
Figura 6: Un arco no dirigido y su equivalencia con dos arcos dirigidos.



En este sentido, un *grafo no dirigido* no es otra cosa que un *grafo dirigido simétrico*: un grafo dirigido en el cual cada vez que existe el arco *(A, B)* existe también el arco *(B, A)* y por lo tanto, la implementación matricial puede hacerse en la misma forma que ya vimos, pero tomando la

precaución de hacer que cada vez que asigne un *True* en el casillero *ady[i][j]*, se asigne también en *ady[j][i]* (ver *Figura 7*).

Figura 7: Implementación matricial de un grafo no dirigido.



En esta ficha de naturaleza introductoria, no mostraremos la forma de implementar un grafo si el grafo es *ponderado* (dirigido o no dirigido). Solo diremos (a modo de idea general) que para su implementación debemos considerar la forma de almacenar el peso o valor de cada arco, y que una idea (muy elemental) es hacer que en la matriz de adyacencias se almacene directamente el peso del arco (si el arco existe) y un cero si el arco no existe. Pero esta idea trae el problema de la *ambigüedad del cero*: el valor cero podría interpretarse como ausencia de arco, o como un arco cuyo peso es cero (si el peso cero fuese admisible). Además, si quisiéramos que un arco tenga asociados varios valores a modo de *pesos múltiples*, esa idea no sería viable.

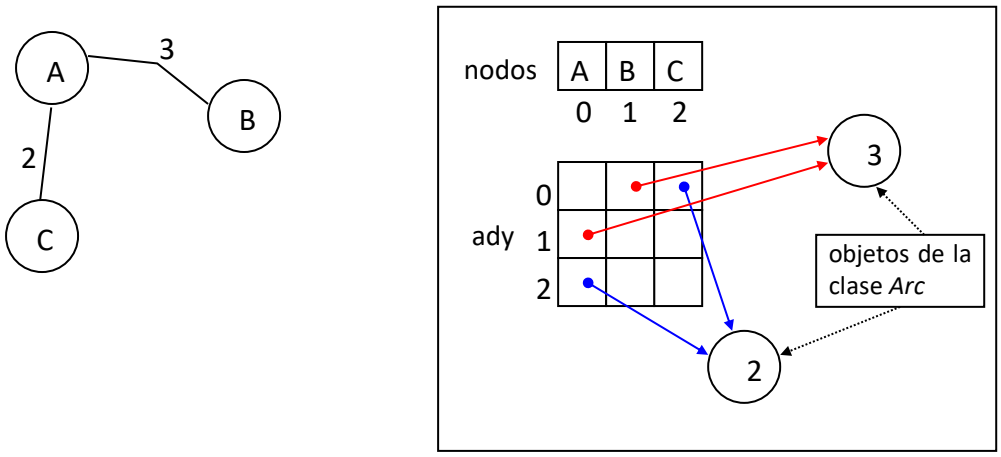
La solución más general para implementar un grafo ponderado (aunque insistimos: solo expondremos la idea) es representar a *los arcos como objetos de una clase Arc*, la cual debe definirse y puede tener tantos atributos como sean necesarios, y almacenar luego en cada casilla de *ady* una referencia a un objeto de esa clase. En principio, si el arco no existe se puede dejar en *None* ese casillero, pero a la larga esa idea también traería problemas a la hora de representar el grafo en forma de cadena (además de otros problemas molestos de control), por lo cual puede ser más útil hacer que cada objeto de la clase *Arc* tenga un atributo booleano para indicar si ese objeto representa un arco válido en el grafo o no. Toda la matriz de adyacencias tendrá referencias no nulas a objetos *Arc*, y cada objeto dirá si el arco representado vale o no.

En el ejemplo de la *Figura 8* en la página siguiente, mostramos la idea para un grafo ponderado y no dirigido sencillo. Suponemos que los casilleros vacíos de la matriz *ady* apuntan ellos también a objetos de la clase *Arc*, aunque no los mostramos. También suponemos que los objetos *Arc* que se muestran, tienen un atributo *boolean* que vale *True* (aunque tampoco lo mostramos aquí). Dejamos para los estudiantes la implementación un grafo ponderado (dirigido o no), en base a estas ideas (no incluimos esa implementación en esta ficha).

La implementación matricial tiene la ventaja de la sencillez, por cuanto el peso de la implementación recae casi totalmente solo en una matriz cuadrada. Pero como contrapartida, el uso de esa matriz hace que se desperdicie espacio en memoria: la matriz usa una cantidad de exactamente n^2 casillas para representar a todo arco posible, existan o no esos arcos, y la forma analítica de la función que mide el tiempo de ejecución de los algoritmos diseñados

para procesar esa matriz completa, también dependerá de n^2 (todos los casilleros serán al menos testeados para ver si representan un arco válido)[1]. Es claro entonces que la representación matricial será aplicable solo si el grafo es denso (intuitivamente: tiene una gran cantidad de arcos) o si por alguna razón se prefiere mantener simplicidad (por ejemplo, por razones didácticas). Además, si el grafo admitiese *arcos paralelos* (es decir: dos o más arcos para unir los mismos dos nodos) la implementación matricial no sería tan práctica ni tan obvia. La alternativa es implementar el grafo mediante *listas multiencadenadas*, lo cual puede hacerse en formas diversas, pero su tratamiento escapa (otra vez…) al alcance introductorio de esta ficha (y de la asignatura).

Figura 8: Implementación matricial de un grafo ponderado no dirigido.



En el modelo *[F30] Grafos* que acompaña a estas notas, mostramos la idea matricial en una implementación general muy básica. El modelo incluye dos módulos *dgraph.py* (con la implementación de la clase *DirectedGraph* para modelar un grafo dirigido en forma matricial), y *ugraph.py* (con la implementación de la clase *UnDirectedGraph* para modelar un grafo no dirigido en forma matricial). En ambas clases se incluye el constructor *__init__()*, el método *__str__()* y varios otros para trabajar en forma elemental con el grafo que se quiera definir. El código fuente completo de la clase *DirectedGraph* es el siguiente:

```
class DirectedGraph:
    def __init__(self, n=10):

        # el vector de nodos, inicialmente con n casillas en None.
        self.nodes = n * [None]

        # la matriz de adyacencias, inicialmente toda en False.
        self.ady = [n * [False] for f in range(n)]

    def __str__(self):
```

[1] En todo caso, buscar la forma analítica de una función que permita predecir el tiempo de ejecución de un algoritmo no es lo mismo que directamente predecir el futuro, como podía hacer el personaje de la película *Next* (del año 2007), dirigida por *Lee Tamahori* y protagonizada por *Nicolas Cage* y *Jessica Biel*. Se trata de la historia de Cris Johnson, un mago de Las Vegas que puede ver lo que ocurrirá en el futuro inmediato, pero solo por un lapso de dos minutos en el futuro. Por causa de esa capacidad, el FBI lo busca y pretende detenerlo para que ayude a predecir y detener ataques terroristas, convirtiendo así la vida del mago en un martirio permanente.

```
        r = "    "
        for nd in self.nodes:
            r += " " + str(nd) + "    "
        r += "\n"

        n = len(self.nodes)
        for i in range(n):
            r += str(self.nodes[i]) + "  "
            for j in range(n):
                c = ""
                if self.ady[i][j] is True:
                    c = "1"
                else:
                    c = "0"
                r += "  " + c + "  "
            r += "\n"

        return r

    def size(self):
        return len(self.nodes)

    def add_arc(self, n1, n2):
        out = False
        i1 = self.search(n1)
        i2 = self.search(n2)
        if i1 != -1 and i2 != -1:
            self.ady[i1][i2] = True
            out = True

        return out

    def remove_arc(self, n1, n2):
        out = False
        i1 = self.search(n1)
        i2 = self.search(n2)
        if i1 != -1 and i2 != -1 and self.ady[i1][i2] is True:
            self.ady[i1][i2] = False
            out = True

        return out

    def is_there_arc(self, n1, n2):
        out = False
        i1 = self.search(n1)
        i2 = self.search(n2)
        if i1 != -1 and i2 != -1:
            out = self.ady[i1][i2]

        return out

    def search(self, x):
        n = len(self.nodes)
        for i in range(n):
        if self.nodes[i] == x:
            return i

        return -1

    def add_nodes(self, ln):
        cn, n = len(ln), len(self.nodes)
```

```
        if self.nodes[0] is not None or cn != n:
            return False

        for i in range(n):
            self.nodes[i] = ln[i]

        return True

    def output_grade(self, x):
        fx = self.search(x)
        if fx == -1:
            return -1

        n = len(self.nodes)
        og = 0
        for c in range(n):
            if self.ady[fx][c] is True:
                og += 1

        return og

    def input_grade(self, x):
        cx = self.search(x)
        if cx == -1:
            return -1

        n = len(self.nodes)
        ig = 0
        for f in range(n):
            if self.ady[f][cx] is True:
                ig += 1

        return ig

    def grade(self, x):
        cx = self.search(x)
        if cx == -1:
            return -1
        g = self.output_grade(x) + self.input_grade(x)

        return g
```

La clase dispone de los dos atributos que hemos indicado: el vector de nodos (designado como *nodes*) y la matriz de adyacencias (*ady*). Se recomienda (aunque no es obligatorio) que el vector de nodos contenga cadenas de caracteres simples para representar el valor o descripción de cada nodo (cadenas como "a", "b", "c" o similares, serán suficientes para el modelo básico que presentamos). La matriz de adyacencias contendrá valores booleanos (como queda dicho: un *True* representará a un arco que efectivamente existe, y un *False* a uno que no existe).

El método constructor *__init__()* toma como parámetro una variable *n* (asumiendo por defecto el valor 10 para ella), que será la cantidad de nodos que se quiere contener en el grafo, y simplemente crea el vector *nodes* con *n* casilleros inicialmente valiendo *None*. Cuando en algún momento se carguen los verdaderos valores de los nodos, esos *None* serán reemplazados por los valores correctos (la carga de los valores finales de cada nodo se hace con otro método de la misma clase designado en este modelo como *add_nodes()*). El método constructor también crea la matriz *ady* de forma que sea cuadrada de *n* filas y *n* columnas. Todos los casilleros de *ady* se inicializan en *False*, para denotar que aun no hay ningún arco en

el grafo a representar. Notemos que el grafo se crea entonces vacío, aunque el espacio para contener *n* nodos y *n*n* arcos está ya reservado, aunque ocupado por valores default (*None* en el vector, y *False* en la matriz).

```
def __init__(self, n=10):
    # el vector de nodos, inicialmente con n casillas en None.
    self.nodes = n * [None]

    # la matriz de adyacencias, inicialmente toda en False.
    self.ady = [n * [False] for f in range(n)]
```

En la práctica, es importante entonces que una vez creado un grafo con este constructor, se proceda a cargar en él los nodos y los arcos que se desea que contenga el grafo para evitar el procesamiento de los valores default citados más arriba. La carga de los nodos se puede hacer en forma simple, masiva y directa con el ya citado método *add_nodes()*:

```
def add_nodes(self, ln):
    cn, n = len(ln), len(self.nodes)
    if self.nodes[0] is not None or cn != n:
        return False

    for i in range(n):
        self.nodes[i] = ln[i]

    return True
```

Este método simplemente toma como parámetro una lista *ln* de valores (repetimos: preferentemente cadenas simples (pero no obligatoriamente)) y agrega todos esos valores al vector de nodos. Si la operación concluyó con éxito, el método retorna *True*. El método no comprueba si los valores contenidos en *ln* son efectivamente cadenas, pero sí comprueba si la cantidad de nodos a agregar coincide con el tamaño del vector de nodos tal como fue creado por el método constructor *__init__()*: si esos tamaños no coinciden, el método aborta y finaliza retornando *False*. Si el vector de nodos ya estaba creado, el método también finaliza sin hacer nada y retorna *False*. Este método es la única forma recomendable de hacer la carga de nodos en este modelo, ya que garantiza que la cantidad de valores que se agregarán al grafo como nodos, coincidirá con el tamaño de creación del grafo, y reemplazará a todos los valores *None* que tenía el grafo en su vector de nodos. La forma de comenzar a usar todo en un programa, puede ser como la siguiente:

```
n = 5
dg = DirectedGraph(n)

v = ["a", "b", "c", "d", "e"]
print("¿Se agregaron nodos al grafo dirigido?:", dg.add_nodes(v))
```

Un método auxiliar incluido en la clase (y que será útil en breve), es el método *search()*:

```
def search(self, x):
    n = len(self.nodes)
    for i in range(n):
        if self.nodes[i] == x:
            return i

    return -1
```

Este método toma una cadena *x* como parámetro y determina por búsqueda secuencial si el vector de nodos contiene alguno cuyo valor coincida con *x* o no. Si se encuentra uno, el método retorna el índice de ese nodo dentro del vector (que será también el índice de ese nodo para entrar en la matriz de adyacencias). Y si no se encuentra el nodo, el método retorna -1.

Una vez incluidos los nodos en el grafo con *add_nodes()*, el paso siguiente es agregar los arcos. La clase *DirectedGraph* dispone del método *add_arc()* para hacerlo:

```
def add_arc(self, n1, n2):
    out = False
    i1 = self.search(n1)
    i2 = self.search(n2)
    if i1 != -1 and i2 != -1:
        self.ady[i1][i2] = True
        out = True

    return out
```

El método une dos nodos cuyos valores son *n1* y *n2* con un arco y retorna *True* en caso de éxito. El método siempre supone que el arco partirá del nodo cuyo valor *n1* y llegará al nodo cuyo valor es *n2*, en caso de poder crear ese arco (recordemos que el grafo representado es dirigido, por lo que el sentido del arco importa). Si alguno de los nodos *n1* o *n2* no existe en el grafo, la operación no se llevará a cabo y retornará *False*.

Notemos que *add_arc()* invoca a *search()* para determinar si *n1* y *n2* existen en el grafo, por lo que si *search()* retorna -1 para cualquiera de ambos, entonces *add_arc()* aborta retornando *False*. Solo si *n1* y *n2* existen, sus índices (retornados por *search()*) se almacenan en las variables *i1* e *i2*. Y con esos índices se entra en la casilla de la matriz *ady[i1][i2]* para marcar con *True* la existencia del arco creado. Como el grafo es dirigido, solo se marca en *True* la posición *[i1][i2]*, pero no la posición *[i2][i1]*.

El método *remove_arc()* hace lo contrario de *add_arc()*: elimina el arco que sale de *n1* y llega a *n2*, y retorna *True* si la operación pudo efectivamente realizarse con éxito:

```
def remove_arc(self, n1, n2):
    out = False
    i1 = self.search(n1)
    i2 = self.search(n2)
    if i1 != -1 and i2 != -1 and self.ady[i1][i2] is True:
        self.ady[i1][i2] = False
        out = True

    return out
```

Como dijimos, elimina el arco entre los nodos *n1* y *n2*. Se supone que el arco a eliminar parte del nodo cuyo valor es *n1* y llega al nodo cuyo valor es *n2*. Si alguno de los nodos *n1* o *n2* no existe en el grafo, la operación no se llevará a cabo y retornará *False*. Igual que antes, el método *search()* se utiliza para comprobar la existencia de *n1* y *n2* y para obtener sus índices de acceso a la matriz. Si el arco existía, simplemente se cambia a *False* el valor del casillero *ady[i1][i2]* y se retorna *True*.

El método *is_there_arc()* solo verifica si existe el arco que parte de *n1* y llega a *n2*. Y el método *size()* (aun más simple) retorna el número de nodos del grafo (que no es otra cosa que el tamaño del vector de nodos del grafo). Dejamos el análisis de ambos para el lector:

```
def is_there_arc(self, n1, n2):
    out = False
    i1 = self.search(n1)
    i2 = self.search(n2)
    if i1 != -1 and i2 != -1:
        out = self.ady[i1][i2]

    return out

def size(self):
    return len(self.nodes)
```

El método *__str__()* retorna una cadena con el contenido del grafo dirigido, en forma matricial, lista para ser visualizada. También dejamos su análisis para el lector:

```
def __str__(self):
    r = "    "
    for nd in self.nodes:
        r += " " + str(nd) + "   "
    r += "\n"

    n = len(self.nodes)
    for i in range(n):
        r += str(self.nodes[i]) + "  "
        for j in range(n):
            c = ""
            if self.ady[i][j] is True:
                c = "1"
            else:
                c = "0"
            r += "  " + c + "  "
        r += "\n"

    return r
```

Finalmente, y solo a modo de aplicación, proponemos tres métodos adicionales sencillos, los cuales toman como parámetro un nodo *x*, y calculan lo que se conoce como el *grado de salida* (o *fuera de grado*) de *x*, el *grado de entrada* (o *entre grado*) de *x*, y el *grado* de *x* [1]:

- *output_grade(x)*: Si *x* existe, calcula y retorna el *fuera de grado* de *x*: la cantidad de arcos que *salen* de *x* (tienen a *x* como nodo de *partida*). Si *x* no existe, retorna -1.
- *input_grade(x)*: Si *x* existe, calcula y retorna el *ente grado* de *x*: la cantidad de arcos que *entran a x* (tienen a *x* como nodo de *llegada*). Si *x* no existe, retorna -1.
- *grade(x)*: Si *x* existe, calcula y retorna el *grado* de *x*: la cantidad de arcos que *inciden en x* (tienen a *x* como nodo de *partida o como nodo de llegada*). Si *x* no existe, retorna -1.

La implementación de esos tres métodos es la siguiente:

```
def output_grade(self, x):
    fx = self.search(x)
    if fx == -1:
```

```
            return -1

        n = len(self.nodes)
        og = 0
        for c in range(n):
            if self.ady[fx][c] is True:
                og += 1

        return og

    def input_grade(self, x):
        cx = self.search(x)
        if cx == -1:
            return -1

        n = len(self.nodes)
        ig = 0
        for f in range(n):
            if self.ady[f][cx] is True:
                ig += 1

        return ig

    def grade(self, x):
        cx = self.search(x)
        if cx == -1:
            return -1

        g = self.output_grade(x) + self.input_grade(x)

        return g
```

Está claro que el *fuera de grado* de *x* consiste simplemente en contar la cantidad de casillas *True* que hay en la *fila de x* en la matriz *ady*. El *entre grado* de *x* es lo mismo, pero en la columna de *x* en la matriz. Y el grado de *x* es solo *la suma de su fuera de grado con su entre grado*. Eso hacen esos tres métodos, y dejamos los detalles para ser analizados por el lector.

El siguiente programa simple muestra una prueba general de la funcionalidad de esta clase (es el programa *test01.py* en el proyecto que acompaña a esta ficha):

```
from dgraph import *


def main():
    n = 5
    dg = DirectedGraph(n)

    v = ["a", "b", "c", "d", "e"]
    print("¿Se agregaron nodos al grafo dirigido?:", dg.add_nodes(v))

    dg.add_arc("a", "b")
    dg.add_arc("a", "c")
    dg.add_arc("b", "d")
    dg.add_arc("d", "a")
    dg.add_arc("e", "b")

    print("El grafo es:")
    print(dg)
    print("Cantidad de nodos del grafo:", dg.size())
```

```
    print()
    print("Cantidad de arcos que salen de 'a':", dg.output_grade("a"))
    print("Cantidad de arcos que llegan a 'a':", dg.input_grade("a"))
    print("Cantidad de arcos incidentes a 'a':", dg.grade("a"))

    print()
    print("Hay arco entre 'a' y 'c'?:", dg.is_there_arc("a", "c"))
    dg.remove_arc("a", "c")
    print("Hay arco (después de borrarlo)?:", dg.is_there_arc("a", "c"))


if __name__ == '__main__':
    main()
```

La clase *UnDirectedGraph* incluida en el módulo *ugraph.py* del proyecto que acompaña a esta ficha, implementa la idea de *grafo no dirigido* y su contenido es casi completamente igual al de la clase *DirectedGraph*. De hecho, las únicas diferencias concretas son los métodos *add_arc()* y *remove_arc()*, y el hecho de que ahora esta clase no cuenta con los métodos *output_grade()* e *input_grade()* (pero sí cuenta con el método *grade()*):

```
class UnDirectedGraph:
    def __init__(self, n=10):
        # el vector de nodos, inicialmente con n casillas en None.
        self.nodes = n * [None]

        # la matriz de adyacencias, inicialmente toda en False.
        self.ady = [n * [False] for f in range(n)]

    def __str__(self):
        r = "    "
        for nd in self.nodes:
            r += " " + str(nd) + "   "
        r += "\n"

        n = len(self.nodes)
        for i in range(n):
            r += str(self.nodes[i]) + "  "
            for j in range(n):
                c = ""
                if self.ady[i][j] is True:
                    c = "1"
                else:
                    c = "0"
                r += "  " + c + "  "
            r += "\n"

        return r

    def size(self):
        return len(self.nodes)

    def add_arc(self, n1, n2):
        out = False
        i1 = self.search(n1)
        i2 = self.search(n2)
        if i1 != -1 and i2 != -1:
            self.ady[i1][i2] = self.ady[i2][i1] = True
            out = True

        return out
```

```
def remove_arc(self, n1, n2):
    out = False
    i1 = self.search(n1)
    i2 = self.search(n2)
    if i1 != -1 and i2 != -1 and self.ady[i1][i2] is True:
        self.ady[i1][i2] = self.ady[i2][i1] = False
        out = True

    return out

def is_there_arc(self, n1, n2):
    out = False
    i1 = self.search(n1)
    i2 = self.search(n2)
    if i1 != -1 and i2 != -1:
        out = self.ady[i1][i2]

    return out

def search(self, x):
    n = len(self.nodes)
    for i in range(n):
        if self.nodes[i] == x:
            return i

    return -1

def add_nodes(self, ln):
    cn, n = len(ln), len(self.nodes)
    if self.nodes[0] is not None or cn != n:
        return False

    for i in range(n):
        self.nodes[i] = ln[i]

    return True

def grade(self, x):
    f = self.search(x)
    if f == -1:
        return -1

    n = len(self.nodes)
    g = 0
    for c in range(n):
        if self.ady[f][c] is True:
            g += 1

    return g
```

El método *add_arc()* debe hacer lo mismo que ya hacía el método equivalente en la clase *DirectedGraph*, pero como ahora el grafo es *no dirigido*, debemos recordar que si es *True* el casillero *[i1][i2]* entonces también debe ser *True* el casillero simétrico *[i2][i1]*. Y eso es todo:

```
def add_arc(self, n1, n2):
    out = False
    i1 = self.search(n1)
    i2 = self.search(n2)
```

```
        if i1 != -1 and i2 != -1:
            self.ady[i1][i2] = self.ady[i2][i1] = True
            out = True

        return out
```

Lo mismo vale para el método *remove_arc()*: debe poner en *False* los dos casilleros simétricos que correspondan al arco que se debe eliminar:

```
    def remove_arc(self, n1, n2):
        out = False
        i1 = self.search(n1)
        i2 = self.search(n2)
        if i1 != -1 and i2 != -1 and self.ady[i1][i2] is True:
            self.ady[i1][i2] = self.ady[i2][i1] = False
            out = True

        return out
```

Como el grafo es no dirigido, no hay forma de distinguir entre el fuera de grado y el entre grado: todos los arcos incidentes a un nodo son de entrada, y todos son también de salida. Por lo tanto, esas dos medidas son simplemente equivalentes al grado del nodo. Por esta razón ya no tenemos los métodos *input_grade()* y *output_grade()*. Solo dejamos al método *grade()*, que lo único que tiene que hacer es recorrer la fila del nodo (o la columna de este, pero no ambas) y llevar la cuenta de la cantidad de casillas valiendo *True* que aparezcan:

```
    def grade(self, x):
        f = self.search(x)
        if f == -1:
            return -1

        n = len(self.nodes)
        g = 0
        for c in range(n):
            if self.ady[f][c] is True:
                g += 1

        return g
```

El resto de los métodos de la clase son iguales a los de la clase *DirectedGraph*. Mostramos a continuación un breve programa de prueba (es el programa *test02.py* en el proyecto que acompaña a esta ficha):

```
from ugraph import *


def main():
    n = 5
    ug = UnDirectedGraph(n)

    v = ["a", "b", "c", "d", "e"]
    print("¿Se agregaron nodos al grafo no dirigido?:", ug.add_nodes(v))

    ug.add_arc("a", "b")
    ug.add_arc("a", "c")
    ug.add_arc("b", "d")
    ug.add_arc("d", "a")
    ug.add_arc("e", "b")
```

```
    print("El grafo es:")
    print(ug)
    print("Cantidad de nodos del grafo:", ug.size())

    print()
    print("Cantidad de arcos incidentes a 'a':", ug.grade("a"))

    print()
    print("Hay arco entre 'a' y 'c'?:", ug.is_there_arc("a", "c"))
    ug.remove_arc("a", "c")
    print("Hay arco (después de borrarlo)?:", ug.is_there_arc("a", "c"))


if __name__ == '__main__':
    main()
```

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.