# AA2021a-07-Estructuras de datos fundamentales 3

Abdiel E. Cáceres González

14 de febrero de 2021

## Índice



## Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.

2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

# PARTE TRES
# Estructuras de datos

## 5. Grafos

Esta leción presenta métodos para representar un grafo y para hacer búsquedas en un grafo. Buscar en un grafo significa seguir aristas sistemáticamente visitando los vértices del grafo hasta encontrar la información que se esté buscando.

Muchas situaciones del mundo real se pueden representar o describir mediante un diagrama que consiste de un conjunto de puntos donde algunos de ellos están unidos por líneas.

Por ejemplo, si los puntos representan personas, un punto unido por una línea con otro punto podría representar que esas personas [representadas por los puntos] están relacionadas por la amistad.

Otra situación muy común es que esos puntos representen lugares, y las líneas representan calles de doble sentido, y así el unir dos puntos significaría que hay una calle que permite llegar de un lugar al otro y de regreso. A veces las calles solo van en un solo sentido y esto también tiene su propia representación.

Situaciones como las anteriores han permitido el origen del concepto de grafo, el cual definiremos enseguida.

### 5.1. Definición

Un grafo $G$ es una estructura de datos $[\![V, A]\!]$ conformada por un conjunto de vértices $V$ y un conjunto de aristas $A$. Aunque el conjunto $V$ puede contener cualquier tipo de dato, en esta lección utilizaremos los números naturales $1, 2, \ldots, n$ para nombrar los $n$ vértices del grafo. Esta licencia no representa una limitante, pues el número puede representar un índice en una tabla, que contenga toda la información pertninente del vértice.

Las aristas son relaciones entre pares de vertices, por lo que $A \subset V \times V$. Entonces un grafo $G$ se representa como $G = [\![V, A]\!]$, donde $n = |V|$ y $m = |A|$.

#### 5.1.1. Conceptos básicos

La **incidencia** es un término que se aplica tanto a los vértices cmo a las aristas. Los vértices $u$ y $v$ son **incidentes** con la arista $\langle u, v\rangle$; la arista $\langle u, v\rangle$ es incidente a los vértices $u$ y $v$.

La **adyacencia** también se refiere tanto a vértices como a las aristas. Un vértices $v$ es adyacente a un vértice $u$ si hay una arista $\langle u, v\rangle$, es decir hay una arista que inicia en $u$ y termina en $v$. Por su parte, dos aristas son adyacentes si comparten un vértice en común.

El número de aristas incidentes con un vértice $v$ [la cantidad de aristas que involucran de algún modo al vértice $v$] se llama **grado** del vértice $v$. El grado de $v$ lo podemos denotar por $\texttt{gr}\,(v)$. Si $\texttt{gr}\,(v)$ es un número par, entonces decimos que $v$ es un **vértice par**; si $\texttt{gr}\,(v)$ es impar, entonces $v$ es un **vértice impar**. Decimos que un vértice $v$ está determinado por el conjunto **aislado** si tiene grado 0.

Los grafos se llaman así porque se pueden representar gráficamente y es precisamente por su representación gráfica que en ocasiones es posible entender muchas de sus propiedades. En ocasiones no es posible tener una representación gráfica porque resulta muy confuso debido al gran número de aristas.

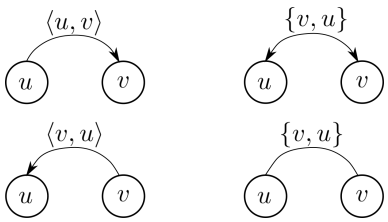
La representación gráfica más usual consiste en círculos etiquetados con flechas que unen pares de éstos. Las gráficas cambian un poco en dependencia de la naturaleza del grafo, ya sea un grafo dirigido o un grafo no-digirido.

Los vértices se representan mediante pequeños círculos etiquetados con la información que contienen, el nombre del vértice generalmente coincide con la información que se muestra.

Las aristas pueden representarse en forma distinta en los grafos dirigidos y en lo no-dirigidos. En los grafos dirigidos, si $u$ y $v$ son vértices, cada vértice se dibuja por un círculo

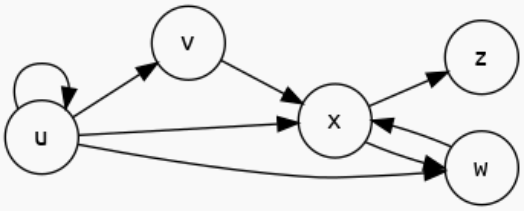
etiquetado; la arista $\langle u, v\rangle$ se dibuja como una flecha que sale del vértice $u$ y apunta al vértice $v$. La arista $\langle v, u\rangle$ es una flecha con origen en $v$ ya que es el vértice inicial, y termina en el vértice $u$ que es el vértice final.

En el caso de los grafos no-dirigidos y debido a que el orden en que aparecen los vértices no es importante, se representan como un conjunto $\{u, v\}$ o bien $\{v, u\}$. Gráficamente pueden ser representados por una línea con doble flecha, una que apunta a $u$ y la otra apuntando a $v$. Frecuentemente se omiten las puntas de flecha, indicando que no hay dirección preestablecida en la arista.




**Ejemplo 1**

Sea $G = \langle V, A\rangle$ un grafo dirigido, donde $V = \{u, v, w, x, z\}$ es el conjunto de vértices y $A = \{\langle u, v\rangle, \langle u, u\rangle, \langle u, w\rangle, \langle w, x\rangle, \langle v, x\rangle, \langle x, w\rangle, \langle u, x\rangle, \langle x, z\rangle\}$ el conjunto de aristas.




No hay diferencia alguna en la distribución espacial de los vértices y aristas, por lo que la ubicación de los vértices y aristas no es reelevante.

Es posible saber si un grafo es dirigido o es no-dirigido atendiendo al conjunto de aristas.

Dado un grafo $G = [\![V; A]\!]$ la proposición «$G$ es no-dirigido» se denota por $\texttt{gnodir}\,(G)$ de acuerdo a la siguiente definición:

$$\texttt{gnodir}\,(G) \leftrightharpoons \forall a \in A : a^{-1} \in A$$

Un grafo que no cumple la condición anterior es un grafo dirigido, esto es que además de las aristas de la forma $\langle u, v\rangle$, también las aristas inversas $\langle v, u\rangle$ pertenecen al conjunto de aristas, por lo que no es necesario dibujar ambas flechas, sino solamente un arco de línea que une los vértices $u$ y $v$ [sin puntas de flecha en los extremos]. El archo de línea puede ser analizado en ambas direcciones.

En un grafo dirigido o digrafo, se tienen que dibujar las flechas que unen los vértices. Aquí el sentido de la flecha es importante, no se puede acceder a un vértice si no hay arista con flecha que le apunte.

### 5.1.2. `GraphViz` para dibujar grafos

`GraphViz` es una herramienta par visualizar grafos que es muy útil a la hora de estudiar grafos, pues tener una idea visual del grafo ayuda a intuir algunas propiedades del grafo, como su densidad [relación entre vértices y aristas], el grado de los nodos, entre otras cosas.

`GraphViz` puede ser descargado desde su sitio en Internet https://graphviz.org/, y puede ser instalado sobre diferentes plataformas de cómputo. Adicionalmente, se han creado algunas bibliotecas especiales para diferentes lenguajes de programación, entre ellos `Python`.

En esta parte de la lección veremos cómo crear y visualizar imagenes de grafos, utilizando el paquete [biblioteca de funciones] `graphviz`. Este paquete facilita la creación y despliegue de grafos, descritos en el lenguaje `DOT` del software `Graphviz` desde `Python`.

En general se puede crear un objeto grafo, ensamblar el grafo agregando vértices y aristas, recuperar el código fuente `DOT` para grabarlo en un archivo y visualizar el mismo grafo.

### Instalación de `GraphViz`

Esta parte se divide en dos: Instalar el motor de `GraphViz` en la computadora y luego instalar en paquete `graphviz` que se utiliza desde `Python` para crear los grafos.

Ya que el motor de `GraphViz` se puede instalar en diferentes plataformas, en esta sección solamente daré algunas pautas generales para hacerlo.

1. Descarga el instalador. https://graphviz.org/download/
2. Instala el programa, que es una herramienta del sistema, por lo que no tendrás una ventana que poder abrir y comprobar que ya se instaló.
   - `Linux-Debian` `$ sudo apt install graphviz`
   - `Linux-Fedora` `$ sudo yum install graphviz`
   - Para `Windows` hay muchas opciones, aquí la que he utilizado antes. Hay que descargar el siguiente Instalador para windows de 64 bits: https://gitlab.com/graphviz/graphviz/-/package_files/6164164/download Para instalarlo asegurate de hacerlo como administrador, para otorgar los permisos necesarios, además de agregar la ruta del programa `dot.exe` a la variable `PATH` del sistema.
   - Para `Mac` También hay varias opciones, aquí la que he utilizado usando `Homebrew`: `$ brew install graphviz`

   Para saber que el programa ya está instalado debes abrir una terminal de comandos y ejecutar `$ dot -V` [con V en mayúscula]. Hecho esto debe aparecer la versión instalada. Si no se ha instalado, aparece un mensaje que dice que no se reconoce ese comando.

Ya instalado `GraphViz`, puede ser utilizado aún sin `Python` o algún otro lenguaje de programación. En http://zhangliyong.github.io/posts/2014/02/23/graphviz-tutorial.html hay un breve tutorial y en Internet se han publicado muchos otros. Lo que respecta al curso es instalar el paquete `graphviz` para utilizarlo desde `Python`. Puedes utilizar cualquiera de las siguientes opciones, dependiendo de tu instalación.

1. `pip install graphviz` Para utilizar esta opción, debes asegurarte de tener instalado `pip` [*`Python` install packeges*].
2. `conda install -c anaconda graphviz` Si instalaste `Python` pot medio de anaconda, puedes utilizar esta opción.

### Uso de `graphviz` en `Python`

Crear grafos utilizando el paquete `graphviz` en `Python` es bastante simple:

```
In []: from graphviz import Digraph
In []: G = Digraph(comment='Digrafo simple')
In []: print(G)
// Digrafo simple
digraph {
}
In []:
```

En el ejemplo anterior se han hecho tres cosas:

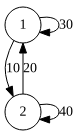
1. Importar la clase `Digraph` del paquete `graphviz`. Esta clase sirve para crear grafos dirigidos, más frecuente y comunmente cnocidos como **digrafos**.

2. Crea un objeto de la clase `Digraph`, lo asigna al identificador `G`.

3. Al imprimir `G` [el Digrafo], se imprime el código fuente asociado al digrafo.

Ahora podemos agregar nodos y aristas:

```
In []: G.node('1', shape='circle')
In []: G.node('2', shape='circle')
In []: G.edge('1','2', label='10')
In []: G.edge('2','1', label='20')
In []: G.edge('1','1', label='30')
In []: G.edge('2','2', label='40')
In []: G
Out []:
```




```
In []:
```

Ahora veamos el código fuente generado:

```
In []: print(G)
// Digrafo simple
digraph {
  1
  2
  1 [shape=circle]
  2 [shape=circle]
  1 -> 2 [label=10]
  2 -> 1 [label=20]
  1 -> 1 [label=30]
  2 -> 2 [label=40]
}
In []:
```

Puedes consultar la página https://graphviz.readthedocs.io/en/stable/manual.html# para tener una guía más completa de cómo utilizar este paquete.

### 5.1.3. Estructura para vértices

Un vértice es un objeto que representa toda la información encapsulada que debe ser considerada como elemento en la relación. Por ejemplo, si un vértice $v$ representa una ciudad, el objeto $v$ claramente de una clase `Vertice`, puede contener información «satélite» como número de habitantes, etc. Hay sin embargo, información que es útil para manejar vértices dentro de un grafo, de modo que podemos contruir una clase `Vertice` diseñada con la estructura genérica $[\![\texttt{Atributos} \| \texttt{Metodos}]\!]$. Para el caso de los vértices podemos crear una clase:

$$\texttt{Vertice} \leftarrowtail [\![\underline{\texttt{k}} = \texttt{NIL}, \texttt{ady} = \texttt{[]}, \texttt{visto} = \texttt{False}, \ldots \| \ldots]\!]$$

Donde:

`k` es el valor clave, que puede ser utilizado como valor constructor. Para fines de esta lección, siempre serán números naturales, empezando desde 1 en adelante.

`ady` contiene la lista de adyacencia del vértice, es decir aquellos vértices que son accesibles desde «este» vértice por medio de un solo paso.

`visto` es un valor booleano, que dice si el vértice ha sido visto o no, esta información es útil cuando se quiere revisar el contenido de todos los vértices sin repetir alguno.

Para esta clase [por ahora] no requiere métodos, pero claramente se requieren métodos para extraer/cambiar la información contenida en ellos.

### 5.1.4. Estructura para aristas

Las aristas están definidas escencialmente un par de vértices, teóricamente si un vértice $v$ está relacionado con otro vértice $u$, decimos que $\langle u, v\rangle$ es una arista en $G$. Para modelar una arista, es buena idea crear una clase `Arista`, puesto que las aristas tienen, además de los vértices que la componen algunos otros atributos como la importancia de la arista, que puede ser interpretada por ejemplo como un camino más transitado o el costo de transitar por ella.

$$\texttt{Arista} \leftharpoondown [\![\underline{\texttt{vi}} = \texttt{NIL}, \underline{\texttt{vf}} = \texttt{NIL}, \texttt{ew} = \texttt{NIL}, \ldots \| \ldots]\!]$$

Donde:

`vi` es el vértice inicial, si $\langle u, v\rangle$ es la arista, este atributo se refiere al vértice $u$. Es una instancia de la clase `Vertice`.

`vf` es el vértice final, si $\langle u, v\rangle$ es la arista, este atributo se refiere al vértice $v$. Es una instancia de la clase `Vertice`.

`ew` es el peso de la arista. Generalmente es un número real, pero puede ser un valor de cualquier sistema métrico.

### 5.1.5. Estructura para grafos

Los grafos, que son el objeto de estudio de esta lección, estarán modelados mediante objetos de una clase `Grafo` que principalmente debe contener una lista de vertices y una lista de aristas, no es necesario establecer la función de peso para un grafo ponderado, ya que el peso de las aristas es asignado en la clase `Arista`. De modo que la estructura de un grafo puede ser:

$$\texttt{Grafo} \leftharpoondown [\![\underline{\texttt{V}} = \texttt{[]}, \underline{\texttt{E}} = \texttt{[]}, \underline{\texttt{dir}} = \texttt{True}, \ldots \| \texttt{adj}\,(k), \ldots]\!]$$

Donde los atributos son:

`V` es un conjunto [modelado por medio de una lista] de objetos de la clase `Vertice`. El valor por defecto de este atributo puede ser una lista vaía `[]`.

`A` es un conjunto [modelado por medio de una lista] de objetos de la clase `Arista`. El valor por defecto de este atributo puede ser una lista vaía `[]`.

`dir` es un valor booleano que indica si el grafo es un grafo dirigido o no lo es. Por defecto es `True`, lo que significa que se trata de un digrafo.

Los métodos serán agregados a medida que la lección avance, pero por poner un ejemplo, uno de los métodos del grafo puede ser calcular los vértices adyacentes a un vértice dado por su clave.

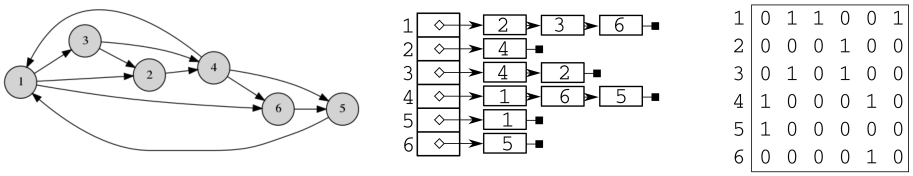
> `adj` $(k)$ : `lista` Es un método público que recibe como entrada un valor $k$, que es la clave de un vértice [de la clase `Vertice`]. El método debe devolver una lista con la clave de los vértices que son adyacentes.

## 5.2. Representación de grafos

Podemos elegir entre dos maneras estándares para representar un grafo $G = [\![V, A]\!]$: como una colección de listas de adyacencia o como una matriz de adyacencia. Cada una de ellas se aplica tanto a grafos dirigidos como a grados no dirigidos. Se suele elegir la representación de listas de adyacencia porque proporciona una manera compacta de representar grafos **dispersos** –aquellos en donde $|A|$ es mucho menor que $|V|^2$–. Muchos de los algoritmos de grafos utilizan la lista de adyacencias como dato de entrada para crear grafos. Podemos preferir para representación de matriz de adyacencia cuando el grafo es **denso** –$|A|$ es cercano a $|V|^2$– o cuando necesitamos ser capaces de decir rápidamente si existe una arista que conecte dos vértices.

### 5.2.1. Listas de adyacencia

La representación como **listas de adyacencia** de un grafo $G = [\![V, A]\!]$ consiste de un arreglo $Adj$ de $|V|$ listas, una por cada vértice en $V$. Para cada $u \in V$, la lista de adyacencia $Adj\langle u\rangle$ contiene todos los vértices $v$ tales que hay una arista $\langle u, v\rangle \in A$. Esto es, $Adj\langle u\rangle$ consiste de todos los vértices adyacentes a $u$ en $G$. [De manera alternativa, pueden contener apuntadores a esos vértices]. Como las listas de adyacencia representan aristas de un grafo, en el psudocódigo las tratamos el arreglo $Adj$ como un atributo del grafo, igual como tratamos al conjunto de aristas $A$. Además, en el pseudocódigo veremos alguna notación como $G.Adj\langle u\rangle$, que representa la lista de adyacencia del grafo $G$ que encabeza el vértice `u`.

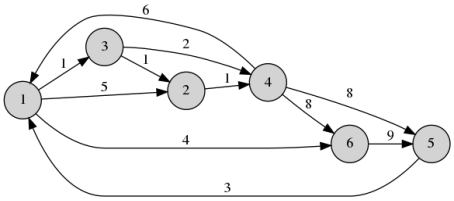



(a) Un grafo $G$ dirigido con 6 vértices y 11 aristas. (b) Las listas de adyacencia del grafo $G$. (c) La matriz de adyacencia del grafo $G$.

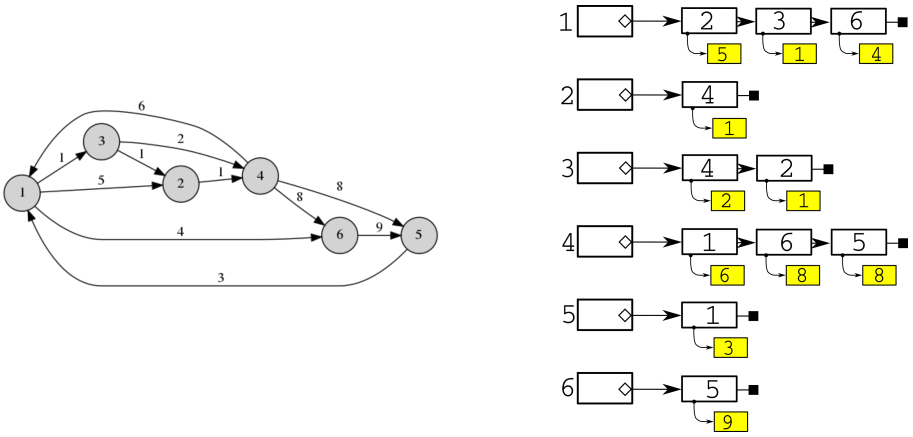
Si $G$ es un grafo dirigido, la suma de las longitudes de todas las listas de adyacencia es $|A|$, es así porque una arista de la forma $\langle u, v\rangle$ está representada por un nodo con clave $v$ en la lista que encabeza $u$, así que $v$ aparece una vez en $Adj\langle u\rangle$.

Si $G$ es un grafo no dirigido, la suma de las longitudes de todas las listas de adyacencia es $2|A|$, porque si $\langle u, v\rangle \in A$, $u$ aparece en la lista de adyacencia de $v$ y también aparece en la lista de adyacencia de $v$.

No importa si el grafo es dirigido o no dirigido, se tiene la propiedad deseable que la cantidad de memoria requerida es $\boldsymbol{\Theta}(|V| + |A|)$



Podemos adaptar la lista de adyacencia para representar **grafos ponderados**, es decir, grafos cuyas aristas tienen un peso asociado, típicamente dado por una función $\mathtt{w} : A \to \mathbb{R}$. Por ejemplo, digamos que $G = [\![V, A, \mathtt{w}]\!]$ es un grafo ponderado con función de peso `w`. Simplemente almacenamos el peso $\mathtt{w}(\langle u, v\rangle)$ de la arista $\langle u, v\rangle \in A$ con el vértice $v$ en la lista de adyacencia de $u$. Otra opción es asociar el peso como un atributo de la arista. La representación como lista de adyacencia es robusta en el sentido de que podemos modificarla para soportar muchas otras variantes de los grafos.



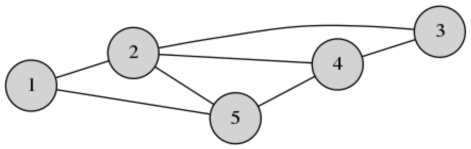
```
1 2 5 3 1 6 4
2 4 1
3 4 2 2 1
4 1 6 6 8 5 8
5 1 3
6 5 9
```

Una desventaja potencial en la representación de listas de adyacenci es que no hay una manera rápida de determinar si una arista dada $\langle u, v\rangle$ está presente en el grafo, la única es buscar $v$ en la lista de adyacencia $Ady\langle u\rangle$. Si el grafo se representa como una matriz de adyacencia, esta desventaja queda superada, pero el costo es el uso de más memora.

### 5.2.2. Matriz de adyacencia

Para la representación de una **matriz de adyacencia** de un grafo $G = [\![V, A]\!]$ suponemos que los vértices están numerados $1, 2, \ldots, |V|$ de alguna manera arbitraria. Entonces la representación de matriz de adyacencia del grafo $G$ consiste de una matriz $MA = (a_{ij})$ de orden $|V| \times |V|$ tal que

$$a_{ij} = \begin{cases} 1 & \text{Si } \langle i, j\rangle \in A. \\ \\ 0 & \text{e.o.c} \end{cases}$$




| | 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 |

La matriz de adyacencia de un grafo requiere un espacio en memoria del orden $\boldsymbol{\Theta}\left(|V|^2\right)$, independientemente del número de aristas en el grafo [que a lo más pueden ser $|V|^2$, considerando que el grafo no tiene aristas paralelas –aristas diferentes que tienen mismo origen y destino–].

Observa la simetría sobre la diagonal principal en la matriz de adyacencia de un grafo no dirigido. Como el grafo es no dirigido, las aristas $\langle u, v\rangle$ y $\langle v, u\rangle$ se refieren a la misma arista, la transpuesta de la matriz de adyacencia $MA$ de un grafo no dirigido es ella misma: $MA = MA^T$. En algunas aplicaciones, se amlacena solamente la información sobre la diagonal principal, de modo que se ahorra casi la mitad de espacio.

Como con la representación de las listas de adyacencia de un grafo, una matriz de adyacencia puede representar también un grafo ponderado. Por ejemplo, si $G = [\![V, A, \mathtt{w}]\!]$ es un grafo ponderado con una función de peso $\mathtt{w}$ en las aristas, podemos simplemente almacenar el peso $\mathtt{w}(u, v)$ de la arista $\langle u, v\rangle \in A$ como la entrada de la matriz en la fila $u$ y columna $v$ de la matriz de adyacencia. Si una arista no existe, podemos almacenar la marca `NIL` como entrada correspondiente en la matriz, aunque en muchos problemas es conveniente utilizar alguno de los valores 0, $\infty$ o $-\infty$.

Aunque la representación de listas de adyacencia es asintóticamente más eficiente en espacio que la representación de matriz de adyacencia, las matrices de adyacencia son más simples, por eso podemos preferiras cuando los grafos son razonablemente pequeñas. Aún mas, las matrices de adyacencia para grafos no ponderados tienen una ventaja mas: requieren solamente un bit por entrada.

### 5.2.3. Representación de los atributos

La mayoría de los algoritmos que operan sobre grafon necesitan mantener atributos para los vértices y/o aristas. Indicamos esos atributos utilizando nuestra notación habitual, como $v.d$ para un atributo $d$ del vértice $v$. Cuando indicamos las aristas como pares de vértices, utilizamos el mismo estilo de notación. Por ejemplo, si las aristas tienen un atributo $f$, denotamos este atributo para la arista $\langle u, v\rangle$ por $\langle u, v\rangle.f$.

Implementar atributos en vértices y aristas en programas reales, puede ser otra historia por completo. No hay una mejor manera de almacenar y acceder a los atributos de vértices y aristas. Para una situación particular, tu decisión podría depender del lenguaje de programación que se utilice y cómo el resto de tu programa utilice el grafo. Si se representa

el grafo usando listas de adyacencia, un diseño puede representar los atributo de manera adicional, como con un arreglo $d\langle 1 \ldots |V| \rangle$ que es paralela al arreglo $Adj$. Si los vértices adyacentes a $u$ están en $Adj\langle u \rangle$, podemos hacer algo similar con los atributos de $u$, llamando a los elementos almacenados en $d\langle u \rangle$. Por ejemplo, en un lenguaje orientado a objetos, los atributos de los vértices se pueden representar como variables de la instancia en una subclase de una clase `Vertice`.

## Ejercicios

### Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

- 3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.
- 2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.
- 1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.
- 0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Escribe un programa en `Python` que se llame `Grafos.py`. Este programa te servirá para hacer muchas tareas, por lo que es muy importante que siguas las instrucciones.

   *a*) Crea una clase llamada `Vertice` con las especificaciones de la página 5

   *b*) Crea una clase llamada `Arista` con las especificaciones de la página 6

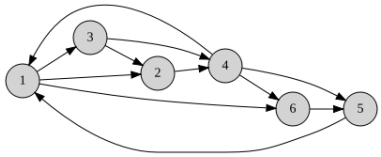
   *c*) Crea una clase llamada `Grafo` con las especificaciones de la página 6 [en otro ejercicio se modifica un poco esta clase].

2. Crea un método para la clase `Grafo` que se llame `inf`. No requiere parámetros de entrada. La salida es la impresión de la lista de claves de los vértices, junto con la lista de las aristas.

```
In []: v1 = Vertice(1)
In []: v2 = Vertice(2)
In []: v3 = Vertice(3)
In []: v4 = Vertice(4)
In []: v5 = Vertice(5)
In []: v6 = Vertice(6)
In []: e1 = Arista(v1,v2)
In []: e2 = Arista(v3,v4)
In []: e3 = Arista(v1,v3)
In []: e4 = Arista(v3,v2)
In []: e5 = Arista(v4,v1)
In []: e6 = Arista(v5,v1)
In []: e7 = Arista(v4,v6)
In []: e8 = Arista(v6,v5)
In []: e9 = Arista(v1,v6)
In []: e10 = Arista(v2,v4)
In []: e11 = Arista(v4,v5)
In []: G = Grafo([v1,v2,v3,v4,v5,v6], [e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11], True)
In []: G.inf()
V=[1,2,3,4,5,6]  E={[1,2],[3,4],[1,3],[3,2],[4,1],[5,1],[4,6],[6,5],[1,6],[2,4],[4,5]}
In []:
```

3. Agrega el paquete `graphviz` al programa. Crea un método para la clase `Grafo` que se llame `show`. Este método no requiere valores de entrada. El proósito es generar una imagen con formato `PNG` con la representación gráfica del grafo. Utiliza la clase `Graph` o `Digraph` para crear la imagen de un grafo en formato `PNG`. Si el grafo es dirigido, entonces utiliza `Digraph`. Si el grafo es no dirigido debes utilizar `Graph` para generar el grafo. Agrega el atributo `nombre` a la clase `Grafo`; este atributo te servirá para darle nombre los archivos `.gv` [el código fuente del grafo] y `.png` [la propia imagen]. Los vértices deben tener la forma de un círculo [por defecto tienen la forma de una elipse]. Si el grafo es ponderado, las aristas deben mostrar la etiqueta del valor de esa arista.

```
In []: G.show()
Out[]:
```




```
In []:
```

4. Agrega los atributos privados `__Adj` y `__MA`. El atributo privado `__Adj` debe tener una lista con las listas de adyacencia de los vértices. El atributo provado `__MA` debe tener una matriz cuadrada [lista de listas] con la matriz de adyacencias del grafo. Por ahora considera que el grafo es no-ponderado, por lo que la matriz de adyacencias debe tener valores 0 o 1. *Consejo:* Crea métodos prvados para calcular las listas de adyacencia y matriz de aprendizado. Estos métodos no se volverán a invocar, pues la idea es tener un acceso de orden $\boldsymbol{\Theta}(1)$.

5. Agrega los métodos públicos `Adj` y `MA`. El primero debe devolver la lista de listas de adyacencias del grafo; el segundo debe devolver la matriz de adyacencia del grafo.

```
In []: G.Adj()
Out[]: [[1, 2, 3, 6], [2, 4], [3, 4, 2], [4, 1, 6, 5], [5, 1], [6, 5]]

In []: G.MA()
Out[]:
[[0, 1, 1, 0, 0, 1],
 [0, 0, 0, 1, 0, 0],
 [0, 1, 0, 1, 0, 0],
 [1, 0, 0, 0, 1, 1],
 [1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0]]

 In []:
```