**PRÁCTICA 29 | GRAFOS**

# Introducción

A diferencia de los árboles binarios que sólo podían tener como máximo dos nodos hijos, ahora se aumenta un poco la complejidad, pero muy probablemente también la utilidad al usar esta nueva estructura de datos. En un grafo, sus nodos o vértices pueden ser adyacentes hasta consigo mismos y con cualquiera de los demás vértices existentes. Esta práctica servirá para terminar de complementar los cálculos o funciones básicas que se pueden hacer sobre dicha estructura de datos, y así, acercarnos a descubrir algunas de sus aplicaciones en la vida real.

# Desarrollo

Esta práctica fue muy versátil en cuanto a complejidad. Algunos puntos estuvieron muy sencillos de codificar. Sin embargo, un método con el que me apoyo para hacer los del final, me tomó bastante tiempo pensarlo, diseñarlo y codificarlo.

Desde el punto dos, que es el método que identifica si hay adyacencia entre dos vértices, hasta el seis, que indica el grado de un vértice, resolver los problemas estuvo bastante sencillo.

En mi diseño busqué, por ejemplo, crear más métodos para aumentar la accesibilidad de la clase a futuras extensiones, pero que a la vez cubrían las necesidades de algunos de los requeridos en la actual práctica. Algunos de esos métodos extra creados fueron: *lazos, aristasParalelas* y *generarCamino*, que son utilizados de momento por funciones solicitadas para la práctica en cuestión, pero que también podrían ser útiles en otro momento.

Hay algunos cambios que se hicieron al proyecto, pero que al final dejaron de verse proyectados en la práctica porque causaban errores o había una mejor forma de hacerlo.

El primer cambio que hice fue agregar dos métodos en la clase **Matriz**,donde uno elimina una fila de ésta y otro hace lo mismo, pero con una columna; todo esto con el fin de ayudar a generar el funcionamiento del método *eliminarVertice*. Al final terminé cambiando y modificando éstos comportamientos a la misma clase **GrafoMatrizA**,con los métodos *recorrerRenglonVertice* y *recorrerRenglonColumna*porque en el código original situado en **Matriz** disminuía su tamaño, y, por ende, en el grafo daba errores. De todos modos, conservé las funciones en **Matriz** por si en algún momento se necesitan.

Otra de las modificaciones que hice y que al último no utilicé fue la de crear una interfaz **IPila** para abstraer a **Pila** y **PilaLista**. El motivo fue por una parte unificar las dos clases, que tienen los mismos métodos, y por otra, en el método *generarCamino* (que es utilizado por las últimas funciones solicitadas en la práctica) regresar una implementación de dicha interfaz conteniendo la ruta generada entre dos vértices en caso de necesitarse. Para este caso cambié el uso de pilas a una implementación de la interfaz **Lista** para evitar tener la limitante, dentro y fuera del método que genera el camino, de sólo poder ver el último elemento insertado. Al igual que con la clase **Matriz**,el cambio se conserva porque no es algo que esté de más tener.

Los tres últimos métodos, como se dijo antes, se apoyan de la función *generarCamino*; este método se encarga de trazar la ruta entre el origen y el destino guardando también los vértices visitados por si algún método externo lo requiere. El método tiene una peculiaridad, y es que comienza a buscar el camino partiendo del destino al que tiene que llegar; de esta manera se puede aumentar la eficiencia del proceso, en el caso de que ningún vértice apunte hacia el nodo que se tiene que llegar se evita hacer iteraciones sin resultados. Finalmente, cabe mencionar que incluí un enum que recibe este mismo método para indicar qué tipo de camino se desea generar, así como también éste mismo hace uso de una clase nueva de nombre **Arista** que ayuda a evitar que el algoritmo se cicle pasando numerosas veces por el mismo par de vértices.

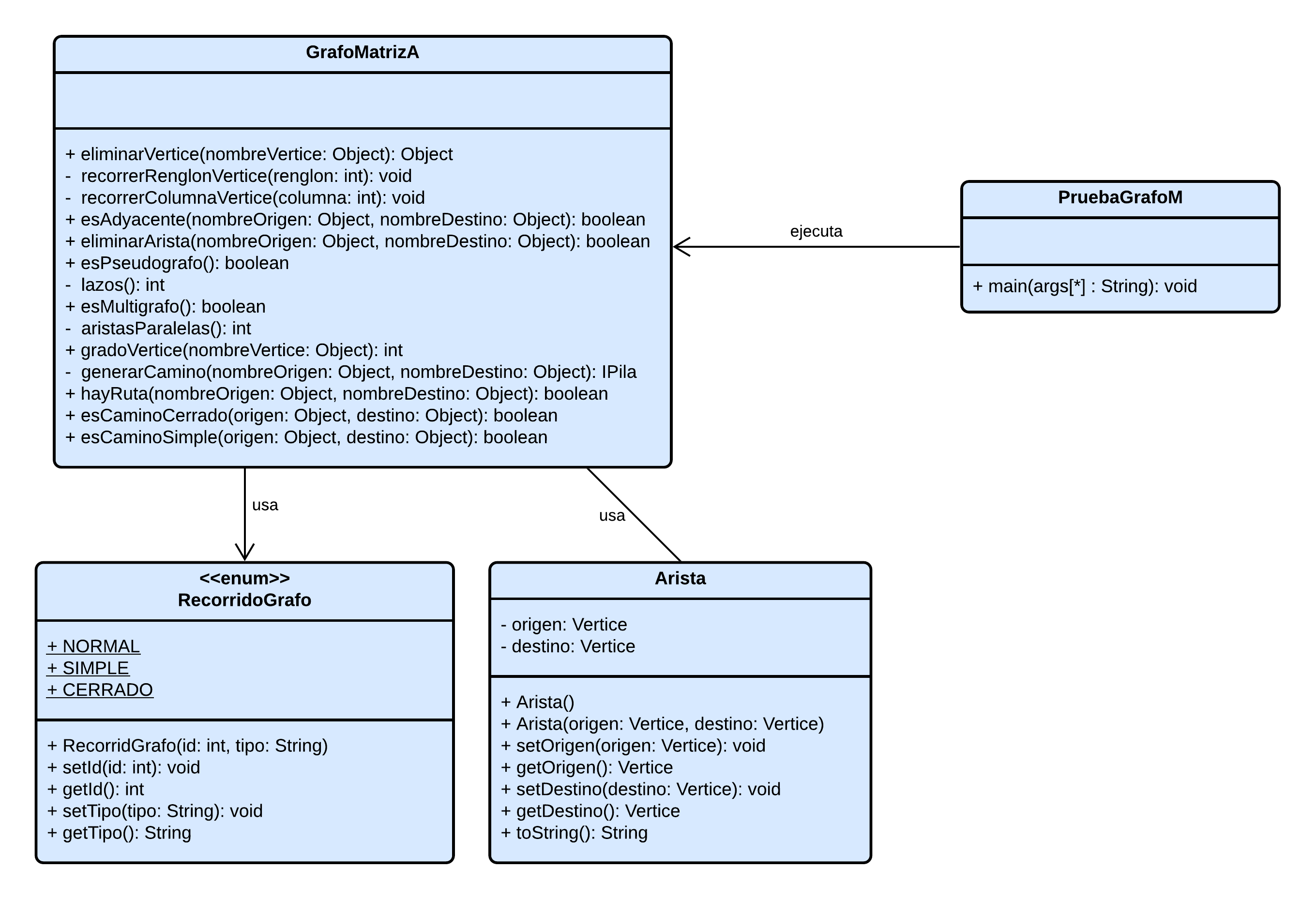
A continuación, se indican los archivos creados y modificados y su ubicación en el proyecto (**no se hará mención de los que no fueron usados en la práctica)**.

* ***GrafoMatrizA.java (paquete ednolineal).***
* ***Arista.java (paquete catalogos).***
* ***RecorridoGrafo.java (paquete utilidades).***
* ***PruebaGrafoM.java (paquete interfaces)***

La clase anterior contiene las pruebas de esta práctica. Para esta ocasión se evitó crear una nueva clase de prueba y usar la que el profesor en los vídeos agregó.

# Diagrama de clases

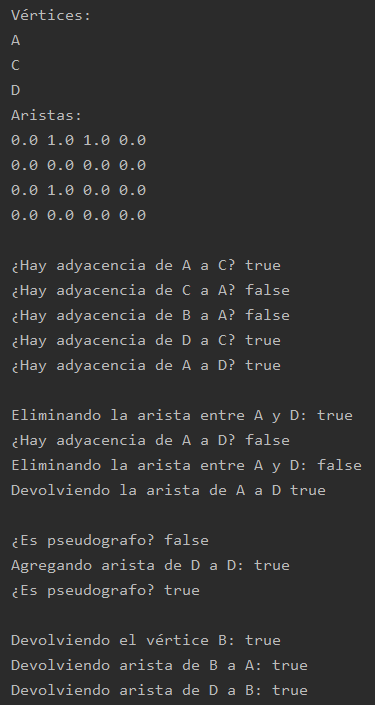
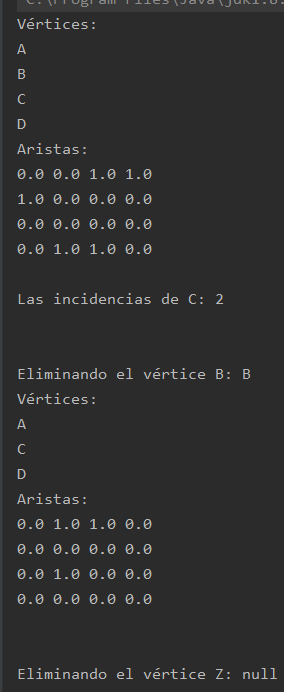
En el diagrama de clases se tiene una vista más cómoda de lo que se hizo en cada una de las clases mencionadas anteriormente. Cabe destacar que para las que ya existían en el proyecto solamente se ilustran los métodos o atributos que fueron agregados. En cambio, para las nuevas se muestra en su totalidad métodos y atributos que fueron incluidos. **En el diagrama tampoco se hará mención de lo que se creó y al último no se utilizó en la práctica**.

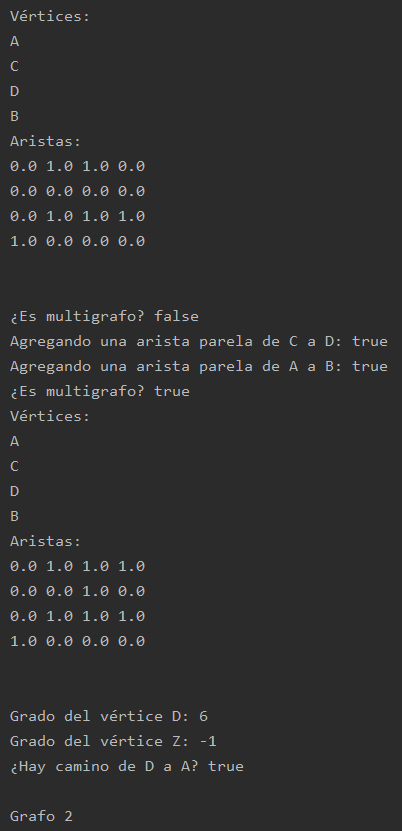
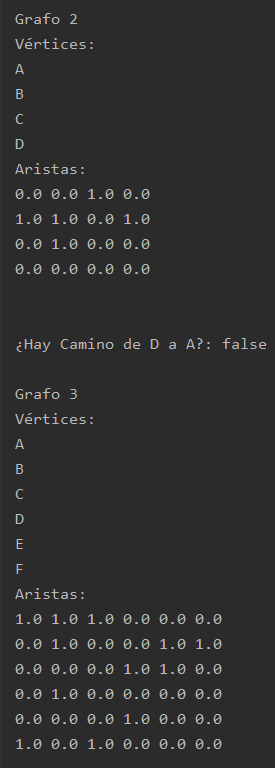


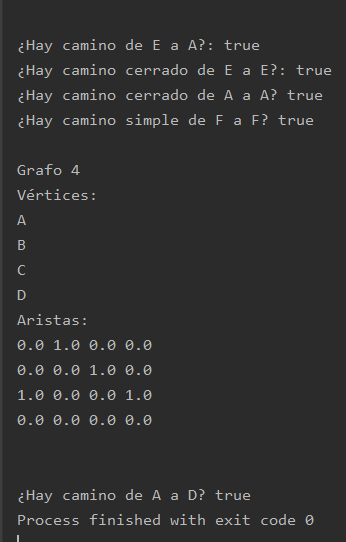
**Si desea revisar la documentación de lo que fue agregado o modificado al proyecto vaya a las siguientes rutas:**

* **/docs/catalogos/Arista.html**
* **/docs/utilidades/RecorridoGrafo.html**
* **/docs/ednolineal/GrafoMatrizA.html**
* **/docs/interfaces/PruebaGrafoM.html**

# Capturas del programa funcionando



# Conclusiones

Es bastante notable que ahora se trabaja con una estructura de datos muy compleja en cuanto a recorrido de sus vértices se trata. Sin embargo, hacer estos ejercicios sigue ayudando a aumentar la capacidad de analizar y diseñar algoritmos para situaciones cada vez más difíciles, además, de por supuesto, entender en qué situaciones es más conveniente usarlas y no sólo usar por usar una estructura de datos específica.