Algoritmo de búsqueda de ciclos con ordenamiento topológico y componentes fuertemente conexas de un grafo dirigido. Eficiencia, método y coste.

**Sumario**

[1. Introducción. 3](#__RefHeading___Toc1380_2657948309)

[1.1 Algoritmo de búsqueda de ciclos 3](#__RefHeading___Toc1382_2657948309)

[1.2 Algoritmo de ordenamiento topológico en grafo acíclico 3](#__RefHeading___Toc1384_2657948309)

[1.3 Algoritmo de componentes conexas en grafo cíclico 3](#__RefHeading___Toc1386_2657948309)

[1.4 Juntando las piezas del rompecabezas 3](#__RefHeading___Toc1388_2657948309)

[1.5 Comentarios en el código 3](#__RefHeading___Toc2462_2657948309)

[2. El algoritmo en cuestión 3](#__RefHeading___Toc1969_2769128295)

[2.1 Implementación de la estructura Grafo 3](#__RefHeading___Toc1394_2657948309)

[2.2 Modificaciones a la DFS 4](#__RefHeading___Toc1396_2657948309)

[2.3 Componentes fuertemente conexas. Algoritmo de Kosaraju 6](#__RefHeading___Toc1398_2657948309)

[3. Funcionamiento, eficiencia, costes, medición y gráficas 7](#__RefHeading___Toc1966_2769128295)

[3.1 Medición 7](#__RefHeading___Toc1402_2657948309)

[3.2 Eficiencia y costes del algoritmo 7](#__RefHeading___Toc1404_2657948309)

[3.3 Gráficas 7](#__RefHeading___Toc1406_2657948309)

[3.4 Funcionamiento y casos simples 10](#__RefHeading___Toc1408_2657948309)

[4. Archivos de la práctica y cómo usarlos 11](#__RefHeading___Toc2465_2657948309)

[4.1 Casos Simples 11](#__RefHeading___Toc1412_2657948309)

[4.2 Casos Largos y Generador de Casos 11](#__RefHeading___Toc1971_2769128295)

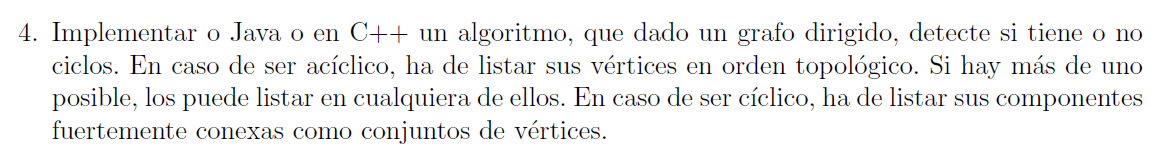
[4.3 CasosGNU y ‘practicaGNU.cpp’, las herramientas para lotes de casos 11](#__RefHeading___Toc1416_2657948309)

[4.4 La versión “user friendly” del algoritmo y el archivo del código C++ 12](#__RefHeading___Toc1418_2657948309)

[5. Bibliografía y sitios web consultados 12](#__RefHeading___Toc1973_2769128295)

# 1. Introducción.

El enunciado de la práctica tal y como figura en el documento de prácticas es:

En este caso se ha elegido el lenguaje C++.

El objetivo es claro, diseñar un algoritmo que: Primero decida si el grafo es acíclico o no (búsqueda de ciclos), luego para un grafo acíclico dirigido, devuelva un orden topológico, y que para un grafo con ciclos, devuelva las componentes fuertemente conexas.

### 1.1 Algoritmo de búsqueda de ciclos

Para la búsqueda de ciclos se usará un algoritmo de Depth-First Search (a partir de aquí referido como DFS), ya que es óptimo, pues tiene complejidad lineal O(V+E) y, como se sabe por teoría, no se pueden encontrar ciclos con una complejidad menor.

### 1.2 Algoritmo de ordenamiento topológico en grafo acíclico

Existen varios algoritmos posibles que devuelven un orden topológico para un grafo acíclico dado. Uno ingenioso utiliza el número de aristas entrantes para cada nodo. Pero también se pueden sacar los nodos por recorrido DFS en orden de finalización lo cual nos da ya un orden topológico.

### 1.3 Algoritmo de componentes conexas en grafo cíclico

Para este resolver este problema hay varias estrategias. Algunas combinan DFS y BFS. También existen el algoritmo de Tarjan, el de Gabow, el de Kosaraju y otros tantos que utilizan pilas. Pero todos ellos tienen una estructura muy similar a la DFS.

### 1.4 Juntando las piezas del rompecabezas

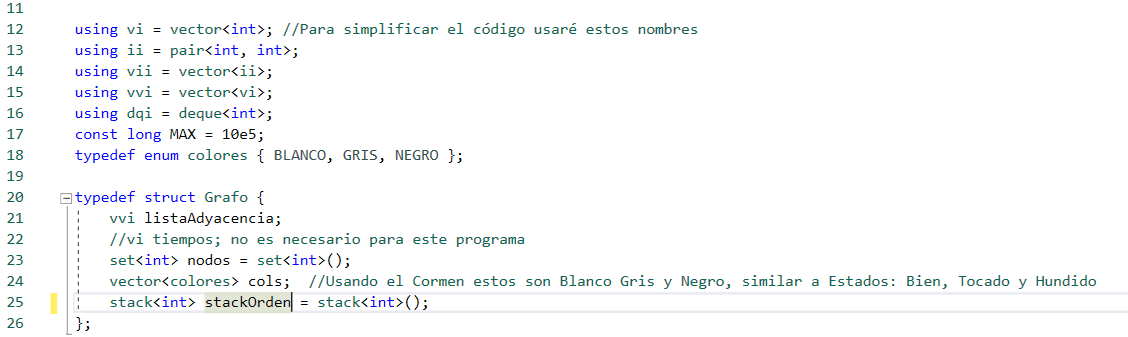
Por todo lo anterior, parece entonces que la solución al problema se puede encontrar a través de una DFS personalizada que, aunque sobrecargada por la complejidad añadida, podrá ser de orden lineal.

### 1.5 Comentarios en el código

En el archivo con el código se han incluido algunos comentarios. Asimismo, las explicaciones de las funciones están en sus encabezados, utilizando el formato /\*\*<texto>\*/. Este formato permite que las explicaciones sean visibles manteniendo el cursor por encima del encabezado de la función o en sus llamadas.

# 2. El algoritmo en cuestión

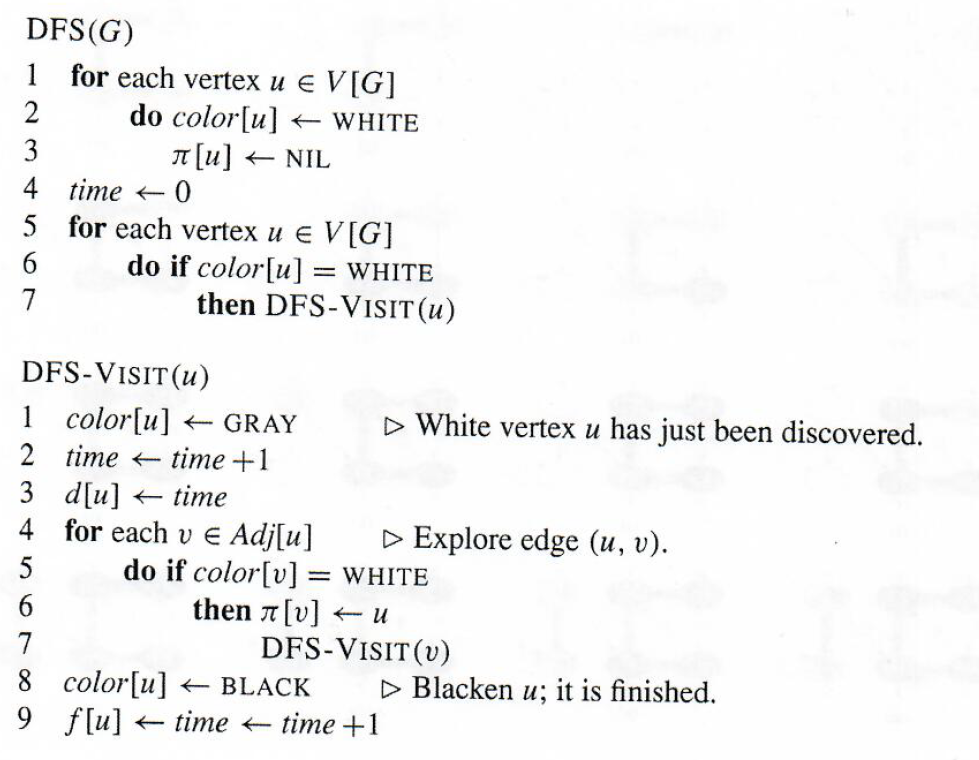
### 2.1 Implementación de la estructura Grafo



Esta implementación de grafo no guarda los tiempos de finalización porque para este problema no es necesario. Tiene un tipo de dato ‘set’ para guardar los nodos que forman parte del grafo. Esto hace que la operación insertar en este tipo de grafo sea de O(logn) pero esto queda fuera del cómputo del coste del algoritmo ya que hay otras formas de guardar los nodos que, permitiendo duplicados, lo hacen en O(1) y no repercuten en el coste verdadero de una versión de este algoritmo hecho para ese tipo de dato, pero teniendo en cuenta que los casos largos están hechos de forma aleatoria, por la estabilidad y la claridad del código del algoritmo, decidí guardar los nodos en un conjunto. Utiliza listas de adyacencia, basadas en un vector de vectores de enteros, y un vector de colores que funciona, tal y como se explica en *Cormen et alter* ([1]), con tres colores: blanco, gris y negro. A esta implementación se le añade una pila en la que se guardarán los nodos en un orden específico que explicaré a continuación.

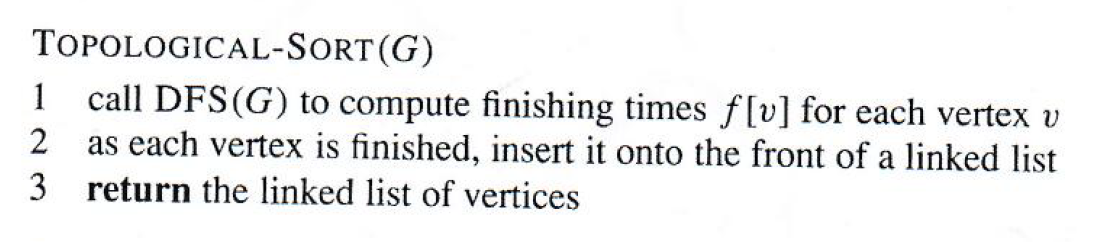
### 2.2 Modificaciones a la DFS

El esqueleto de una DFS según [1] es:

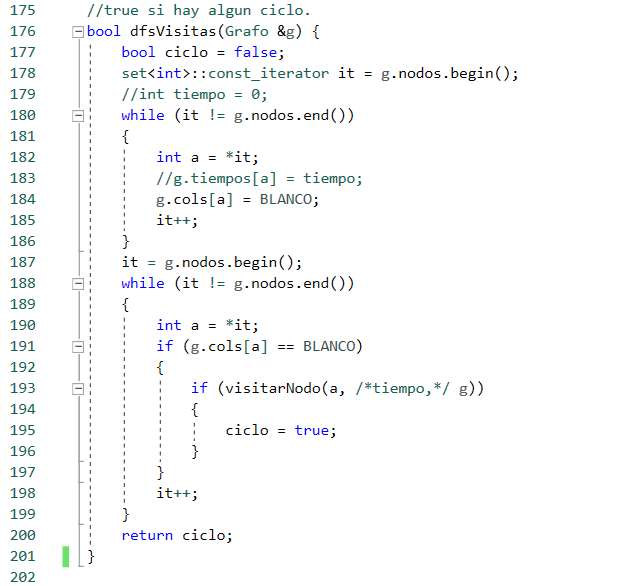


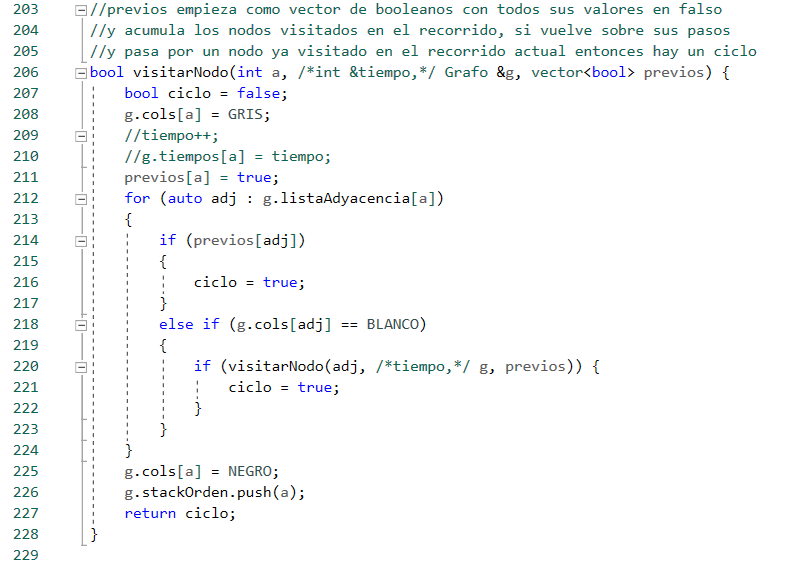
(La adaptación a código C++ de este pseudocódigo se encuentra incorporada con el resto del algoritmo.)

Asimismo, el orden topológico se puede obtener según [1] siguiendo el siguiente algoritmo:

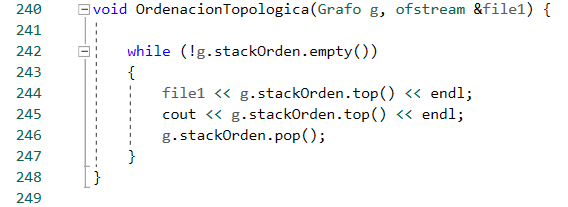
Una DFS que añade cada nodo de un grafo acíclico a una lista, al finalizar el recorrido de dicho nodo, ya devuelve una lista que no es más que una cadena (conjunto totalmente ordenado) con inicio en dicho nodo. Esto ya supone una modificación de la DFS de búsqueda de ciclos relativamente simple, sólo hay que añadir a una lista el nodo según se termina su recorrido, en mi caso a una pila.

Con las observaciones anteriores, la DFS modificada quedaría de la forma siguiente:



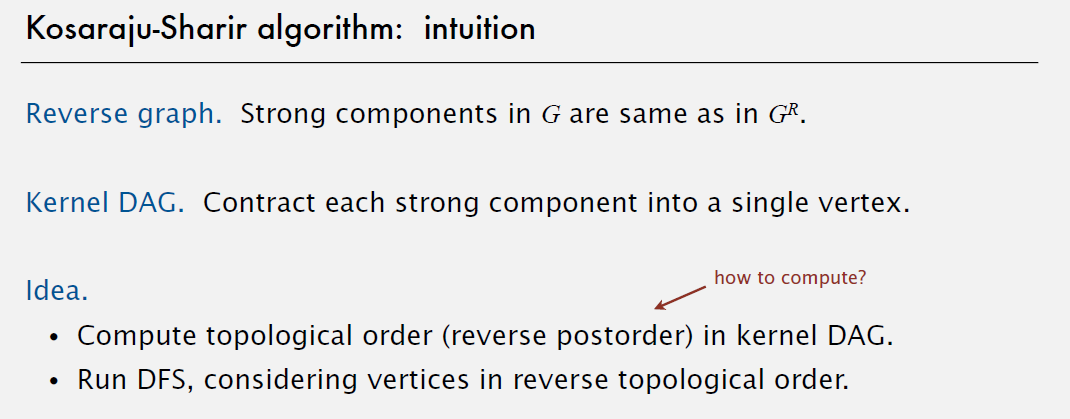
Que utiliza la función auxiliar para el recorrido:

Con esta DFS ya se resuelve tanto el problema de encontrar ciclos como la obtención de un orden topológico (si lo hay). En efecto, si para un grafo G el procedimiento dfsVisitas(…) devuelve *falso*, el orden se encuentra en la pila stackOrden. Si devuelve *verdadero* significa que es un grafo cíclico. Por tanto, lo único que queda por resolver es cómo conseguir las componentes fuertemente conexas.

Como forma parte de la solución, se muestra a continuación cómo se obtiene el orden topológico de la pila conseguida al final de la DFS modificada:

### 2.3 Componentes fuertemente conexas. Algoritmo de Kosaraju

El algoritmo de Kosaraju, segúnSedgewick y Wayne en [2], se indica a continuación:



Queriendo decir esto que las componentes fuertemente conexas se mantienen en el grafo inverso. La idea es calcular el grafo inverso y aplicar una DFS en ciertos nodos en cierto orden. Ya que el grafo inverso mantiene las componentes conexas, si se consigue ejecutar la DFS para todos los nodos en cierto orden, cada recorrido de DFS se corresponderá con una componente conexa. Este orden es de vital importancia porque, de empezar en un nodo equivocado, se pueden tener en cuenta como nodos pertenecientes a una componente conexa nodos adyacentes a una componente conexa en el grafo invertido.

Este orden tan específico se corresponde con un *orden particular*: el orden de finalización de la DFS ejecutada en el grafo dirigido original, el ya guardado en *stackOrden* tras la llamada a *dfsVisitas*.

Sabiendo esto la solución al problema es evidente. Usando la idea del algoritmo de Kosaraju y Sharir y la DFS modificada lo único que queda es:

* Crear una función que obtenga el grafo inverso de un grafo dado y le quite las marcas de color,
* Crear una función simple que haga un recorrido en profundidad y saque por pantalla y marque los nodos de un grafo empezando en un nodo. Y por último
* Invocar a esta función dando como nodo inicial los nodos dados en el orden que nos proporciona la DFS modificada.

El resultado es:



Estos elementos ya conforman la solución completa al problema.

# 3. Funcionamiento, eficiencia, costes, medición y gráficas

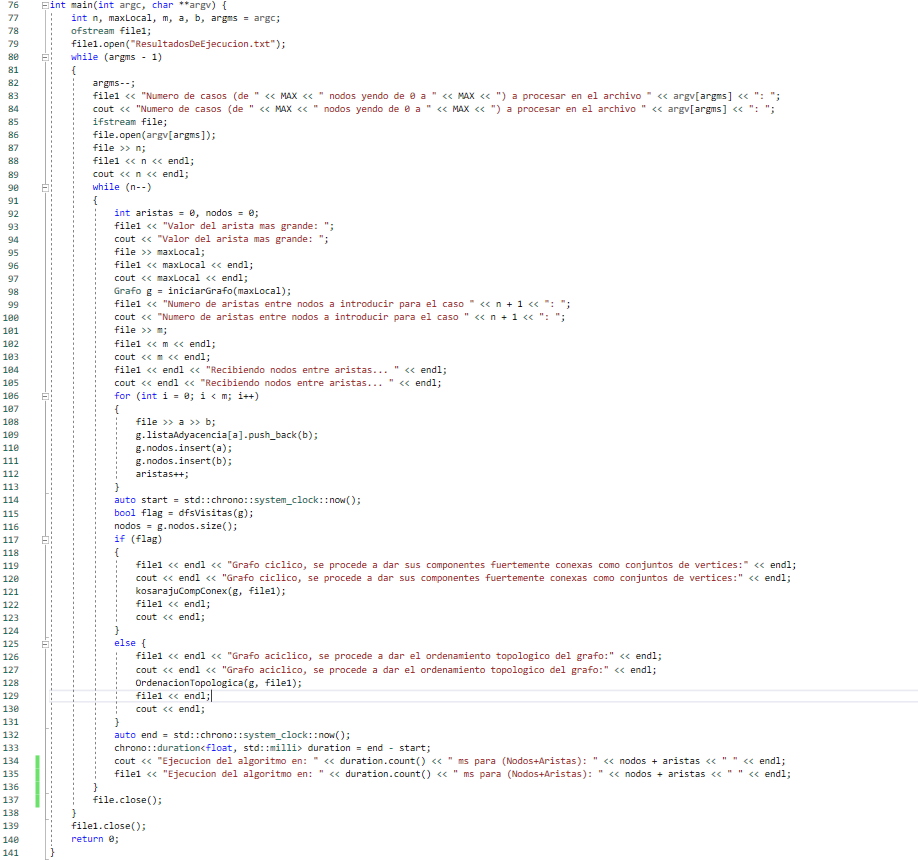
### 3.1 Medición

Para medir cuánto dura una ejecución del algoritmo para cierto grafo se usará la librería ‘chrono’. Para ello se medirá el tiempo del sistema en el momento anterior a cierta ejecución y en otro posterior, restando al segundo el primero se puede conseguir con una precisión de milisegundos el tiempo empleado en la ejecución.

### 3.2 Eficiencia y costes del algoritmo

Esta sección podría estirarse con cálculos para demostrar que el coste del algoritmo diseñado no es de O(n^m) donde ‘n’ sería el tamaño de los datos de entrada y ‘m’ el número de bucles que aparecen en el código. Sin embargo, y con el fin de mantener esta memoria sencilla, explicaré el coste lineal a través del funcionamiento de la DFS que, aunque modificada, es el único algoritmo utilizado en la solución.

Es importante observar que, cuando se recorre un grafo en profundidad partiendo de un nodo, los nodos visitados en ese recorrido se marcan y no serán visitados en un futuro. De esta forma, si el primer recorrido DFS pasase por todos los nodos entonces ningún otro recorrido DFS lanzado en ningún nodo recorrería ninguno. Es decir, la DFS modificada tiene coste del orden de O(V+E) ya que recorre cada nodo y cada arista a lo sumo una vez. Las otras funciones que se usan son la de revertir un grafo; que tiene trivialmente coste lineal sobre los aristas del grafo, la de ordenación topológica; que tiene de coste el tamaño de la pila que es el número de nodos, y la dfsImprimir(…) que es otra DFS del mismo coste. Analizando la parte de la función ‘*main’* que corresponde al algoritmo:

Se puede observar que el coste del algoritmo que conformaría la solución, cuyo principio estaría marcado por la variable ‘*start’* y cuyo final vendría dado por la variable ‘end’ en la función *‘main’* sería:

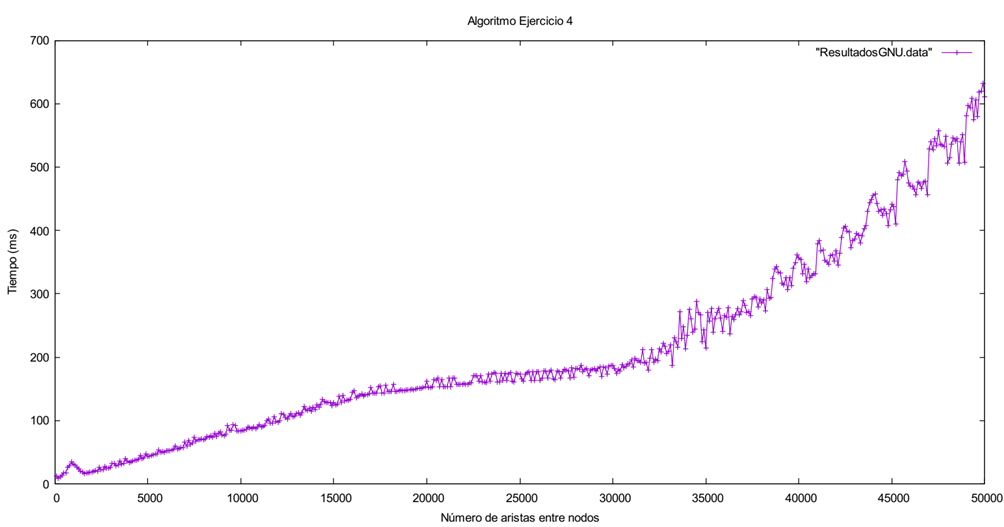
· **Cíclico**: (Coste de una DFS) + (Coste Kosaraju) = (Coste de una DFS) + (Coste de invertir) + (Coste de una DFS)

· **Acíclico**: (Coste de una DFS) + (Coste Topológico) = (Coste de una DFS) + (Coste de vaciar pila)

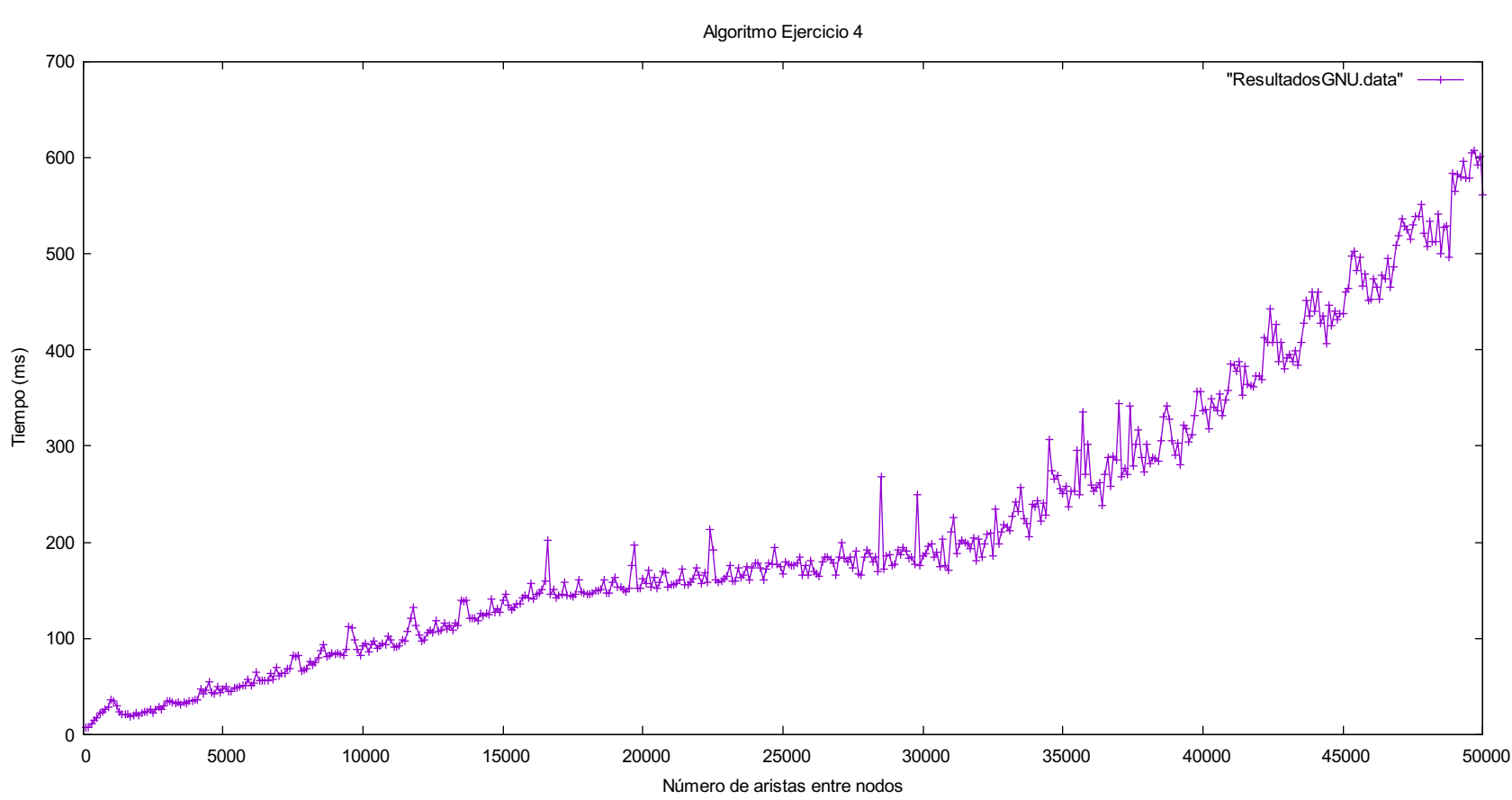
En ambos casos se obtiene el mismo orden de magnitud, que es lineal, pero en el caso de un grafo cíclico las constantes escondidas en la *big-O* son mayores. Esto se verá reflejado en las gráficas.

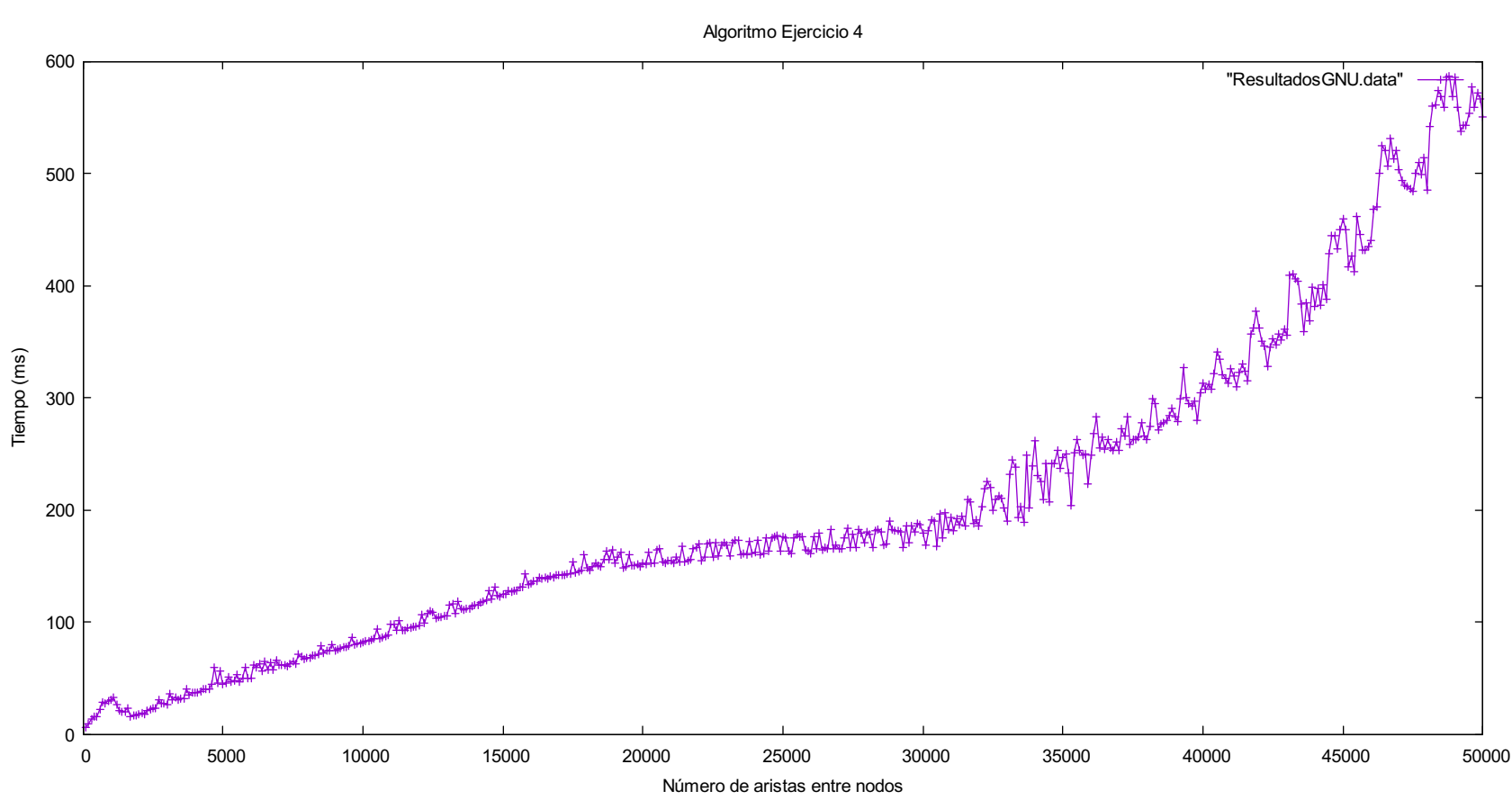
### 3.3 Gráficas

Para este apartado se han programado dos herramientas especiales: un generador de 500 casos de 100 a 50000 nodos en incrementos de 100 nodos y un algoritmo que escribe en un archivo el número de nodos del grafo del caso y el tiempo que le ha costado acceder a los nodos según el planteamiento del problema (no saca nada por pantalla). Es decir, el orden topológico caso de ser acíclico o nombrando las componentes conexas de ser cíclico. Estas gráficas se han generado con el ordenador desconectado de la red y con ningún proceso abierto más que los esenciales para ejecutar estas dos herramientas. Estos resultados que son emplazados por la segunda herramienta en “ResultadosGNU.data”, luego se pasaron por GNU para conseguir las siguientes gráficas en 5 ocasiones para 5 conjuntos de 500 casos aleatoriamente diferentes (usando como semilla aleatoria el tiempo del sistema):

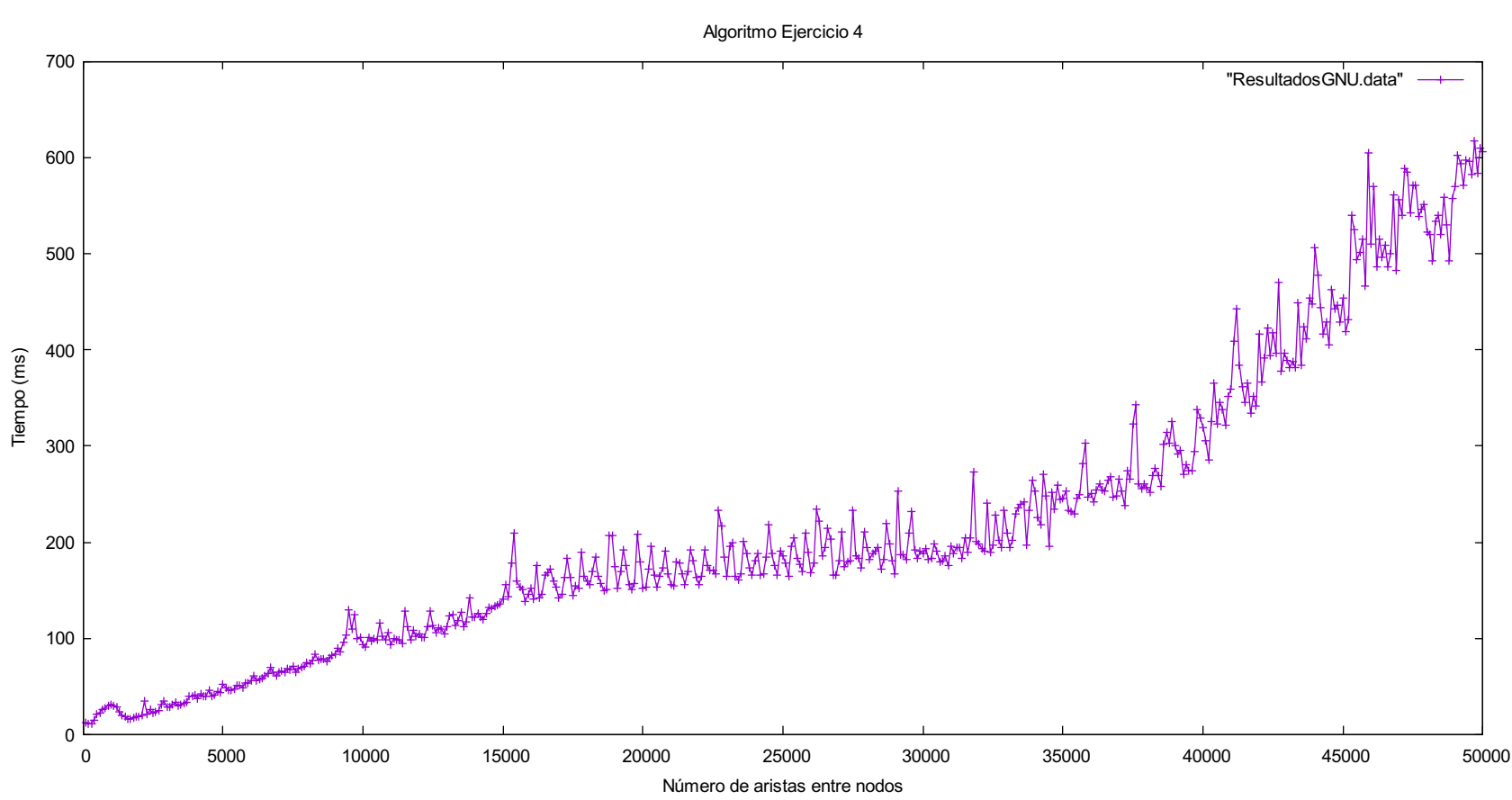


Gráfica 1

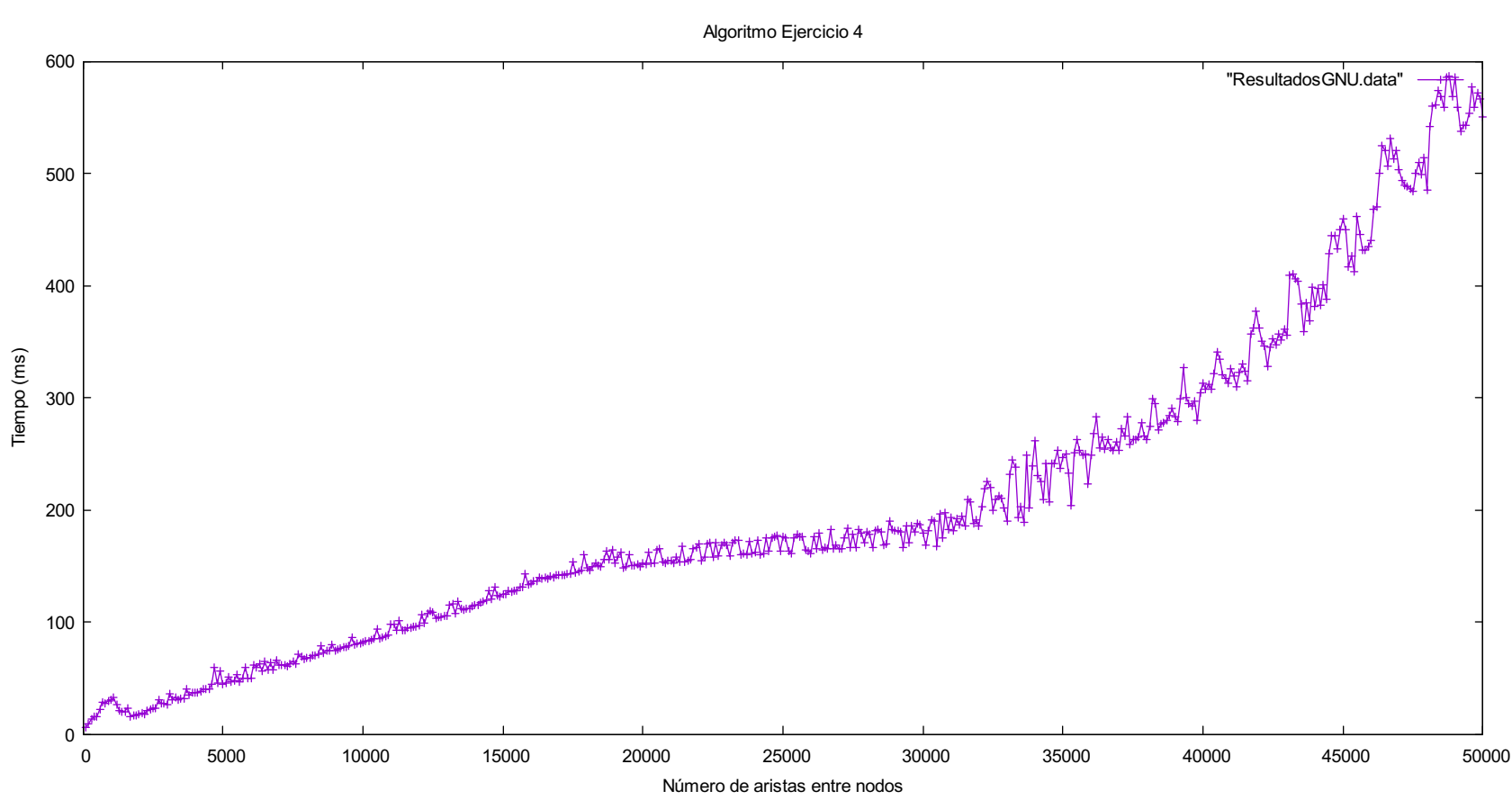
Gráfica 2



Gráfica 3



Gráfica 4



Gráfica 5

Se pueden observar varias características de las tablas temporales obtenidas. La primera sería la pequeña subida en tiempo de los casos pequeños. Esto se debe a que aparentemente los casos pequeños, en el generador de casos, suelen ser grafos cíclicos, que como ya hemos visto tendrían un coste parecido a ‘3n’ comparado con un ‘2n’ para los acíclicos. Para los grafos cíclicos el algoritmo tarda un poco más porque aparte del coste lineal de invertir el grafo debe ejecutar una DFS, pero ambos tienen coste lineal en teoría. Esto hace que las gráficas tengan picos como un filo serrado. Por último, cabe destacar el valle entre los 20000 y los 35000 nodos. Supongo que esto se debe a optimizaciones internas de caché, accesos preventivos a memoria y demás optimizaciones de la estructura del computador. Realmente, no dispongo de una explicación clara y con certeza para el valle que aparece en ese intervalo. Poco a poco la gráfica vuelve a ser creciente, pareciéndose a un crecimiento de un algoritmo de coste lineal. En mi experiencia este algoritmo es capaz de procesar casos con un millón de aristas sin problema.

Cabe destacar que en este caso el coste real se corresponde con el coste teórico, es decir lineal, si se hubiese usado una matriz de adyacencia en vez de una lista de adyacencia el coste sería cuadrático. Ya que al tener que inspeccionar cada posición de la matriz las DFSs tendrían coste cuadrático.

### 3.4 Funcionamiento y casos simples

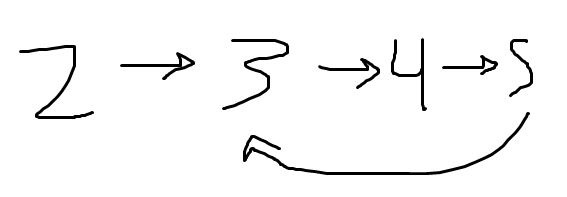
La forma en la que funciona el algoritmo sería una secuencia de los algoritmos ya explicados. La secuencia de instrucciones es como sigue:

1: Llamar a la DFS modificada que llena una pila por orden de finalización y devuelve verdadero o falso según si hay ciclos o no.

2. Si devuelve verdadero: Aplicar Kosaraju usando el orden de la pila.

3. Si devuelve falso: Sacar el orden topológico de los nodos guardado en la fila.

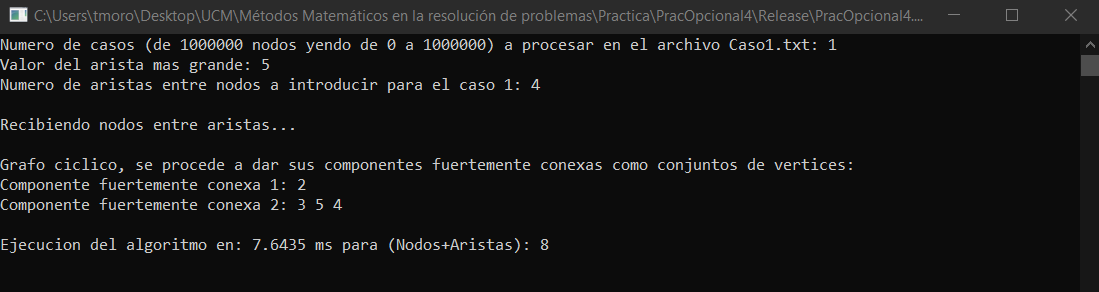
Supongamos el caso simple proporcionado en la carpeta de archivos “Caso1.txt” mostrado aquí:

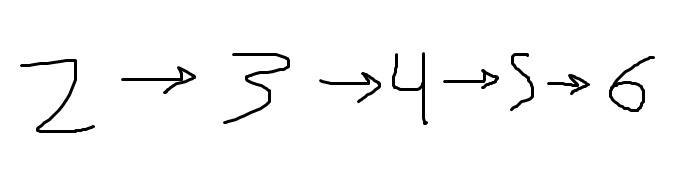


Es por tanto cíclico. Nuestro algoritmo, tal y como muestra el ‘*main’,* lanza la DFS para encontrar ciclos que guarda el orden topológico. La pila tras el DFS es:

[ 5,4,3,2 ]

y el resultado de la llamada es *verdadero*. Por tanto ha detectado un ciclo. Por ello, en el bloque *if* elige Kosaraju. La función de invertir grafo devuelve el grafo invertido de nodos blancos y luego se recorre en profundidad el grafo invertido tomando como puntos de entrada los de la pila. Éste sería el conjunto de nodos descrito arriba, de izquierda a derecha, ya que las pilas son LIFO.

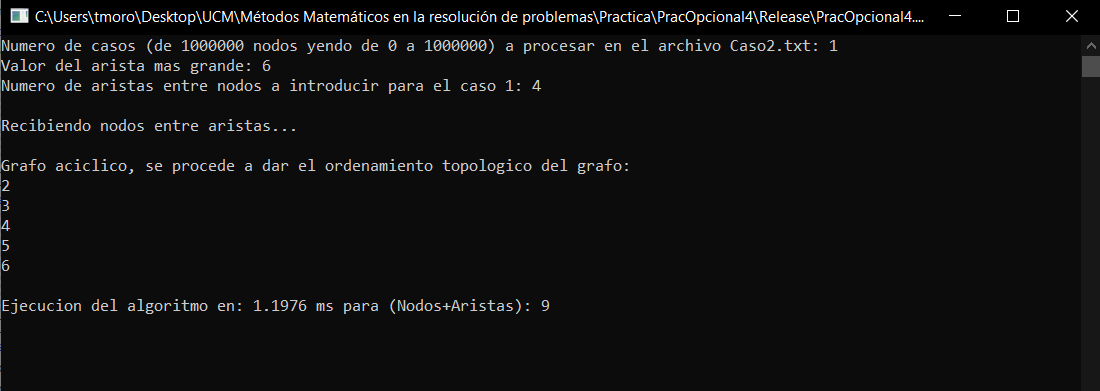
Por tanto, el recorrido sobre el 2 en el grafo inverso sólo da el 2, ya que en el grafo inverso el 2 no tiene nodos adyacentes. Luego empieza un recorrido en profundidad sobre el 3, que devuelve la componente conexa que se mantiene en el grafo inverso, con los tres nodos restantes. Esto es evidente en el resultado de la ejecución:

Ahora se tratará otro caso simple acíclico, contenido en la carpeta con los archivos de la práctica, el archivo “Caso2.txt”:

La DFS modificada guarda el orden topológico del grafo acíclico. La pila tras el DFS es tal que:

[ 6,5,4,3,2 ]

El resultado de la DFS es *falso*. Por tanto, la función de orden topológico extrae los contenidos de la pila por pantalla, que son los nodos dados en un orden topológico:



# 4. Archivos de la práctica y cómo usarlos

### 4.1 Casos Simples

La carpeta “CasosSimples” contiene los ficheros “Caso1.txt” y el “Caso2.txt”, entre otros. Éstos son casos simples de pocos nodos hechos a mano cuyas soluciones se pueden sacar sin uso de ordenador que sirven para hacer el seguimiento teórico y comprobar que la solución proporcionada es correcta.

### 4.2 Casos Largos y Generador de Casos

* + 1. La carpeta “CasosLargos” contiene varios grafos de distintos tamaños creados con el archivo ‘GeneradorDeCasos.cpp’. Éste programa es intuitivo de usar, sólo hay que compilarlo y ejecutarlo. En la misma carpeta del ‘cpp’ sacará los casos del formato que le sean pedidos. Estos casos de la carpeta no son de tamaño apto para hacer a mano.

### 4.3 CasosGNU y ‘practicaGNU.cpp’, las herramientas para lotes de casos

En la carpeta “PracticaGNU” se encuentran las herramientas “GeneradorGNU.cpp” y “PracticaGNU.cpp”, así como sus ejecutables. “GeneradorGNU.exe” (la compilación del código en “GeneradorGNU.cpp”) creará 500 archivos en su misma carpeta con casos de 1 grafo de 100 a 50000 nodos titulados “CasoGNU<n>.txt”, donde *n* es el número de nodos del grafo. “PracticaGNU.exe” (la compilación del código en “PracticaGNU.cpp”) resolverá el problema sin mostrar nada en pantalla ni escribir nada en un archivo los grafos de cada uno de estos archivos y escribirá en “ResultadosGNU.data” el valor *n* y el tiempo (en milisegundos) que le ha costado resolverlo. Este archivo es perfecto para GNU ya que con pedirle:

>> plot “ResultadosGNU.data”

GNU sacará una nube de puntos conforme a los resultados de la ejecución.

Para conseguir una gráfica como las enseñadas basta con:

1: Tener “GeneradorGNU.exe” y “PracticaGNU.exe” en el mismo directorio.

2: Ejecutar “GeneradorGNU.exe”.

3: Ejecutar “PracticaGNU .exe”.

4: Ejecutar la línea de código anterior en GNUplot en el directorio.

5: Exportar la gráfica.

### 4.4 La versión “user friendly” del algoritmo y el archivo del código C++

Dentro de la carpeta principal se encuentran “PracOpcional4.cpp” y “PracOpcional4.exe”, siendo esta segunda el resultado de compilar la primera. Para su correcto funcionamiento se debe ejecutar con argumentos de entrada nombres de archivos de texto en el mismo directorio en el que se encuentre el ejecutable que contengan grafos y especifiquen casos de la siguiente forma:

1: El número de casos.

2: El valor del nodo más grande del caso a tratar ‘v’.

3: El número de aristas ‘e’ a introducir para ese caso, ninguna de ellas deberá sobrepasar ‘v’.

4: ‘e’ líneas con el número del primer nodo y el segundo, siendo el primero el origen y el segundo el destino de la arista dirigida.

Una vez tengamos un archivo de texto de estas características en el mismo directorio que el ejecutable se va al directorio desde la línea de comandos y se ejecuta la línea:

>> PracOpcional4.exe <nombre del archivo>.txt

Esto debería mostrar por pantalla la solución tal y como pide el problema. Además, también proporciona el resultado en un archivo de texto titulado “ResultadosDeEjecución.txt”.

# 5. Bibliografía y sitios web consultados

[1]. Cormen, T.H *et alter*; (2009): *Introduction to Algorithms*. (2nd ed.). MIT Press and McGraw-Hill

[2]. Sedgewick, R; Wayne, K. (2011): *The textbook* Algorithms, (4th Edition). Addison-Wesley Professional.

3. https://www.geeksforgeeks.org/strongly-connected-components/

4. [https://en.wikipedia.org/wiki/Kosaraju%27s\_algorithm](https://en.wikipedia.org/wiki/Kosaraju's_algorithm)