Representación del conocimiento
Práctica 3

Víctor Castañeda Balmori, Mario Cuesta Rivavelarde,
Carlos García Arenal, Laro Ayesa Sánchez y Alexandru Solovei Popa

31/10/2025

En esta práctica, el objetivo era codificar y analizar el algoritmo de Chow-Liu en python. Para la implementación, hemos dividido el algoritmo en 3 pasos, tal y como se ha visto en clase.

1. Codificación y complejidad teórica

1.1. Obtener pesos (ordenados)

Este paso consiste en obtener la información mutua de todas las variables entre sí, mediante la estimación de máxima verosimilitud. Además, en nuestra implementación, devolvemos los pesos obtenidos entre cada variable, en forma de tabla ordenada de mayor peso a menor, lo que nos facilita el siguiente paso del algoritmo.

Crear $K = (V, E, W)$ tal que $w_{ij} = I(X_i, X_j, \mid D)$ para todo $\{v_i, v_j\} \in E$

Implementación en la función *obtener_pesos*. Asumiendo que las variables tienen cardinalidad k, la complejidad del paso 1 es $O(n^2 * k^2)$.

1.2. Crear árbol de recubrimiento máximo

En este paso, hemos diseñado 2 algoritmos distintos para calcular lo mismo. Simplemente seguir un algoritmo como el de Kruskal para a partir de la tabla de nodos e información mutua (pesos) ordenada, obtener un árbol de recubrimiento máximo.

Busca un arbol de recubrimiento de peso máximo $G = (V, E)$ de K

- **Implementación con DFS (búsqueda en profundidad):**

  La implementación de la función *obtener_arbol* corresponde a este paso. Añadimos los nodos en el grafo en $O(n)$, siendo n el número de nodos. Recorremos los pesos ordenados, la tabla de pesos ordenados tiene tantas filas como combinaciones de 2 nodos hay, es decir, $\binom{n}{2}$, $n*(n-1)/2$, es decir, $O(n^2)$. Además, mientras se recorren los pesos ordenados, se debe comprobar si añadiendo esa posible arista, se crea un ciclo en el grafo o no. Es decir, si ya hay una forma

de llegar desde el nodo1 al nodo2 en el grafo. Eso lo comprobamos con una búsqueda en anchura en la función *forma_ciclo*, *O(n+e)* , siendo e el número de aristas.
Por lo tanto, la complejidad de este paso es $O(n^2 * (n+e))$. Además, el número de aristas en la construcción del árbol de recubrimiento máximo, va a ser como mucho n - 1, por lo que podemos afirmar que la complejidad es $O(n^3)$.

- **Implementación con conjuntos**

  Esta implementación (*obtener_arbol_conjuntos*) consiste en mantener en una lista de conjuntos, las componentes conexas del árbol que construimos. Es decir, cada elemento de la lista corresponde con unos nodos que están conectados en el árbol. De forma que cada nodo solo puede aparecer en un elemento de la lista y con esa lista podemos crear el árbol de recubrimiento máximo.
  El algoritmo recorre la tabla de pesos de igual forma que antes $O(n^2)$. Luego recorre la lista de componentes conexas construidas hasta el momento en el árbol $O(n)$. Después en función de si los nodos se encuentran o no en las componentes conexas, se añade la arista o no al grafo. De esta forma, si ninguno de los dos nodos están en el árbol, se añade la arista y la nueva componente conexa. Si uno de los dos no está, se añade la arista y se actualiza la componente conexa de la lista. Si están en componentes conexas separadas, se unen y se añade la arista. El caso en el que no se añade una arista es aquel en el que ambos nodos están en la misma componente, ya que esto formaría un ciclo al añadirla.
  En el peor de los casos, el número de componentes conexas en la lista será $n/2$, por lo que el bucle de dentro es $O(n)$ y la complejidad del algoritmo $O(n^3)$. Sin embargo, en el mejor caso, la lista siempre tiene longitud 1 (una única componente conexa) y sería $O(n^2)$. En el apartado de complejidad empírica, vemos como esta variante es mucho mejor, en general, que la implementación con DFS.

1.3. Asignar direccionalidad

En este paso, simplemente se decide un nodo raíz y mediante una búsqueda en anchura se asigna la direccionalidad del árbol. Da igual el nodo que se escoja como raíz, porque al ser un árbol, las relaciones de la red bayesiana (dependencias) van a ser iguales.

Elige $v \in V$ a partir del que asignar direccionalidad a G hacia fuera

La complejidad de este último paso es *O(n+e)*. Dado que es un árbol, el número de aristas va a ser $n - 1$, luego la complejidad es $O(n)$. Su implementación es la función *asignar_direccionalidad*.

1.4. Complejidad del algoritmo

Tras el análisis realizado, la complejidad del algoritmo será $O((n^2 * k^2) + (O(n^3)))$ siendo n el número de nodos y k la cardinalidad.

2. Complejidad empírica del algoritmo

Para la complejidad empírica se han hecho pruebas para cada paso del algoritmo. El código usado esta en el fichero *graficas.py*.

2.1. Obtener Pesos

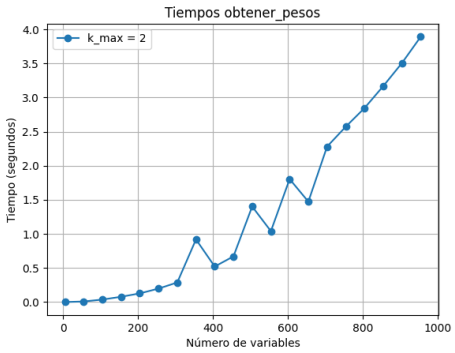

Figura 1: Cardinalidad 2

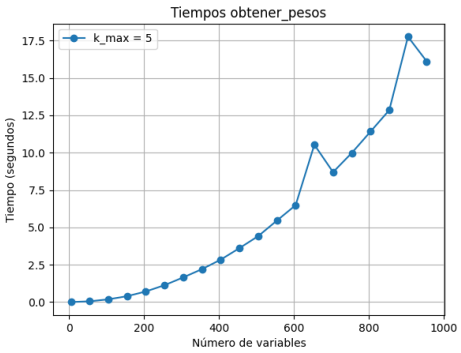

Figura 2: Cardinalidad 5

Figura 3: Comparacion cardinalidades

Las variables tienen todas esa cardinalidad así que teóricamente la relación de los tiempos debería ser $5^2/2^2 = 6{,}25$, en las gráficas se puede ver que nos da algo menos esa relación. Por lo demás todo concuerda con lo mencionado en la parte teórica.

2.2. Árbol de Recubrimiento Máximo

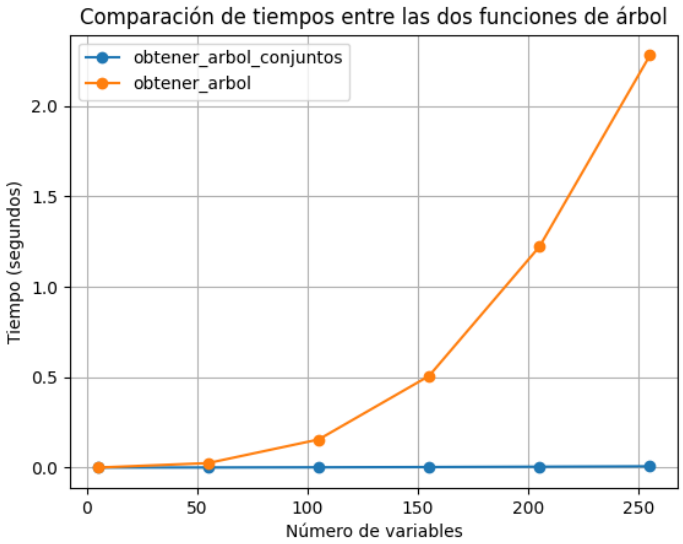

Figura 4: Comparación entre funciones

Se puede ver que a pesar de teóricamente tener la misma complejidad, la implementación con conjuntos es mucho mas rápida. Para la prueba se han hecho diez pasadas con diferentes variables y se ha hecho la media del tiempo. Aunque pueda parecer en Figura 4 que la complejidad del que usa conjuntos es lineal, si observamos la Figura 5 vemos que si se corresponde con la teórica.

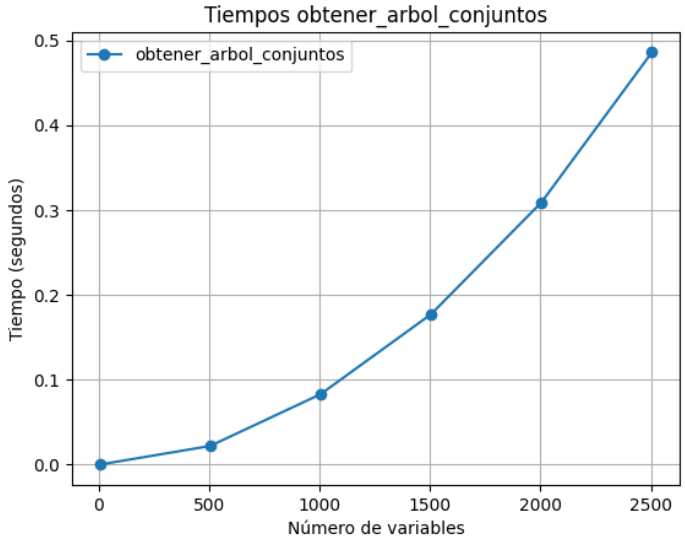

Figura 5: Función de conjuntos

2.3. Asignar Direccionalidad

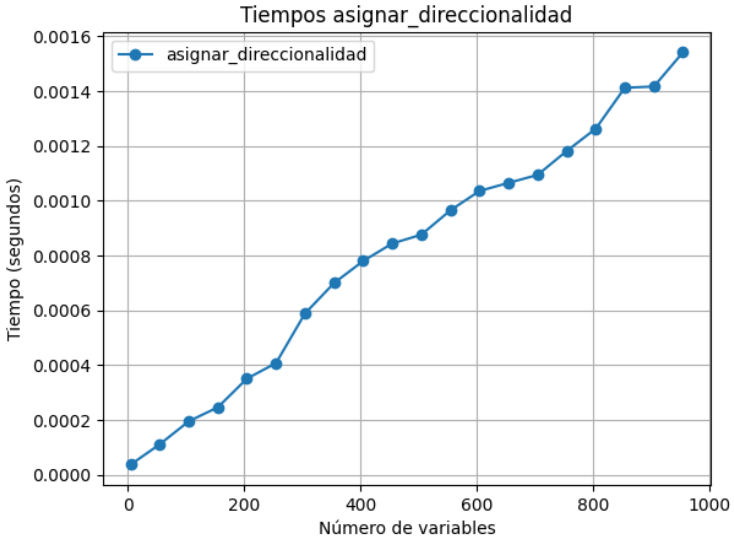

Figura 6: Asignar direccionalidad

Poco que mencionar, la complejidad es lineal como se ha dicho anteriormente.

2.4. Complejidad Total

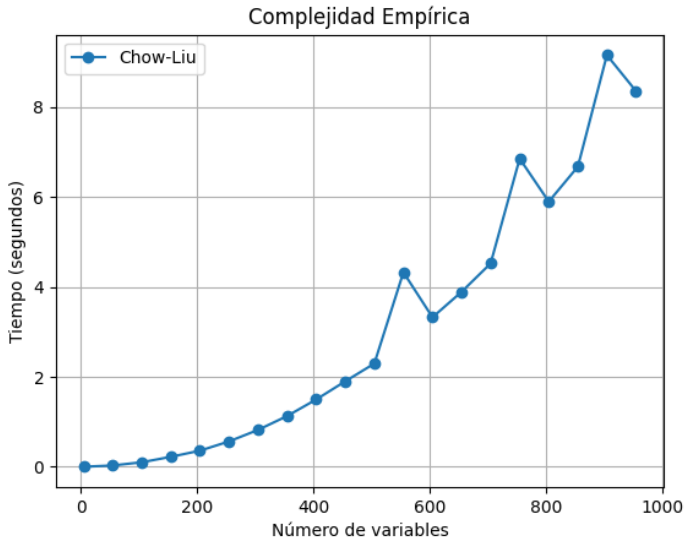

Figura 7: Asignar direccionalidad

Para la complejidad total se ha usado la función que usa conjuntos para obtener el árbol de recubrimiento máximo y se ha usado una cardinalidad aleatoria para las variables de entre 2 y 5. La forma es prácticamente igual a la del cálculo de los pesos si se usa la misma entrada.

3. Evaluación de la eficacia del algoritmo

Para comprobar el correcto funcionamiento de nuestras implementaciones, creamos un fichero llamado *creacion_aleatoria.py*.
Este script de Python genera un número predefinido de variables. A cada una de esas variables se le asigna un valor de cardinalidad aleatorio, entre 2 y un máximo. Además, se generan las dependencias entre variables y los valores de las tablas de forma aleatoria. Así, en cada ejecución se consigue un ejemplo distinto, aunque con un número fijo de variables.
Al ejecutar el script, se utiliza tanto la implementación original del algoritmo, que utilizaba búsqueda en profundidad; como la otra implementación, que utiliza conjuntos. Por ello, se muestra el árbol de pesos mínimos generado por cada implementación. Esto permite comparar como ambas implementaciones producen el mismo resultado.

A continuación, se muestra el árbol de Chow-Liu resultante tras asignar direccionalidad en cada caso, que puede ser distinto debido a la forma en la que se construye el grafo.

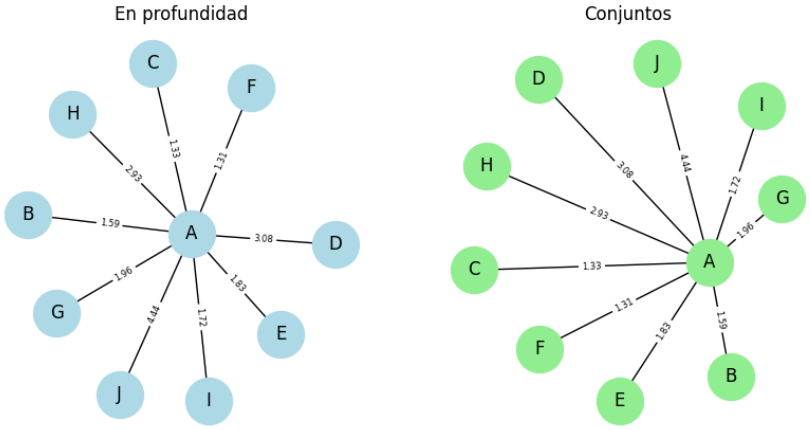

Figura 8: Ejemplo de arboles de pesos mínimos.

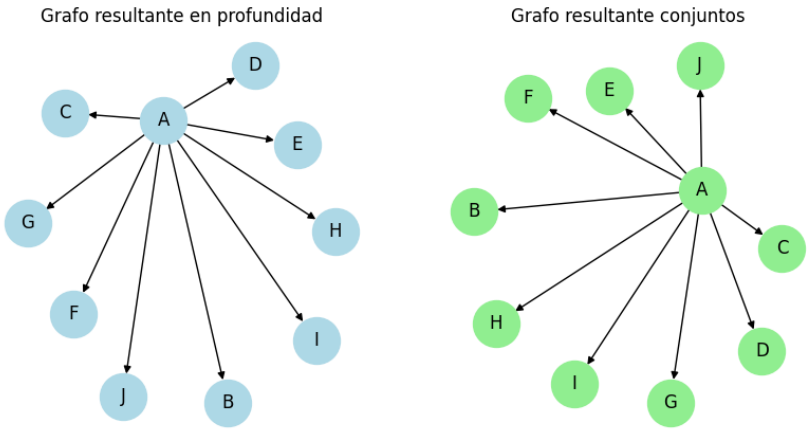

Figura 9: Ejemplo de árbol de Chow-Liu.

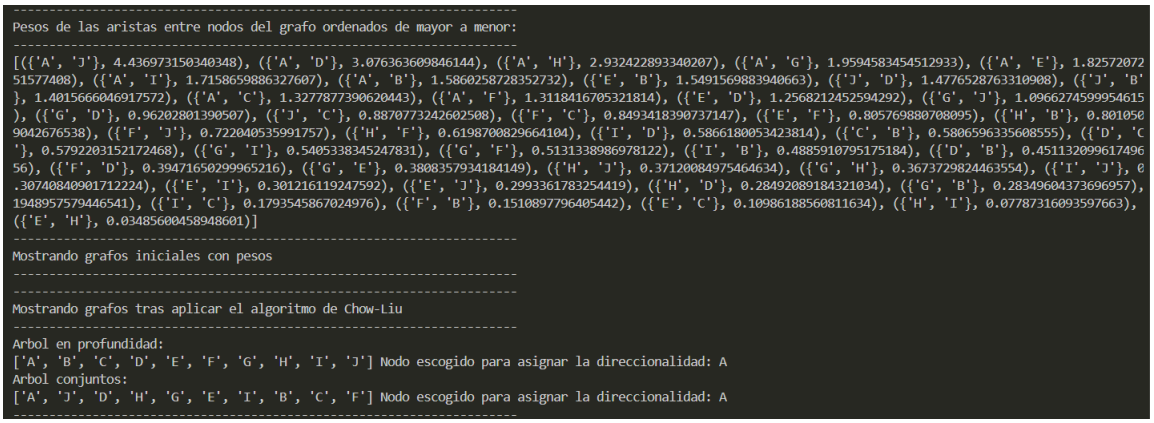

Figura 10: Ejemplo de output de la terminal, con los pesos de las aristas ordenados de mayor a menor.

Así pudimos comprobar, inicialmente, que nuestra implementación original era eficaz; y después, que la implementación con conjuntos también produce el resultado correcto, pero de forma más eficiente, como ya se explicó en el apartado anterior.