Teoría avanzada de la computación

Algoritmos aplicados a Optimización Combinatoria
Entrega Final

Grado en Ingeniería Informática - 4.º Curso

Campus de Leganés Grupo 83

Carlos Gallego Jimenez

Carlos Rubio Olivares

Jorge Rodríguez Fraile

Índice

1. Introducción

El objetivo de esta práctica es estudiar la complejidad y optimización de problemas de diferentes clases de complejidad. Estos problemas están relacionados con el uso de grafos:

- **Conteo de los clústers de un grafo:** Contar el número de subgrafos encontrados en un grafo G.
- **Problema de k-colorabilidad:** Dado un grafo G y un número de colores k, asignar a cada nodo del grafo un color de manera que sus nodos adyacentes tengan un color distinto.
- **Problema k-clique:** Conteo de los subgrafos completos de k nodos encontrados en un grafo G.
- **Problema max-clique:** Dado un grafo G, determinar el valor máximo de k que permite hallar una solución al problema k-clique.

Para cada uno de estos problemas se hará un estudio tanto analítico como empírico para conocer sus tendencias y comportamiento. Además, será necesario establecer los peores casos que se pueden encontrar en los algoritmos que los resuelven y por qué son estos. Cualquier punto o idea que se establezca durante este trabajo será argumentado y estará apoyado en la bibliografía utilizada.

2. Ejercicio 1

a. Estudio analítico

El estudio analítico de este problema se hará explicando el proceso realizado en la función de propagación de marca del código. Esta función se llama desde la función principal del programa, *contar_clusters*. Esto implica que se deba comprobar si un nodo no ha sido marcado y en dicho caso, llamar a *propagar_marca*.

En una situación extrema (grafo completamente inconexo, por ejemplo), el programa hará tantas llamadas de *propagar_marca* como nodos haya. Esto se deriva en ejecutar el bucle que encontramos en *propagar_marca* tantas veces como nodos encontremos en el grafo, aunque en este caso, como no se encuentran conexiones, no se efectúa ningún tipo de recursividad.

Podemos ver, por tanto, dos procesos anidados en los que se recorren todos los nodos, por lo que cada uno tiene complejidad *O(n),* lo que da como resultado una complejidad total de $O(n^2)$, qué es lo que debemos esperar ver en los estudios empíricos que se realicen en secciones posteriores. Este tipo de análisis podemos extrapolarlo a cualquier tipo de grafo donde existiera la recursividad, ya que la complejidad se mantiene constante para diferentes grafos.

A continuación, se presenta la justificación de estos razonamientos mediante diferencias finitas, en la que se cuentan las operaciones elementales hechas. Se ejecutan sucesivas llamadas de *contar_clusters*, desde 1 nodo hasta 10 nodos, en primer lugar para uno completamente conexo y después con uno totalmente inconexo.

Nodos (n)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operaciones Elementales	**25**	**55**	**97**	**151**	**217**	**295**	**385**	**487**	**601**	**727**
Dif. 1		30	42	54	66	78	90	102	114	126
Dif. 2			12	12	12	12	12	12	12	12
Dif. 3				0	0	0	0	0	0	0

Tabla 1. Diferencias finitas para el algoritmo base de conteo de clústers para grafos completos desde 1 hasta 10 nodos.

Nodos (n)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operaciones Elementales	**25**	**57**	**101**	**157**	**225**	**305**	**397**	**501**	**617**	**745**
Dif. 1		32	44	56	68	80	92	104	116	128
Dif. 2			12	12	12	12	12	12	12	12
Dif. 3				0	0	0	0	0	0	0

Tabla 2. Diferencias finitas para el algoritmo base de conteo de clústers para grafos completamente inconexos desde 1 hasta 10 nodos.

Se puede apreciar en ambas tablas que las ecuaciones asociadas se corresponderán a una ecuación de orden 2 ($T(n) = xn^2 + yn + z$), dado que en Dif. 3 se vuelven 0 las diferencias. Sus resoluciones son las siguientes:

$$\left(\begin{array}{ccc|c} 1 & 1 & 1 & 25 \\ 4 & 2 & 1 & 55 \\ 9 & 3 & 1 & 97 \end{array}\right);\left(\begin{array}{ccc|c} 1 & 1 & 1 & 25 \\ 3 & 1 & 0 & 30 \\ 8 & 2 & 0 & 72 \end{array}\right);\left(\begin{array}{ccc|c} 1 & 1 & 1 & 25 \\ 3 & 1 & 0 & 30 \\ 2 & 0 & 0 & 12 \end{array}\right);\left(\begin{array}{ccc|c} 0 & 1 & 1 & 19 \\ 0 & 1 & 0 & 12 \\ 1 & 0 & 0 & 6 \end{array}\right);\left(\begin{array}{ccc|c} 0 & 0 & 1 & 7 \\ 0 & 1 & 0 & 12 \\ 1 & 0 & 0 & 6 \end{array}\right);\begin{array}{r} x=6 \\ y=11 \\ z=7 \end{array}$$

$$\left(\begin{array}{ccc|c} 1 & 1 & 1 & 25 \\ 4 & 2 & 1 & 57 \\ 9 & 3 & 1 & 101 \end{array}\right);\left(\begin{array}{ccc|c} 1 & 1 & 1 & 25 \\ 3 & 1 & 0 & 32 \\ 8 & 2 & 0 & 76 \end{array}\right);\left(\begin{array}{ccc|c} 1 & 1 & 1 & 25 \\ 3 & 1 & 0 & 32 \\ 2 & 0 & 0 & 12 \end{array}\right);\left(\begin{array}{ccc|c} 0 & 1 & 1 & 19 \\ 0 & 1 & 0 & 14 \\ 1 & 0 & 0 & 6 \end{array}\right);\left(\begin{array}{ccc|c} 0 & 0 & 1 & 5 \\ 0 & 1 & 0 & 14 \\ 1 & 0 & 0 & 6 \end{array}\right);\begin{array}{r} x=6 \\ y=14 \\ z=5 \end{array}$$

Que dan lugar a las ecuaciones: $T(n) = 6n^2 + 12n + 7$ y $T(n) = 6n^2 + 14n + 5$. Ambas T(n) serán menores o iguales que $c * g(n)$, para $\forall n \geq n_0$ con $g(n) = n^2$, $n_0 = 1$ y $c = 20$.

Como se cumple dicha condición, la complejidad queda demostrada ser $O(g(n)) = O(n^2)$, para ambos casos extremos.

b. Estudio empírico

Durante esta sección se expondrán las gráficas relacionadas con el estudio empírico del problema de conteo de clústers. Los gráficos mostrarán en el eje horizontal el n.º de nodos del grafo, mientras que en el eje vertical se mostrará el tiempo de ejecución asociado.

Para ejecutar estas pruebas, se ha realizado una modificación del código. Se crea una función denominada *prueba_empírico* a la que se le pasa un n.º de nodos. Hecho esto, se genera un grafo con $\alpha = n/2$ aristas, siendo n el número máximo de arcos ($n(n-1)/2$), ya que según el anexo II del enunciado en este punto hay un cambio de fase considerable en la ejecución de este algoritmo. Una vez creado el grafo con estas características se

llama a la función *contar_clusters* y se inicia una variable del tipo *clock_t* para medir el tiempo pertinente a la ejecución.

Los gráficos mostrarán una serie de ejecuciones utilizando este método, yendo desde 0 a 20.000, de 10 en 10 nodos. Junto a esta gráfica se mostrará su conversión en escala logarítmica para poder estudiar su comportamiento de manera más precisa.

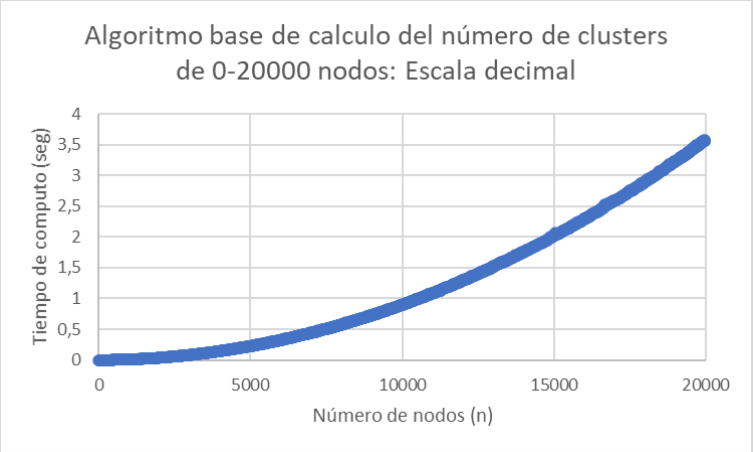

Figura 1. Tiempo de cómputo del algoritmo base en el problema de conteo de clústers en escala decimal.

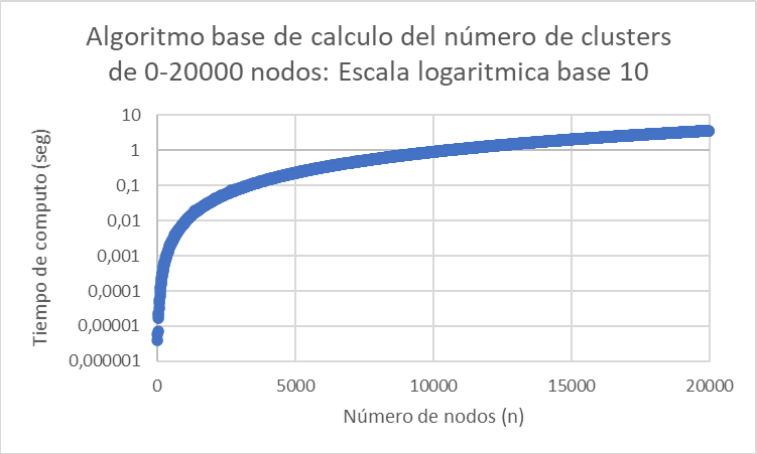

Figura 2. Tiempo de cómputo del algoritmo base en el problema de conteo de clústers en escala logarítmica.

Podemos ver una clara tendencia polinómica (aproximadamente de n^2) en la figura 1. Esta idea puede ser argumentada debido a que al pasar los resultados a escala logarítmica, en la figura 2, se sigue una tendencia logarítmica. Todo este planteamiento empírico es compatible con lo que se ha planteado en el estudio analítico del algoritmo.

c. Circunstancias para el peor caso

Para estudiar los peores casos que se pueden dar en este problema, se opta por un método parecido al de la sección anterior, mediante casos empíricos. De esta manera, se vuelve a modificar el comportamiento del código.

En este caso, se crea un grafo con el número máximo de aristas posibles, es decir, cada nodo tendrá una arista al resto de los nodos creando así un grafo completo. Una vez hecho esto se llama a la función *contar_clusters* para calcular su tiempo de ejecución. Hecho esto, se eliminará una arista y se repetirá el proceso explicado anteriormente. La función principalmente parte de un grafo completo hasta uno completamente inconexo y mide el tiempo de ejecución de cada iteración.

En las gráficas mostradas en esta sección aparecerán en el eje horizontal el número de arcos o aristas y en el eje y el tiempo de cómputo. Se espera que las gráficas apoyen lo descrito en el anexo II y que el cambio de fase se encuentre cerca de los valores de $\alpha = n/2$ con n como el n.º máximo de aristas.

Cabe recalcar que se realizan pruebas sobre grafos con diferentes tamaños de nodos para tener un conjunto de datos de mayor calidad.

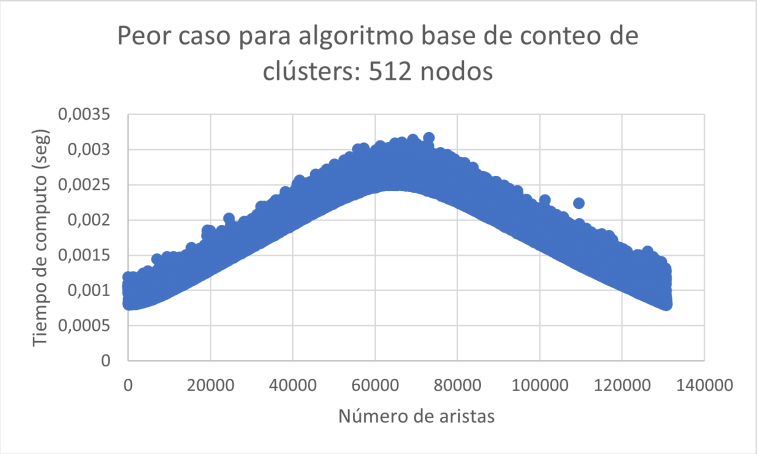

Figura 3. Estudio empírico del peor caso en el problema de conteo de clústers para 512 nodos.

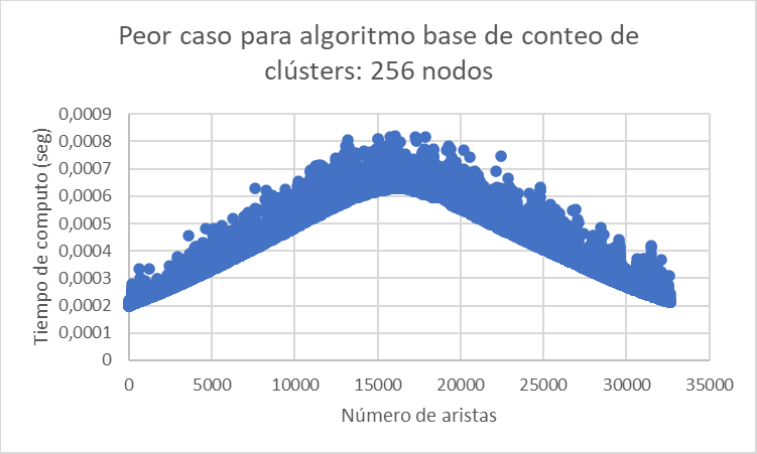

Figura 4. Estudio empírico del peor caso en el problema de conteo de clústers para 256 nodos.

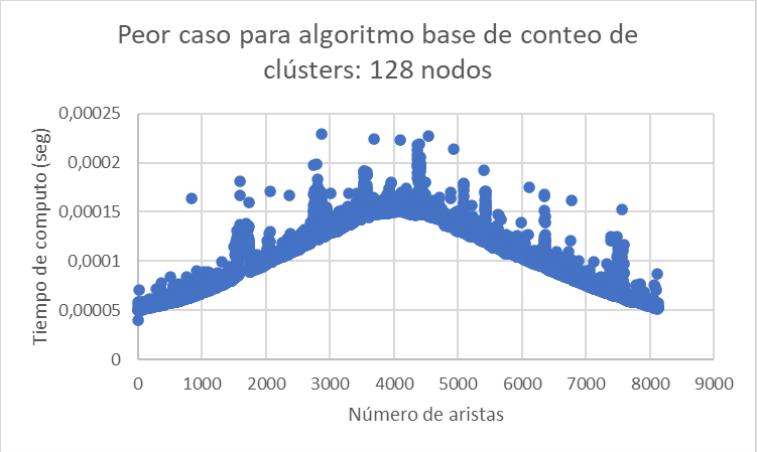

Figura 5. Estudio empírico del peor caso en el problema de conteo de clústers para 128 nodos.

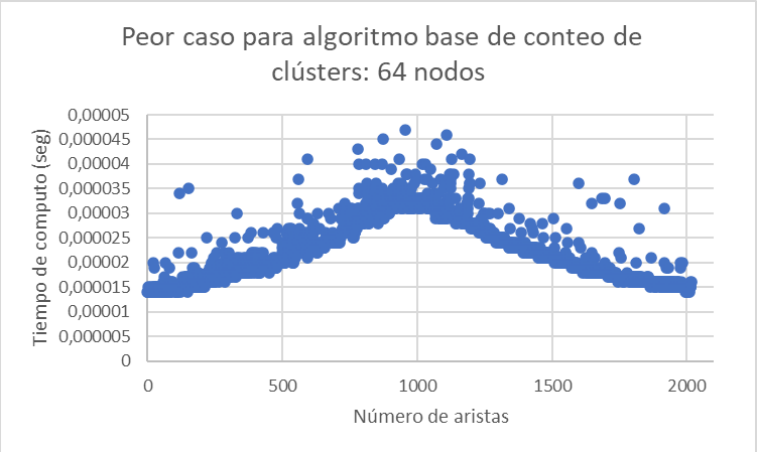

Figura 6. Estudio empírico del peor caso en el problema de conteo de clústers para 64 nodos.

Podemos ver que tal y como se había comentado, hay un aumento considerable del tiempo de cómputo en valores cercanos a $\alpha = n/2$. Esto se puede ver de manera mucho más clara en el grafo de 256 y 512 nodos, ya que valores cercanos al epicentro de la gráfica tienden a tener un tiempo de ejecución mayor. Las pruebas empíricas realizadas en esta sección apoyan, por tanto, a lo planteado en el anexo II.

d. Mejora de la eficiencia

Para empezar, se estudió la posibilidad de, en lugar de hacer una búsqueda en profundidad, emplear el algoritmo de Warshall que se menciona en el enunciado.

Se implementó y probó esta alternativa a pesar de tener una clara complejidad de $O(n^3)$, siendo n el número de nodos del grafo. Este algoritmo lo que permitía era de una manera rápida saber desde un mismo nodo todos los nodos a los que se podría conectar e identificar más directamente los clústers, pero el elevado coste hizo que se desechara esta alternativa.

A continuación se muestra una comparativa de Warshall y el algoritmo dado para los 1700 primeros valores de nodos, con las aristas del peor caso. La primera representa el tiempo

obtenido y en la segunda se aplica el logaritmo base 10 para ver claramente que es de un orden mayor de complejidad polinómica.

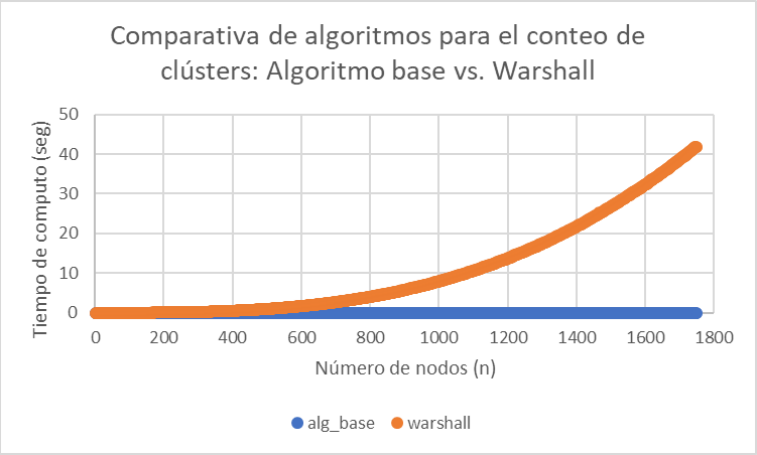

Figura 7. Estudio de mejora para el algoritmo de conteo de clústers base mediante el Algoritmo de Warshall.

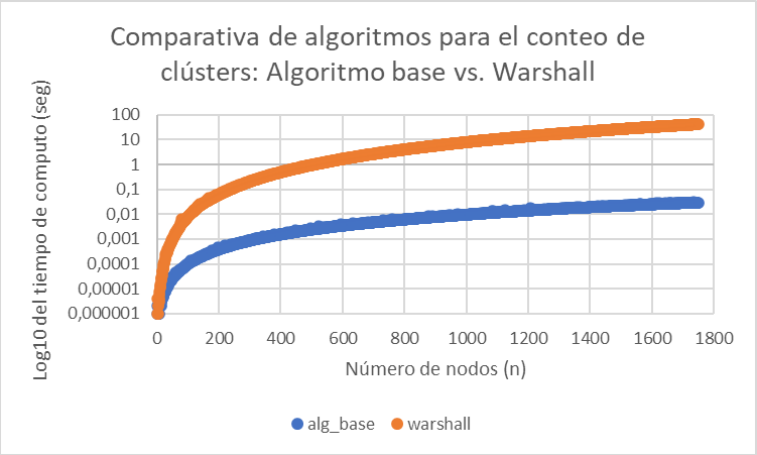

Figura 8. Estudio de mejora para el algoritmo de conteo de clústers base mediante el Algoritmo de Warshall empleando escala logarítmica.

Como se puede ver, los resultados son claramente peores, por lo que se pasa a probar la búsqueda en amplitud, visto que el algoritmo base que lo hace en profundidad da buenos resultados, es otra buena aproximación al problema.

Se ha implementado BFS (Breadth-first search) mediante una cola de nodos, en la que se van introduciendo sucesivamente los nodos con los que está conectado el nodo y sucesivamente se van sacando nodos de esta cola para ir visitándolos y marcándolo como conectados a dicho clúster. Una vez hemos visitado todos los nodos que hemos podido incluir en la cola, se pasa a revisar si queda algún nodo más por visitar. Si es así, este pertenecerá a otro clúster y se repite el proceso de ir metiendo nodos y visitándolos. Cuando se han marcado todos los nodos como visitados, nuestro contador de nodos nos devolverá cuántas veces hemos tenido que elegir un nuevo nodo para hacer esta expansión en amplitud.

Al igual que pasaba con la búsqueda en profundidad, la búsqueda en amplitud, tiene una complejidad de $O(n^2)$, con n siendo el número de nodos del grafo.

Se muestra a continuación unas gráficas de comparación entre el algoritmo base y BFS para una serie de pruebas, tiempo de cómputo para sucesivos números de nodos para el peor caso, van desde 0 hasta 20000 nodos con saltos de 10 en 10. Primero tiempo real y después, el logaritmo base 10.

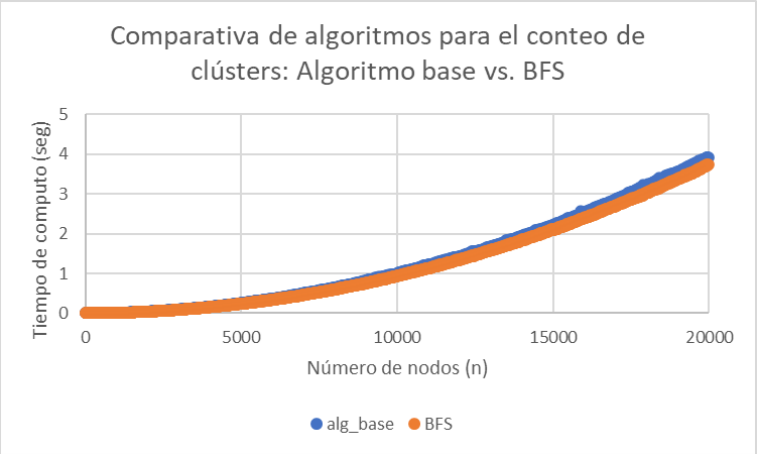

Figura 9. Estudio de mejora para el algoritmo de conteo de clústers base mediante búsqueda en profundidad.

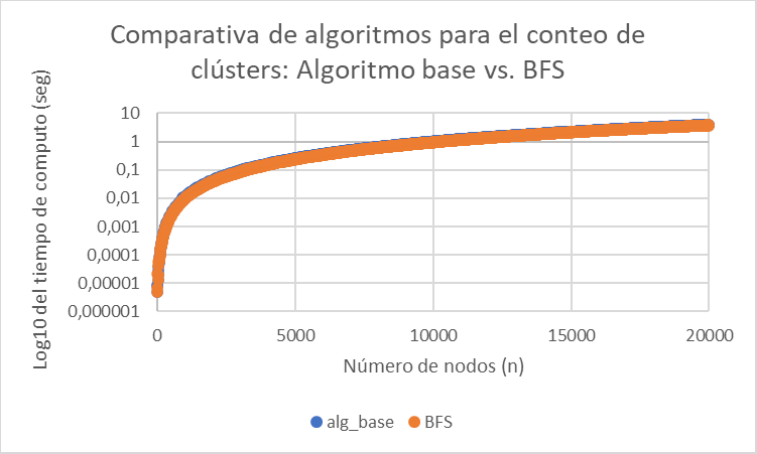

Figura 10. Estudio de mejora para el algoritmo de conteo de clústers base mediante búsqueda en profundidad empleando escala logarítmica.

Como podemos ver en la gráfica, los resultados son prácticamente idénticos, la mejora que supone la búsqueda en amplitud es de un 6,27 % de media sobre el algoritmo base. Para ver mejor esta diferencia en la siguiente gráfica se muestra cuántos segundos de diferencia hay del algoritmo base respecto a BFS.

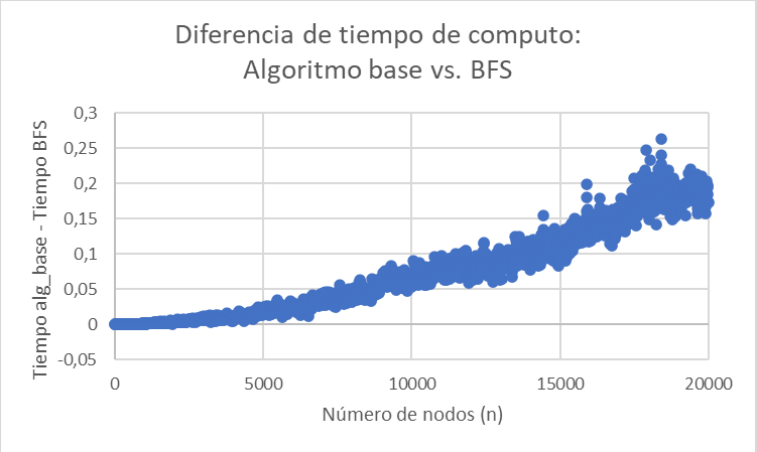

Figura 11. Mejora de tiempos de búsqueda en profundidad respecto al algoritmo base de conteo de clústers.

Esta similitud ya se podía intuir desde un principio conociendo la complejidad de ambos algoritmos, aunque se refuerza con la gráfica logarítmica al tomar la misma forma. La implementación de este último algoritmo es algo mejor (≈6 %), pero nos hace ver que la implementación del algoritmo base proporciona resultados acordes a otros de su misma complejidad.

De todos los probados, los más destacables son el algoritmo base y BFS, aunque el de Warshall se mueve en el rango de las complejidades esperadas, entre cuadrado y cúbico.

3. Ejercicio 2. K-Colorabilidad k=3

a. Método de búsqueda exhaustiva

Para el ejercicio 2 el método de búsqueda exhaustiva que se utiliza es el implementado en el código *grafos2*, en concreto *explora_k_colorabilidad(*k, arcos*)*. Este código toma como entradas el número de colores k que se empleará para colorear los nodos y el número de arcos de media por nodo.

Este método de búsqueda utiliza *asigna_colores* que es una función recursiva que genera todas las asignaciones posibles de colores llamando a la función con el índice del nodo siguiente para cada uno de los colores, guardando esta configuración en una matriz y la asignación del color en un array. Cuando se llega al último nodo la función, se comprueba si hay conflictos entre ellos por medio de la función *comprueba_conflictos()* que comprueba si los nodos adyacentes tienen el mismo color, por medio de comprobaciones en la matriz y el array de asignaciones, y devuelve el número de conflictos en una configuración determinada, esto se realiza para todas las configuraciones, ya que asignamos todos los colores posibles a cada uno de los nodos.

Para obtener los tiempos que se tardarían en obtener todas las configuraciones posibles se llama a esta función con diferente número de nodos.

b. Tamaño grafo para tiempo razonable

Para determinar el número de nodos que debe tener un grafo para poder encontrar todas las soluciones de su colorabilidad para k=3 utilizamos la función de *explora_k_colorabilidad* explicada en el apartado anterior y se grafican los resultados obtenidos teniendo en cuenta que el tiempo razonable asumido es una hora. En la gráfica se muestra en el eje x el número de nodos de un grafo y en el eje y el tiempo de cómputo en segundos. Se añade también una línea que delimita la una hora de ejecución, para entender la magnitud de los cómputos que se muestran.

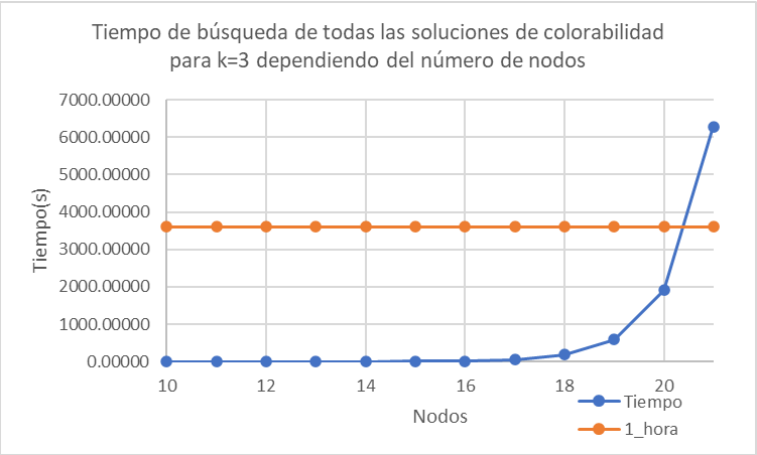

Figura 12. Tiempo de búsqueda de todas las soluciones colorabilidad con k=3, de 10 a 21 nodos.

Se puede observar cómo hay un aumento considerable de tiempo de cómputo para grafos con nodos mayores a 20 nodos. De esta manera, el dominio aceptable para este problema de k-colorabilidad debería encontrarse entre los 1 y 20 nodos, siendo 20 nodos el máximo número de nodos para los cuales se puede obtener todas las configuraciones en un tiempo razonable.

c. Estudio analítico: Búsqueda exhaustiva

A continuación se hará un estudio analítico de la complejidad del problema. Se hará en dos partes. En primer lugar, se verá la complejidad del proceso de asignación de colores, sin tener en cuenta la comprobación de conflictos. Después se realiza el estudio sobre el proceso de comprobar conflictos. Una vez se tenga la complejidad de ambos procesos, se procede a calcular la complejidad del proceso completo. Es importante recalcar que el estudio se realiza utilizando diferencias finitas sobre un grafo completo e incompleto.

Nodos (n)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operaciones Elementales	**36**	**129**	**408**	**1245**	**3756**	**11289**	**33888**	**101685**	**305076**	**915249**
Dif. 1		93	279	837	2511	7533	22599	67797	203391	610173
Div. 1			3	3	3	3	3	3	3	3
Dif. 2				0	0	0	0	0	0	0

Tabla 3. Diferencias finitas sobre la función de asigna_colores sin contar comprobar_conflictos para un grafo completo.

Podemos ver que hay un patrón al dividir la fila 'Dif.1', esto implica que hay una tendencia exponencial de base 3, dado que sale 3 y los pasos de N van de uno en uno. Por otro lado, como sale en el segundo nivel de las diferencias, se puede ver que tendrá asociada una constante, lo que da como resultado la siguiente expresión: $T(n) = x3^n + y$. La resolvemos con los casos conocidos.

$$\left(\begin{array}{cc|c} 3 & 1 & 36 \\ 9 & 1 & 129 \end{array}\right); \left(\begin{array}{cc|c} 3 & 1 & 36 \\ 0 & -2 & 21 \end{array}\right); \left(\begin{array}{cc|c} 3 & 0 & 93/2 \\ 0 & 1 & -21/2 \end{array}\right); \left(\begin{array}{cc|c} 1 & 0 & 31/2 \\ 0 & 1 & -21/2 \end{array}\right); \begin{array}{r} x = 31/2 \\ y = -21/2 \end{array}$$

La complejidad del proceso es, por tanto, de: $T(n) = 31/2 * 3^n - 21/2$. Se puede comprobar fácilmente que salen los resultados exactos.

Nodos (n)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operaciones Elementales	36	129	408	1245	3756	11289	33888	101685	305076	915249
T(n)	36	129	408	1245	3756	11289	33888	101685	305076	915249

Tabla 4. Comprobación de la corrección del coste de asigna_colores sin comprueba_conflictos para un grafo completo.

Se muestra a continuación la tabla de diferencias finitas asociadas al proceso de comprobación de conflictos.

Nodos (n)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operaciones Elementales	**4**	**18**	**39**	**67**	**102**	**144**	**193**	**249**	**312**	**382**
Dif. 1		14	21	28	35	42	49	56	63	70
Dif. 2			7	7	7	7	7	7	7	7
Dif. 3				0	0	0	0	0	0	0

Tabla 5. Diferencias finitas sobre la función de comprueba_conflictos para un grafo completo.

En este caso se puede ver que se seguirá una tendencia polinómica de grado 2. Resolvemos la ecuación $T(n) = xn^2 + yn + z$:

$$\left(\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 4 & 2 & 1 & 18 \\ 9 & 3 & 1 & 39 \end{array}\right); \left(\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 3 & 1 & 0 & 14 \\ 8 & 2 & 0 & 35 \end{array}\right); \left(\begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 3 & 1 & 0 & 14 \\ 2 & 0 & 0 & 7 \end{array}\right); \left(\begin{array}{ccc|c} 0 & 1 & 1 & 1/2 \\ 0 & 1 & 0 & 7/2 \\ 1 & 0 & 0 & 7/2 \end{array}\right); \left(\begin{array}{ccc|c} 0 & 0 & 1 & -3 \\ 0 & 1 & 0 & 7/2 \\ 1 & 0 & 0 & 7/2 \end{array}\right); \begin{array}{r} x = 7/2 \\ y = 7/2 \\ z = -3 \end{array}$$

La complejidad total de este proceso es de $T(n) = 7/2n^2 + 7/2n - 3$, que podemos verificar en la siguiente tabla.

Nodos (n)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Operaciones Elementales	4	18	39	67	102	144	193	249	312	382
T(n)	4	18	39	67	102	144	193	249	312	382

Tabla 6. Comprobación de la corrección del coste de comprueba_conflictos para un grafo completo.

Una vez calculadas las complejidades de ambos procesos, es posible combinar las expresiones obtenidas y realizar el estudio del algoritmo completo. A través del código se puede observar que la llamada al proceso de asignación de colores se efectúa 3^n veces,

con esto podemos hacer la combinación de las ecuaciones. Tendrá la forma: $T(n) = 3^n(xn^2 + yn + z) + k$.

Nodos (n)	1	2	3	4	5	6	7	8	9	10
OE Completo	48	279	1353	6024	25302	101685	394743	1490430	5501388	19929027
OE Inconexo	48	264	1218	5214	21252	83460	318198	1184250	4320408	15500352

Tabla 7. Diferencias finitas sobre la función de asigna_colores completa, para un grafo conexo y para uno inconexo (OE son Operaciones elementales).

$$\left(\begin{array}{cccc|c} 3 & 3 & 3 & 1 & 48 \\ 36 & 18 & 9 & 1 & 278 \\ 243 & 81 & 27 & 1 & 1353 \\ 1296 & 324 & 81 & 1 & 6024 \end{array}\right); \left(\begin{array}{cccc|c} 3 & 3 & 3 & 1 & 48 \\ 11 & 5 & 2 & 0 & 77 \\ 80 & 26 & 8 & 0 & 435 \\ 431 & 107 & 26 & 0 & 1992 \end{array}\right); \left(\begin{array}{cccc|c} 3 & 3 & 3 & 1 & 48 \\ 0 & 978 & 576 & 0 & 11275 \\ 0 & 294 & 152 & 0 & 3125 \\ 431 & 107 & 26 & 0 & 1992 \end{array}\right); \left(\begin{array}{cccc|c} 3 & 3 & 3 & 1 & 48 \\ 0 & 978 & 576 & 0 & 11275 \\ 0 & 0 & 2 & 0 & 25 \\ 431 & 107 & 26 & 0 & 1992 \end{array}\right); \begin{array}{l} x = 17/6 \\ y = 25/6 \\ z = 25/2 \\ k = -21/2 \end{array}$$

Al combinar las fórmulas anteriores nos quedamos con la siguiente expresión $T(n) = 1/6 * 3^n(17n^2 + 25n + 75) - 63/6$ para el grafo conexo, pero si hacemos el mismo proceso para el inconexo tenemos $T(n) = 1/2 * 3^n(4n^2 + 10n + 25) - 21/2$. Ambas tienen el mismo orden de complejidad. Ambas T(n) serán menores o iguales que $c * g(n)$, para $\forall n \geq n_0$ con $g(n) = 3^n n^2$, $n_0 = 8$ y $c = 3$. Como se cumple dicha condición, la complejidad queda demostrada ser $O(g(n)) = O(3^n n^2)$, para ambos casos.

Nodos (n)	1	2	3	4	5	6	7	8	9	10
OE Completo	48	279	1353	6024	25302	101685	394743	1490430	5501388	19929027
T(n) Completo	48	279	1353	6024	25302	101685	394743	1490430	5501388	19929027
OE Inconexo	48	264	1218	5214	21252	83460	318198	1184250	4320408	15500352
T(n) Inconexo	48	264	1218	5214	21252	83460	318198	1184250	4320408	15500352

Tabla 8. Comprobación de la corrección del coste de asigna_colores completo (OE son Operaciones elementales).

d. Estudio empírico: Búsqueda exhaustiva

Debido a que el orden de complejidad es el mismo independientemente de que el grafo sea completo o inconexo, se opta por realizar el estudio empírico sobre grafos inconexos para reducir tiempos de ejecución. Para optimizar las ejecuciones se añade el flag '-O3'. Se mostrarán 4 gráficas. En primer lugar, la búsqueda de 1 a 23 nodos en escala decimal y logarítmica. También se presentarán gráficas asociadas al número de operaciones elementales utilizando testigos, de nuevo, mostrando ambas escalas.

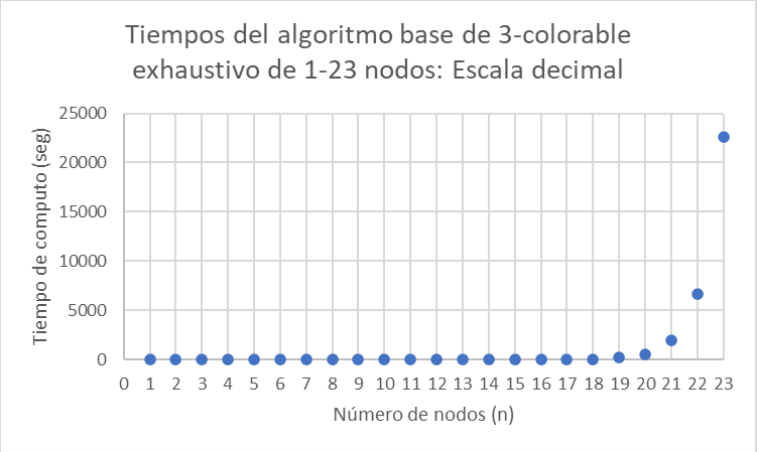

Figura 13. Estudio empírico del algoritmo base en escala decimal. Número de nodos vs. Tiempo de cómputo.

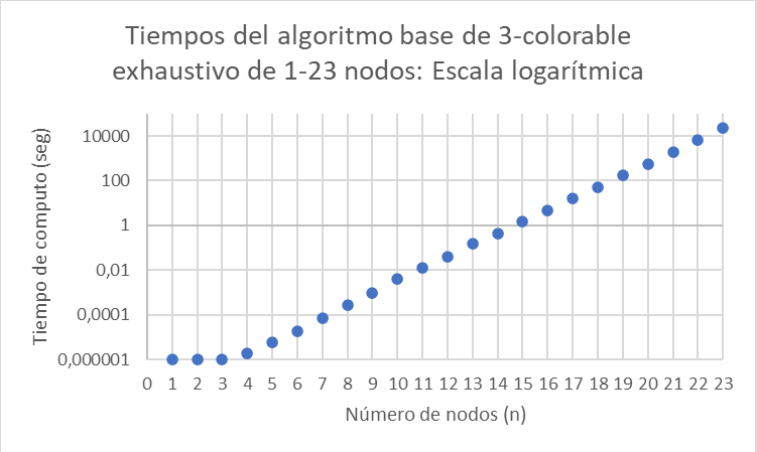

Figura 14. Estudio empírico del algoritmo base en escala logarítmica. Número de nodos vs. Tiempo de cómputo.

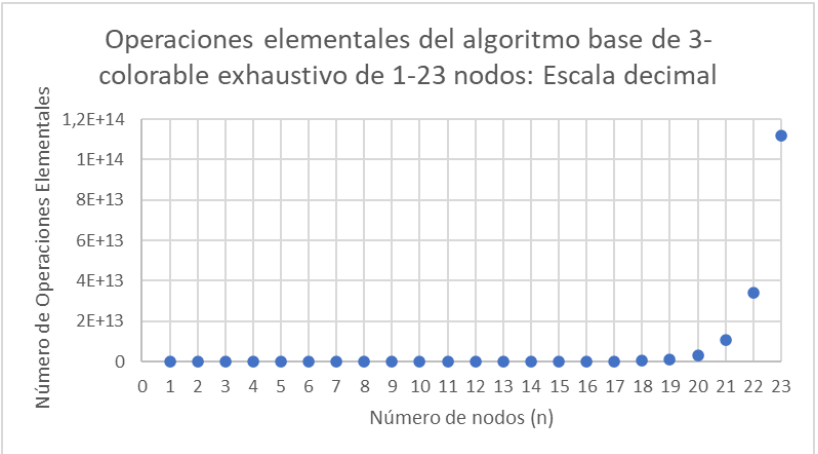

Figura 15. Estudio empírico del algoritmo base en escala decimal. Número de nodos vs. Número de operaciones elementales.

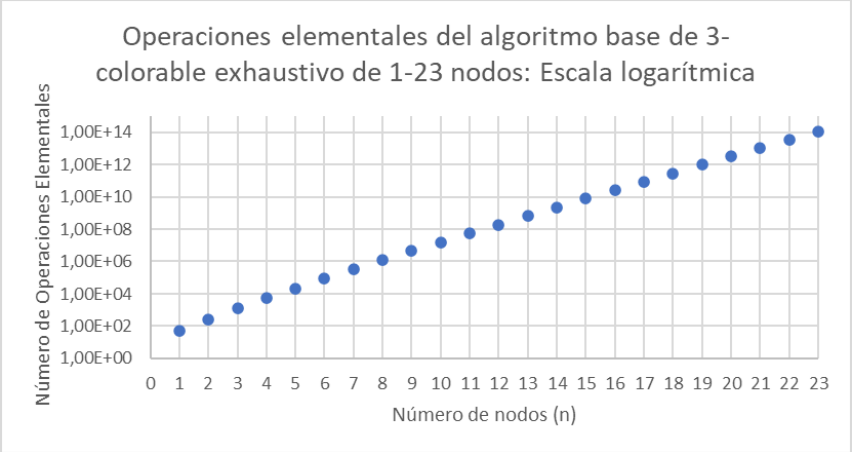

Figura 16. Estudio empírico del algoritmo base en escala logarítmica. Número de nodos vs. Número de operaciones elementales.

En los dos estudios hechos, se puede ver una clara tendencia exponencial. Esto se ve apoyado por las gráficas en escala logarítmica, que muestran una clara tendencia lineal. Se puede ver también lo establecido en apartados anteriores, y es que grafos con valores mayores a 22 nodos tienen un tiempo de ejecución muy elevado.

e. Soluciones según la densidad búsqueda exhaustiva

En el estudio de densidad se creará un grafo de 17 nodos completo y se irán eliminando aristas de manera aleatoria hasta que quede completamente inconexo. De esta manera se mostrarán las soluciones encontradas mientras se decrementan las aristas frente a la densidad del grafo y su completitud. Por último, también se muestra el número de comprobaciones realizadas, tanto las fallidas como las soluciones. En este último gráfico también se mostrarán las verificaciones totales ejecutadas.

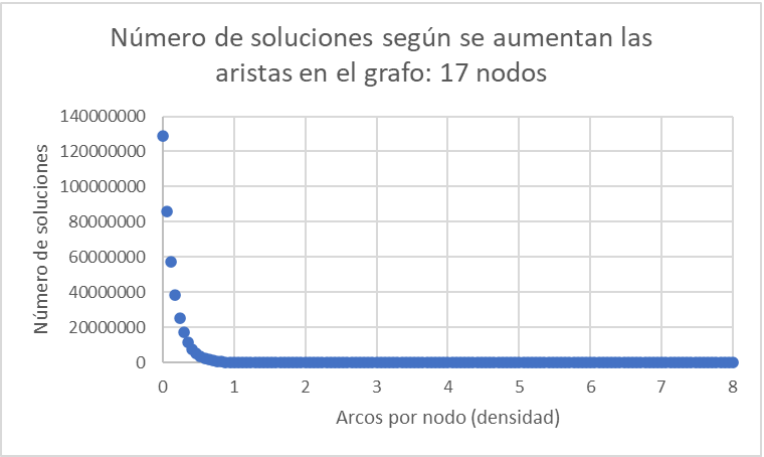

Figura 17. Estudio empírico sobre la densidad del grafo del algoritmo base en escala decimal. Número de soluciones vs. Densidad.

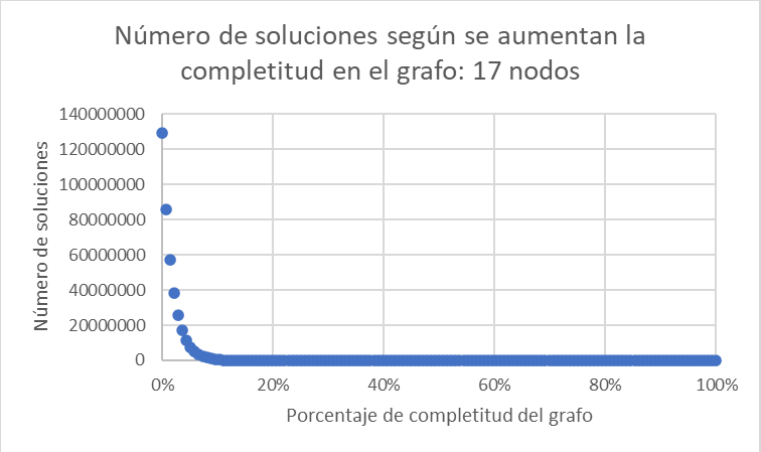

Figura 18. Estudio empírico sobre la densidad del grafo del algoritmo base en escala decimal. Número de soluciones vs. Completitud.

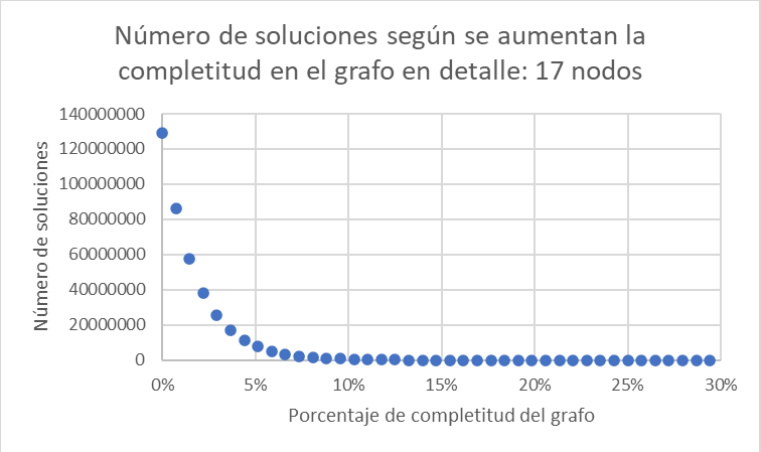

Figura 19. Estudio empírico sobre la densidad del grafo del algoritmo base en escala decimal. Número de soluciones vs. Completitud.

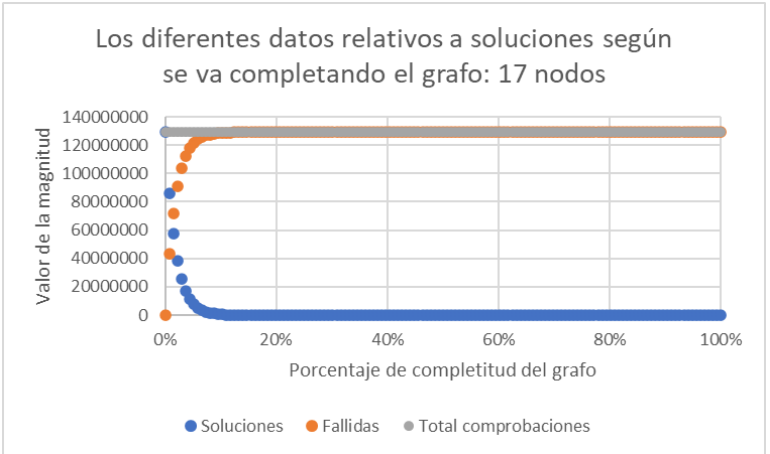

Figura 20. Representación gráfica del número de casos probados en busca de todas las soluciones, según la completitud del grafo de 17 nodos utilizado. Véanse en ella el número de combinaciones fallidas, soluciones y las totales.

Se puede apreciar que el número de soluciones disminuye muy rápidamente según se va poblando el grafo, en los porcentajes se puede ver algo más claro. Entrando en detalle, en cuanto nos acercamos a 2,2 arcos por nodos, se deja de poder encontrar una solución, cosa que se puede ver en la siguiente tabla:

Nodos	Arcos	Densidad	Completitud	Soluciones	Fallidas	Total
17	35	2,058823529	26 %	84	129140079	129140163
17	36	2,117647059	26 %	84	129140079	129140163
17	37	2,176470588	27 %	84	129140079	129140163
17	38	2,235294118	28 %	0	129140163	129140163
17	39	2,294117647	29 %	0	129140163	129140163
17	40	2,352941176	29 %	0	129140163	129140163

Tabla 9. Datos relativos a pruebas realizadas para encontrar todas las soluciones según se incrementaba el número de aristas. Sirva de aclaración para ver el umbral de soluciones.

Este valor es muy cercano al 2,35 que indicaba Hayes en On the Threshold [1], a partir del cual no habría soluciones para 3-colorabilidad. Además, este punto es el peor caso con diferencia, porque está en el límite entre tener o no solución y le cuesta considerablemente más encontrarla.

f. Implementación método de backtracking

En la implementación del *backtracking* para este problema se debe modificar la función *asigna_colores.*

La idea principal es pintar un nodo y visitar sus conexiones de manera que todas ellas tengan una combinación válida. En caso de que haya una combinación inválida, se descoloran los nodos y se prueba otra combinación volviendo al nodo inicial.

En el código se crea una función denominada *comprueba_adyacencia(int nodo, int nodos)*, que devuelve un 1 en caso de que nodo tenga conflicto con alguno de sus adyacentes y 0 en caso contrario. De esta manera, cuando se asigna color a un nodo se hace una comprobación para saber si dicha configuración es parcialmente válida, de modo que si hay un conflicto no avanza la rama y se prueba otro color para ese nodo. En caso de no tener conflicto se llama de nuevo a la función de asignación de colores con el índice del nodo adyacente.

De esta manera nos aseguramos en cierta manera la implementación de la poda con estas funciones, ya que en caso de que sea inválido colorear con k un nodo i, todas las derivaciones de pintar ese nodo con ese color no se visitarán.

g. Coste computacional del método de backtracking para una solución

Para entender el coste computacional del algoritmo nos fijamos en la función *asigna_colores_backtracking* que es la funcionalidad de asignación de colores asociada al backtracking. Esta función incluye una recursividad, por lo que necesitaremos un caso base para desglosar la recursividad.

```
1  void asigna_colores_backtracking(int ind, int nodos, int k) {
2      // T(n) = 1 + max(1 + n^2 + 1 + 2; 2 + 3(3+ n + 7 + T(n-1)))
3      int j;
4      unsigned long long cf;
5      if (ind >= nodos) {
6          cf = comprueba_conflictos(nodos);
7          if (cf == 0) {
8              sin_conflictos++;
9          } else {
10             con_conflictos++;
11         }
12     } else {
13         for (j = 0; j < k; j++) {
14             asignaciones[ind] = j;
15             if (!comprueba_adyacencia(ind, nodos)) {
16                 asigna_colores_backtracking(ind + 1, nodos, k);
17             }
18             if (sin_conflictos >= 1) return;
19             asignaciones[ind] = -1;
20         }
21     }
22 }
```

A continuación se muestran los cálculos realizados para resolver la recurrencia por despliegue y obtener la complejidad.

$$T(1) = 29$$
$$T(n) = 3n + 3T(n-1)$$

$$\begin{aligned} T(n) &= 3n + 3(3(n-1) + 3T(n-2)) \\ &= 3n + 3^2 n - 1 \cdot 3^2 + 3^2 T(n-2) \end{aligned}$$

$$\begin{aligned} T(n) &= 3n + 3^2 n - 1 \cdot 3^2 + 3^2(3(n-2) + 3T(n-3)) \\ &= 3n + 3^2 n + 3^3 n - 1 \cdot 3^2 - 2 \cdot 3^3 + 3^3 T(n-3) \end{aligned}$$

$$\begin{aligned} T(n) &= 3n + 3^2 n + \cdots + 3^{k-1} n + 3^k n - 1 \cdot 3^2 - 2 \cdot 3^3 - \cdots - (k-2)3^{k-1} \\ &\quad - (k-1)3^k + 3^k T(n-k) \end{aligned}$$

$$n - k = 1;\ k = n - 1$$

$$\begin{aligned} T(n) &= 3n + 3^2 n + \cdots + 3^{n-2} n + 3^{n-1} n - 1 \cdot 3^2 - 2 \cdot 3^3 - \cdots - (n-3)3^{n-2} \\ &\quad - (n-2)3^{n-1} + 3^{n-1} T(1) \end{aligned}$$

$$\begin{aligned} T(n) &= n\left(\sum_{k=1}^{n-1} 3^k\right) + \left(\sum_{k=1}^{n-2} -k3^{k+1}\right) + 29 \cdot 3^{n-1} \\ &= n\left(\frac{3^{n-1}3 - 3}{3-1}\right) + \frac{1}{4}(-2 \cdot 3^n n + 5 \cdot 3^n - 9) + 29 \cdot 3^{n-1} \\ &= \frac{3^n}{2} n - \frac{3}{2} n - \frac{3^n}{2} n + \frac{5}{4} 3^n - \frac{9}{4} + 29 \cdot 3^{n-1} \\ &= 3^{n-1}\left(\frac{5}{4}3 + 29\right) + n\left(\frac{3^n}{2} - \frac{3}{2} - \frac{3^n}{2}\right) - \frac{9}{4} \\ &= \frac{131}{4} 3^{n-1} - \frac{3}{2} n - \frac{9}{4} \end{aligned}$$

Por lo tanto, se tiene un orden exponencial de base 3, que da como resultado una complejidad de $O(3^n)$. Lo que supone una mejora con respecto al previo.

h. Comparativa del método exhaustivo y de backtracking

En esta sección se comprobará la eficiencia del método de backtracking frente al algoritmo base. De esta manera se mostrará el tiempo de ejecución frente al número de nodos y frente al número de operaciones base. Las gráficas se mostrarán tanto en escala decimal como logarítmicas. Además, se mostrarán comparativas tanto de los algoritmos buscando todas las soluciones posibles como de una sola.

Comenzando por los de una sola solución:

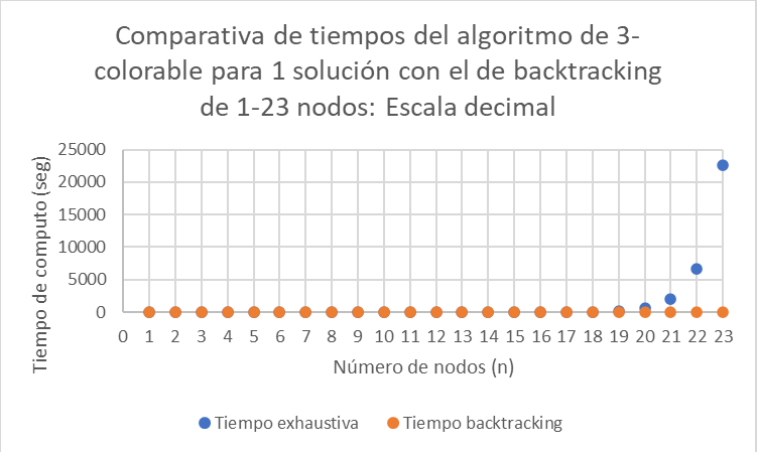

Figura 21. Comparación de tiempos de cómputo entre la búsqueda exhaustiva y la de una sola solución con backtracking para el caso de tiempo razonable.

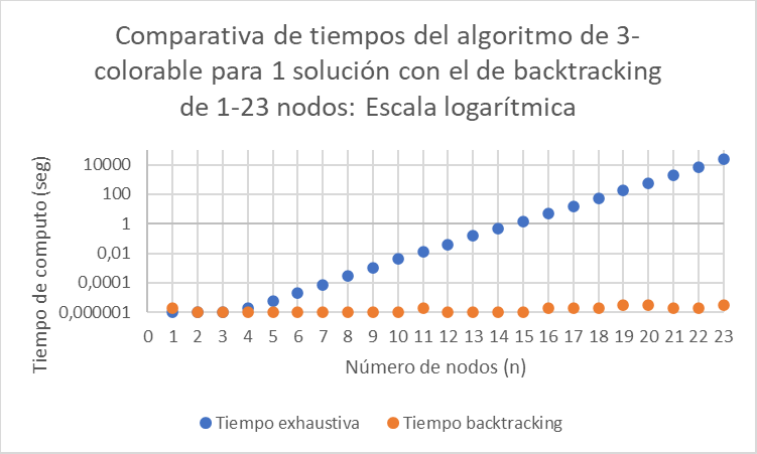

Figura 22. Comparación de tiempos de cómputo en logaritmo 10 entre la búsqueda exhaustiva y la de una sola solución con backtracking para el caso de tiempo razonable.

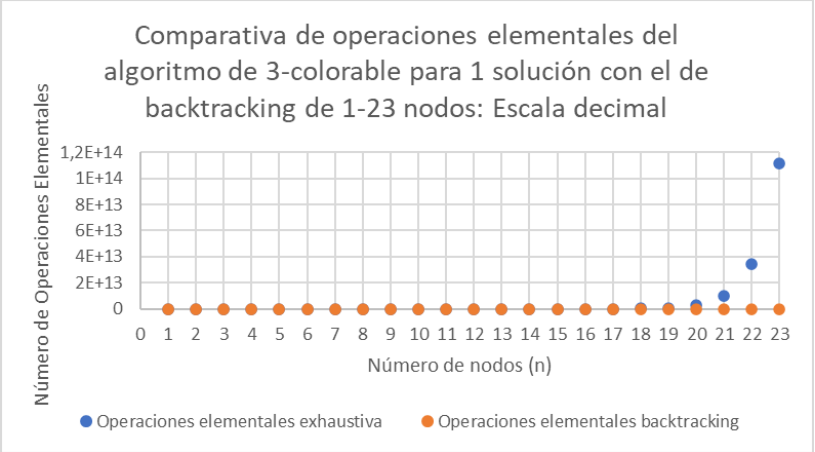

Figura 23. Comparación del número de operaciones elementales realizadas entre la búsqueda exhaustiva y la de una sola solución con backtracking para el caso de tiempo razonable.

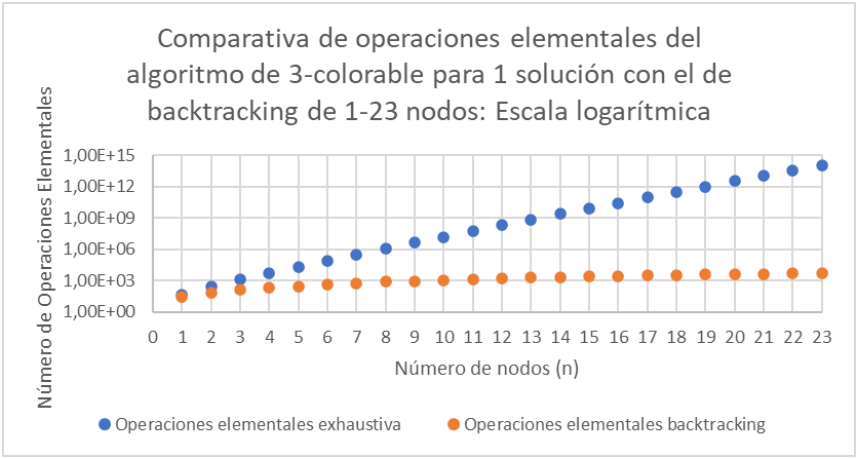

Figura 24. Comparación del logaritmo 10 del número de operaciones elementales realizadas entre la búsqueda exhaustiva y la de una sola solución con backtracking para el caso de tiempo razonable.

Se puede ver de manera clara que el método de backtracking es mucho mejor que el algoritmo base, exactamente un 87,55 % menos de tiempo de ejecución. Esto supone una gran mejora, ya que es posible comprobar grafos con un mayor número de nodos. La mejora es tan grande, puesto que encontrar una sola solución le permite terminar, además, de que corta las ramas de raíz cuando no son válidas.

A continuación se muestra una comparativa para ambos algoritmos cuando se buscan todas las soluciones:

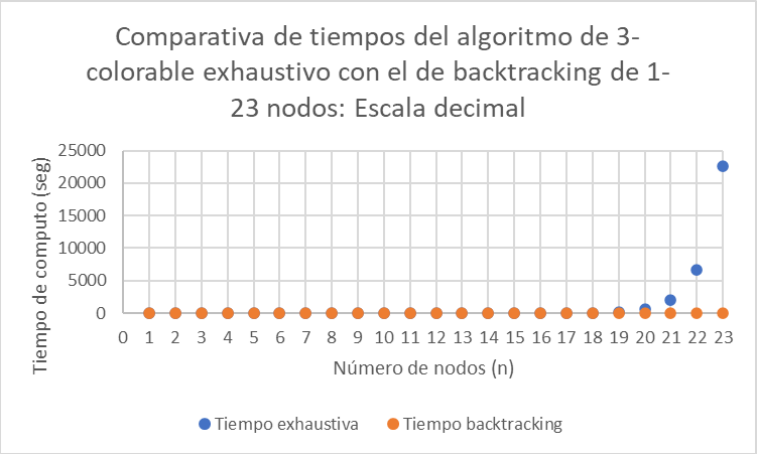

Figura 25. Comparación de tiempos de cómputo entre la búsqueda exhaustiva base y con backtracking para el caso de tiempo razonable.

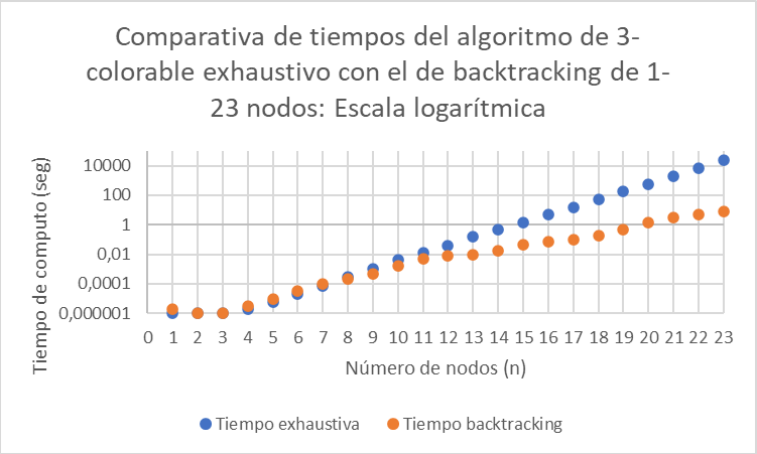

Figura 26. Comparación del logaritmo 10 de los tiempos de cómputo entre la búsqueda exhaustiva base y con backtracking para el caso de tiempo razonable.

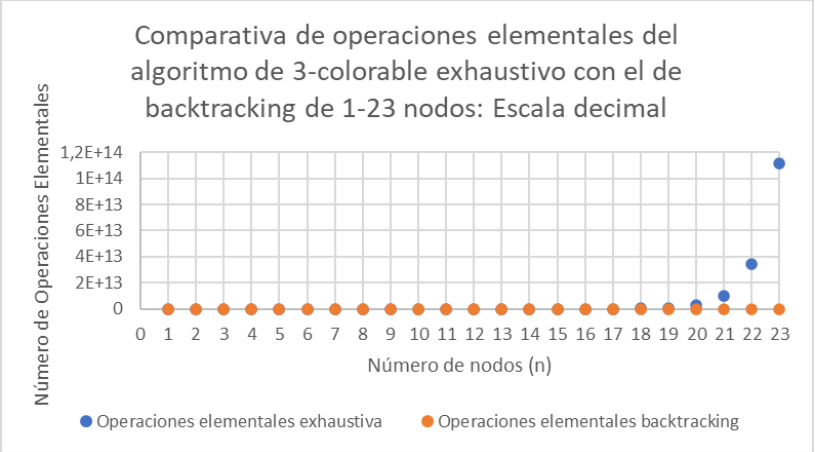

Figura 27. Comparación del número de operaciones elementales entre la búsqueda exhaustiva base y con backtracking para el caso de tiempo razonable.

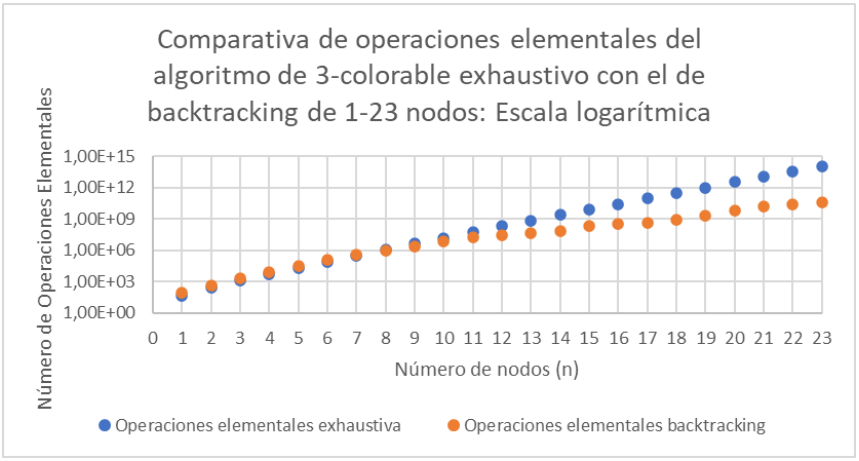

Figura 28. Comparación del logaritmo 10 del número de operaciones elementales entre la búsqueda exhaustiva base y con backtracking para el caso de tiempo razonable.

En este caso, el método backtracking sigue siendo mejor aunque por menos diferencia, en este caso un 41,47 % menos de OE, al inicio el tiempo de ejecución es mayor en el backtracking debido al proceso de mirar conflictos adyacentes. Esta reducción es por el probar a obtener otras posibles soluciones, aun habiendo encontrado una, lo que obliga a recorrer o descartar todos los posibles caminos.

i. Heurística y poda implementada

Para la heurística implementada se sigue lo planteado en el enunciado. La idea de esta heurística es la de colorear primero los nodos con más conexiones para que, en caso de que una solución sea imposible, esta se encuentre de manera más rápida.

Para realizar la implementación, una de las necesidades principales es un array que nos ordene los nodos de más a menos conexiones. Para ello, se utiliza el método de mergesort para dos arrays: Un array de conexiones y otro de índices. Cuando se haga un cambio en el array de conexiones, el mismo cambio se efectuará sobre el array de índices.

Mergesort es un algoritmo de ordenamiento muy conocido, basado en el 'divide y vencerás'. El array a ordenar se divide en 2 mitades, ordena estas dos mitades para después unirlas y crear una lista ordenada. Se escoge este algoritmo, ya que su complejidad es de $O(n * log(n))$

Por otro lado, para utilizar el mergesort se necesita tener un array con el n.º de conexiones de cada nodo, para ello se utiliza la función *contar_conexiones* que recorre la matriz y cuenta para cada nodo su número de conexiones asociadas.

j. Tamaño de grafo para tiempo razonable con poda y heurística

En este apartado se pone a prueba el tiempo de cómputo del algoritmo implementado para determinar la cantidad de nodos que puede tener nuestro grafo para resolver en tiempo razonable la cuestión de si es 3-colorable. Para poner un caso que no sea trivial y conociendo cómo funciona el algoritmo, se le han proporcionado sucesivos grafos desde 1 hasta 100 nodos, en saltos de 3. La peculiaridad de este grafo es que es una cadena de 4-cliques, de esta manera primero pinta los vértices que conectan los cliques para posteriormente darse cuenta de que no es posible llegar a pintar ninguno por completo.

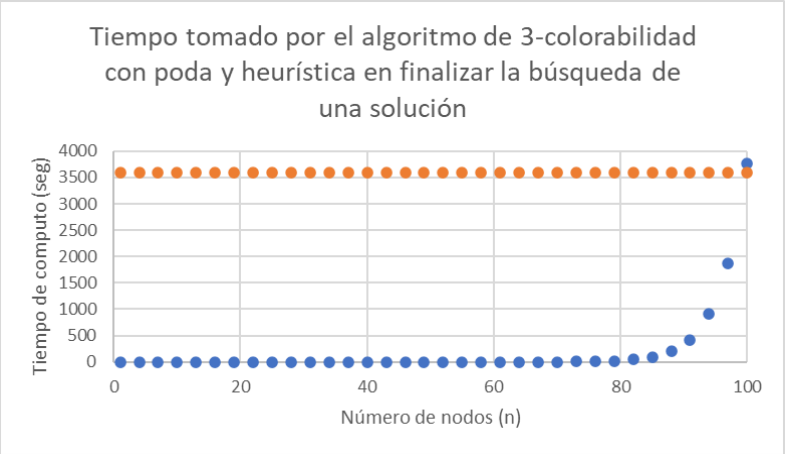

Figura 29. Estimación del número de nodos del grafo para encontrar una solución en tiempo razonable, establecido en 1 hora, siendo los grafos 4-cliques unidos. Representa desde 1 hasta 100 nodos con su tiempo en segundos.

k. Estudio analítico: Backtracking con poda y heurística

Para realizar este estudio se parte del que se ha realizado para la versión que añadía backtracking, de manera que el núcleo del algoritmo ha sufrido muy leves modificaciones que no cambiaron su orden. Lo clave de este nuevo análisis es el coste que supone la heurística que se ha añadido y que deberá ser tenida en cuenta para el rendimiento del algoritmo completo.

Comenzando por el método para contar el número de conexiones por nodo, contar_conexiones(nodos), tenemos el siguiente código:

```
void contar_conexiones(int nodos) {
    // T(n)= 9/2n^2+7/2n+4
    for (int i = 0; i < nodos; i++){
      conexiones[i] = 0;
    }
    for (int i = 0; i < nodos-1; i++) {
        indices[i] = i;
        for (int j = i+1; j < nodos; j++) {
            if (matriz[i][j] == 1 && i != j) {
              conexiones[i]++;
              conexiones[j]++;
            }
        }
    }
    indices[nodos-1] = nodos-1;
}
```

Figura 30. Método contar_conexiones para el conteo de aristas por nodo.

Contamos las operaciones elementales que se ejecutan y tenemos la siguiente ecuación:

$$T(n) = 2 + 3 + 1 + 3 + (n-1)(4 + 4 + p(9)) + 4;\ p = \frac{n}{2}$$

$$T(n) = 9 + (n-1)(8 + 9\frac{n}{2}) + 4$$
$$= \frac{9}{2}n^2 + \frac{7}{2}n + 4$$

Lo que nos deja una complejidad de orden polinómico de grado 2, que sería $O(n^2)$.

Tras hacer el conteo se pasa a ordenar los vectores con el mergesort que como se ha comentado tiene una complejidad de $O(n * log(n))$.

Una vez preparado todo, se pasa al núcleo del problema colorear los vértices en orden más poblado a menos, que se mantiene igual que el calculado previamente, es decir, $O(3^n)$.

En resumen, tenemos una secuencia de 3 operaciones que son $O(n^2 + n * log\, n + 3^n)$, el núcleo del problema es claramente superior a los otros dos, por lo que la complejidad se puede reducir a realizar la asignación de colores. Lo que resulta en: $O(3^n)$.

I. Estudio empírico: Backtracking con poda y heurística

El estudio empírico se hará utilizando una densidad de 2,4 arcos por nodo. Se muestran los tiempos de ejecución frente al número de nodos y el n.º de operaciones elementales, como se ha ido haciendo en estudios empíricos anteriores. Para apoyar las conclusiones sobre las posibles tendencias que se sigan, se emplearán escalas decimales y logarítmicas.

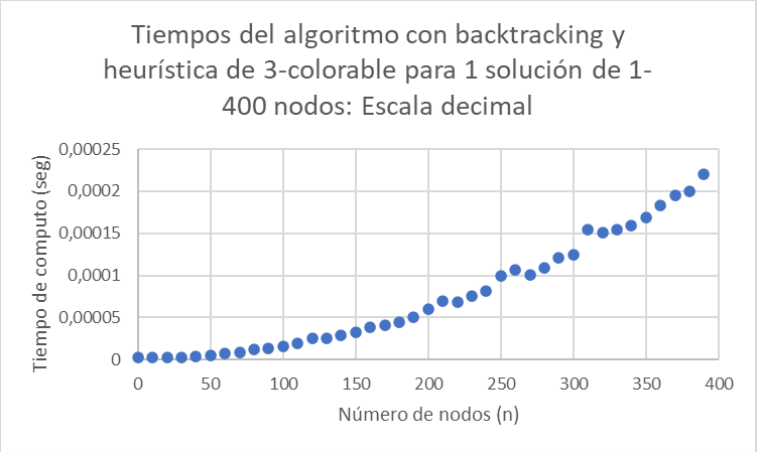

Figura 31. Tiempos para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos que forman el árbol, desde 1 hasta 400 nodos y tiempo en segundos.

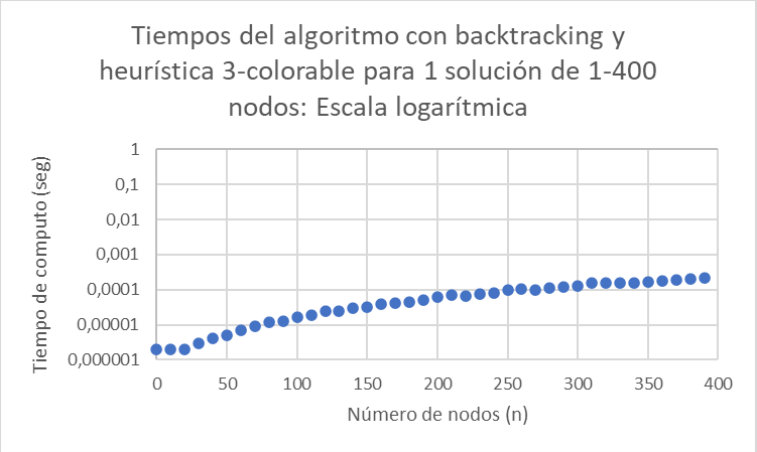

Figura 32. Tiempos para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos que forman el árbol, desde 1 hasta 400 nodos y tiempo en logaritmo base 10 de los segundos.

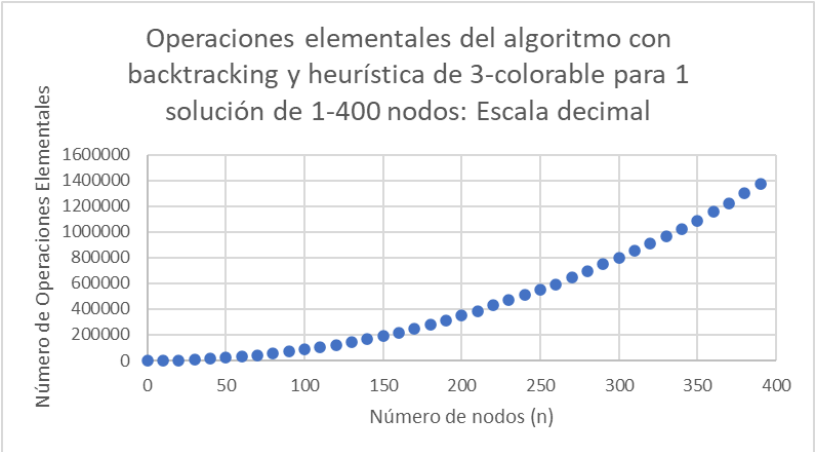

Figura 33. Operaciones elementales para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos que forman el árbol, desde 1 hasta 400 nodos.

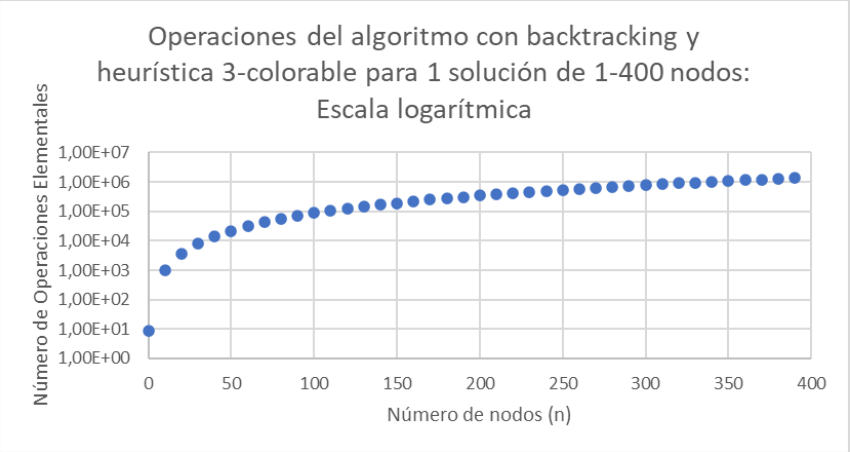

Figura 34. Logaritmo 10 de las operaciones elementales para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos que forman el árbol, desde 1 hasta 400 nodos.

Llegando a 400 nodos, se sobrepasa de manera significativa el tiempo de ejecución necesario para comprobar soluciones, por lo que no se ve de manera clara la tendencia exponencial que está asociada al proceso, para esto, se muestran a continuación el caso expuesto en el de tiempo razonable, que trata de poner el grafo en su peor situación, que haya sucesivos 4 cliques. En esta peor situación se ve claramente su tendencia exponencial, respaldada por la linealidad de su representación logaritmo base 10.

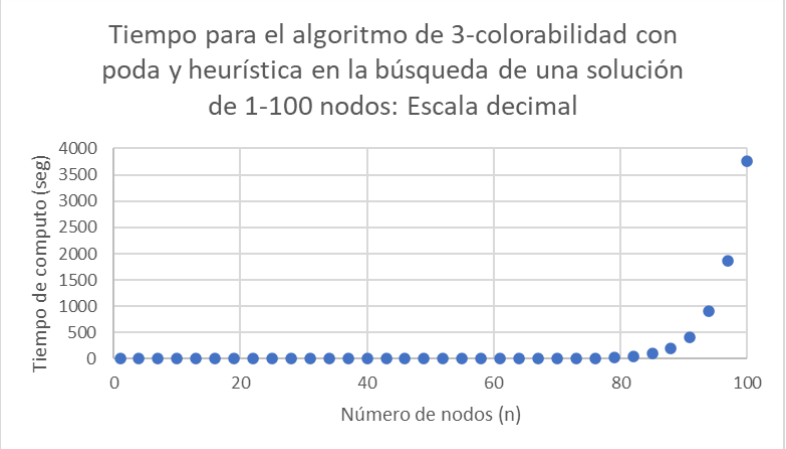

Figura 35. Tiempos para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos, en una estructura de grafo de 4-cliques conectados, desde 1 hasta 100 nodos y tiempo en segundos.

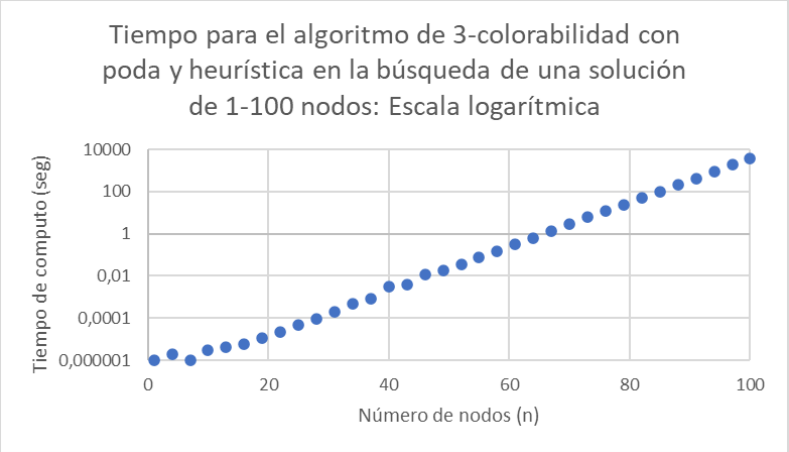

Figura 36. Tiempos para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos, en una estructura de grafo de 4-cliques conectados, desde 1 hasta 100 nodos y tiempo como el logaritmo 10 de los segundos.

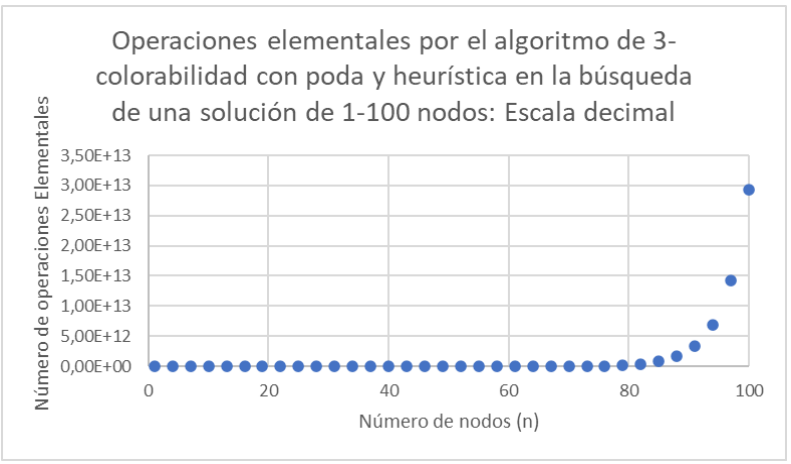

Figura 37. Operaciones elementales para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos, en una estructura de grafo de 4-cliques conectados, desde 1 hasta 100 nodos.

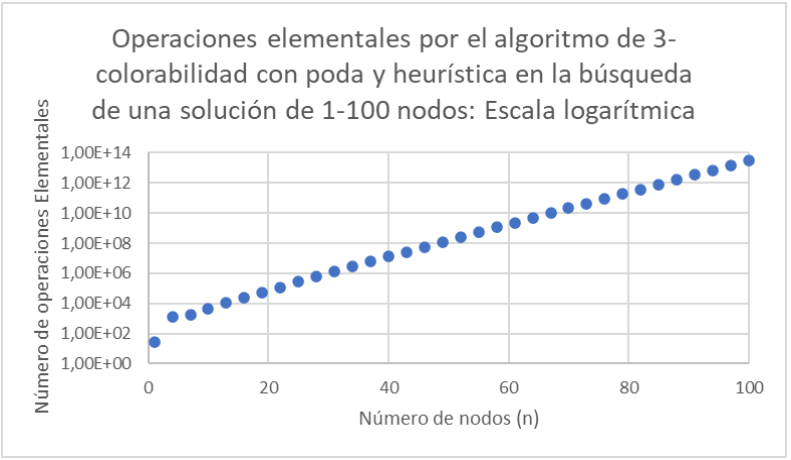

Figura 38. Logaritmo base 10 de las operaciones elementales necesarias para encontrar una configuración de grafo 3-colorable, según se aumenta el número de nodos, en una estructura de grafo de 4-cliques conectados, desde 1 hasta 100 nodos.

m. Influencia de la densidad

En este caso se muestra la importancia de la densidad del grafo dentro del algoritmo de backtracking con poda y heurística implementadas. Como en apartados anteriores, se mostrará el n.º de soluciones frente a la densidad y completitud, por otro lado, también se mostrarán el número de posibles soluciones que se han probado, tanto válidas como inválidas.

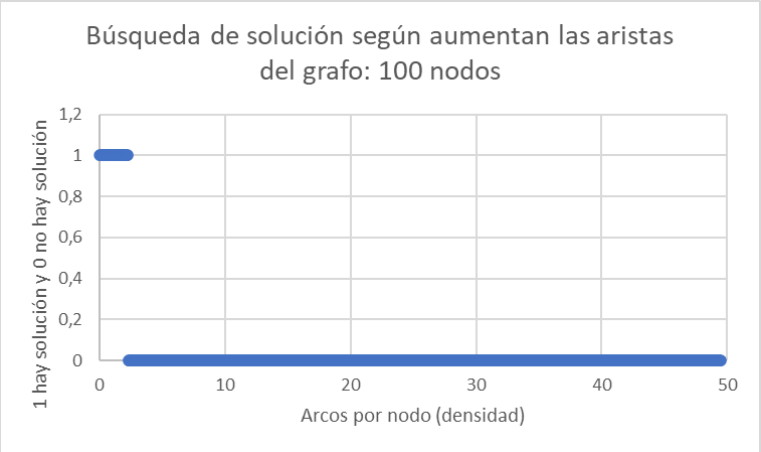

Figura 39. Representación de la existencia de soluciones en sucesivos grafos de 100 nodos según se va aumentando el número aristas que lo componen. 1 denota la existencia de solución y 0 que no existe.

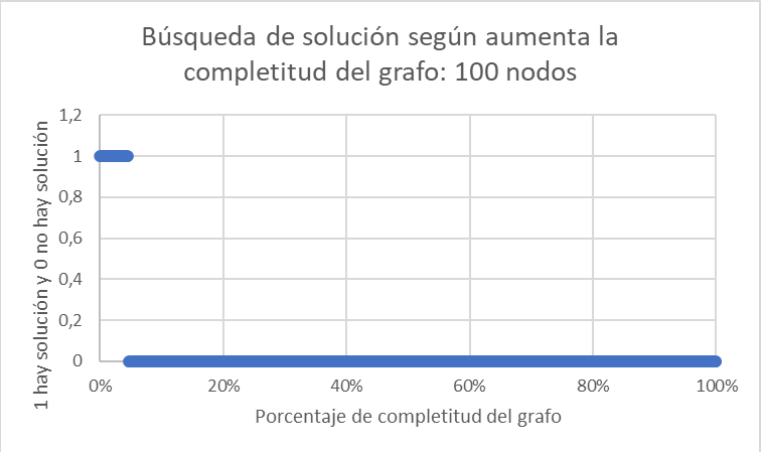

Figura 40. Representación de la existencia de soluciones en sucesivos grafos de 100 nodos según la completitud del grafo, siendo 0 % vacío y 100 % completo. 1 denota la existencia de solución y 0 que no existe.

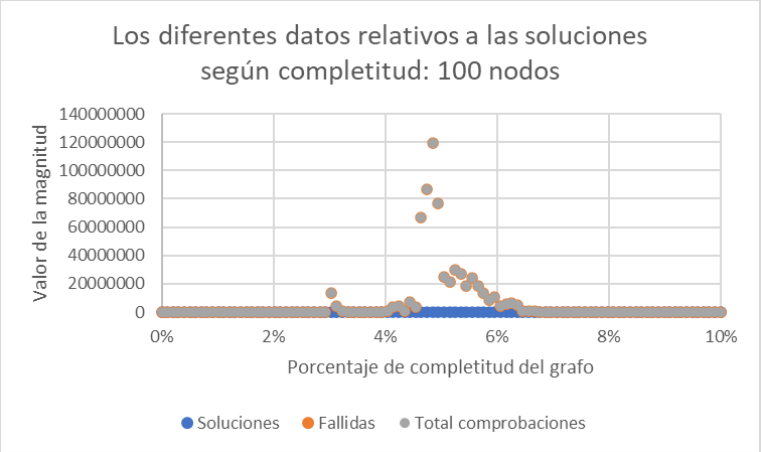

Figura 41. Representación de diferentes magnitudes relativas a la búsqueda de una solución según como de completo este el grafo, siendo 0 % vacío y 100 % completo. Las magnitudes son solución que indica si la hay con 1 o 0, fallidas configuraciones no aptas probadas y total suma de soluciones y fallidas.

En este caso podemos ver como el mayor número de comprobaciones se realiza en torno al 5 % de completitud. En otros valores de densidad el valor se estabiliza debido a que con densidades mucho mayores hacen falta pocas verificaciones para ver que no hay solución posible. En casos con valores menores el caso es el contrario, las comprobaciones son menores para encontrar una solución.

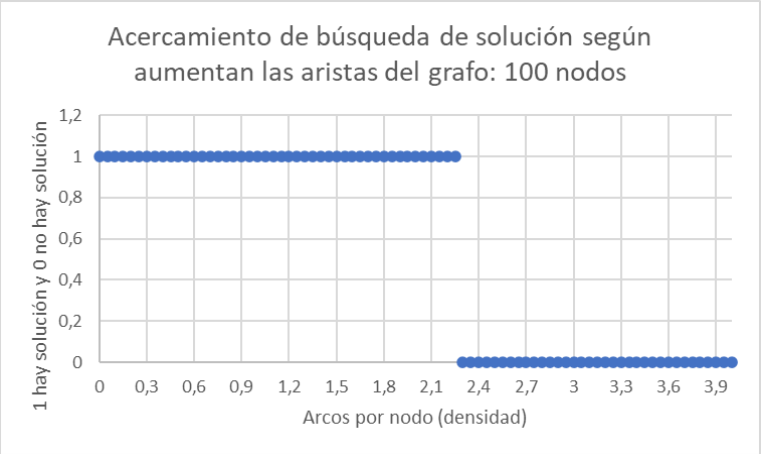

Figura 42. Gráfica detallada de soluciones vs. densidad, para ver con claridad umbral a partir del cual dejan de aparecer configuraciones 3-colorable.

El resto de las gráficas parecen mostrar ideas algo más triviales, como que cuando se pasa de una densidad de 2,5, el número de soluciones pasa a ser 0, esto se debe a que este algoritmo solo busca una solución posible, por lo que el número de soluciones en este caso pasa a un segundo plano. En la última gráfica, además, podemos ver otra vez claramente la afirmación de Hayes [1] de que a partir de una densidad, 2,35, no habrá soluciones que en este caso se da exactamente. Se produce, de nuevo, que el peor caso para tratar de buscar una solución con este algoritmo es un grafo con la mencionada densidad de arcos por nodo, 2,5, como ocurría previa inclusión de la heurística.

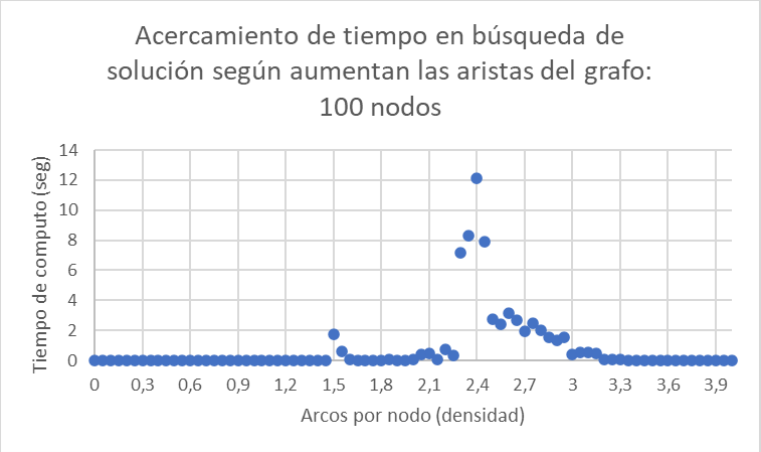

Figura 43. Gráfica detallada de tiempo de cómputo vs. densidad, para ver con claridad el peor caso en 2,5 arcos por nodo en 3-colorable.

4. Ejercicio 3. K-Colorabilidad k=2

a. Tamaño grafo para tiempo razonable

En esta nueva sección se continúa con el problema de colorabilidad mediante poda y heurísticas, pero ahora reducimos los colores que utilizan de 3 a 2. Se pone a prueba este algoritmo con una serie de ejecuciones en las que se va aumentando el número de nodos que conforman el grafo y tratando de encontrar la primera de las soluciones válidas. En este caso, con densidad de 1 arco por nodo se va desde 0 hasta 1030 nodos, punto en el que deja de subir exponencialmente el tiempo que toma encontrar una solución en el grafo o ver que no es posible.

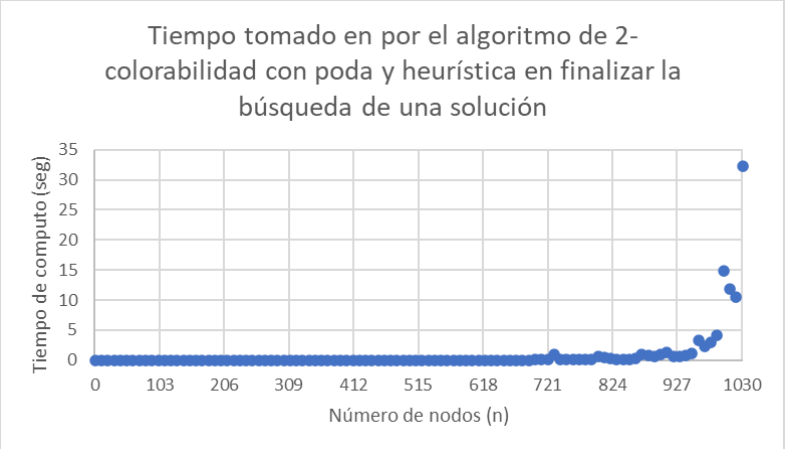

Figura 44. Estimación del número de nodos del grafo para encontrar una solución en tiempo razonable. Representa desde 1 hasta 1030 nodos con su tiempo en segundos.

Se observa en la gráfica que al final del recorrido el tiempo ha subido exponencialmente, se detiene precisamente en este punto porque gracias a las heurísticas pasado ese punto el algoritmo es capaz de podar al árbol rápidamente y ver que no tiene soluciones válidas.

Este comportamiento se tratará con más precisión en el análisis empírico.

b. Estudio analítico

Se analiza la implementación realizada del método de búsqueda de una solución, como ya se realizó en los apartados previos, y da como resultado la siguiente expresión:

$$T(1) = 29$$
$$T(n) = 2n + 2T(n-1)$$

$$\begin{aligned} T(n) &= 2n + 2(2(n-1) + 2T(n-2)) \\ &= 2n + 2^2 n - 1 \cdot 2^2 + 2^2 T(n-2) \end{aligned}$$

$$\begin{aligned} T(n) &= 2n + 2^2 n - 1 \cdot 2^2 + 2^2(2(n-2) + 2T(n-3)) \\ &= 2n + 2^2 n + 2^3 n - 1 \cdot 2^2 - 2 \cdot 2^3 + 2^3 T(n-3) \end{aligned}$$

$$\begin{aligned} T(n) &= 2n + 2^2 n + \cdots + 2^{k-1} n + 2^k n - 1 \cdot 2^2 - 2 \cdot 2^3 - \cdots - (k-2)2^{k-1} \\ &\quad - (k-1)2^k + 2^k T(n-k) \end{aligned}$$

$$n - k = 1;\ k = n - 1$$

$$\begin{aligned} T(n) &= 2n + 2^2 n + \cdots + 2^{n-2} n + 2^{n-1} n - 1 \cdot 2^2 - 2 \cdot 2^3 - \cdots - (n-3)2^{n-2} \\ &\quad - (n-2)2^{n-1} + 2^{n-1} T(1) \end{aligned}$$

$$\begin{aligned} T(n) &= n\left(\sum_{k=1}^{n-1} 2^k\right) + \left(\sum_{k=1}^{n-2} -k2^{k+1}\right) + 29 \cdot 2^{n-1} \\ &= n\left(\frac{2^{n-1}2 - 2}{2-1}\right) + (-2^n n + 3 \cdot 2^n - 4) + 29 \cdot 2^{n-1} \\ &= -2n + 2^n n - 2^n n + 3 \cdot 2^n - 4 + 29 \cdot 2^{n-1} \\ &= 2^{n-1}(29 + 3 \cdot 2) - 2n - 4 \\ &= -2n + 2^n n - 2^n n + 3 \cdot 2^n - 4 + 29 \cdot 2^{n-1} \\ &= 35 \cdot 2^{n-1} - 2n - 4 \end{aligned}$$

Resuelta esta recurrencia se concluye que el método que explora el grafo tiene un orden exponencial en base 2, $O(2^n)$.

En cuanto al proceso que se realiza previamente en el que se cuentan y ordenan los vértices según el número de aristas que tengan, se mantiene exactamente igual, es decir, el conteo de aristas por vértices es de orden polinómico cuadrado ($O(n^2)$) y la ordenación lineal logarítmico ($O(n\ log\ n)$).

Una vez visto todo, se puede concluir que la complejidad de esta nueva búsqueda es $O(2^n)$, el mayor de los tres órdenes.

c. Estudio empírico

Se prueba rápidamente que deja de haber soluciones alrededor del 1,3 % de completitud, lo que equivale a 0,64 aristas por nodo, como se ve reflejado en la siguiente gráfica con un aumento repentino de complejidad. Este nuevo umbral a partir del cual deja de haber soluciones es inferior al 2,35 del 3-colorable, lo que encontrar una solución sea más difícil para el mismo espacio, aunque era de esperar al tener menos posibles colores y haber observado este comportamiento en los apartados anteriores.

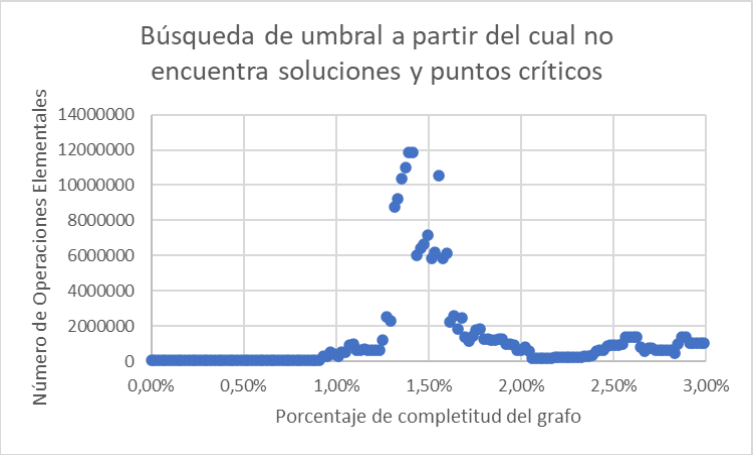

Figura 45. Representación del número de operaciones elementales asociado al problema de colorabilidad k=2 frente a la completitud del grafo.

Realizamos pruebas en este punto límite, 0,64 aristas por nodo, para tamaños de grafo sucesivamente superiores en número de aristas y así ver cómo se comporta nuestro algoritmo. Se muestran los resultados en las siguientes gráficas, en los que el rango de nodos es entre 1 y 2000 nodos, haciendo saltos de 10 en 10. En primer lugar, se presenta la de tiempo de cómputo en segundos y seguida de la de testigos como número de operaciones elementales.

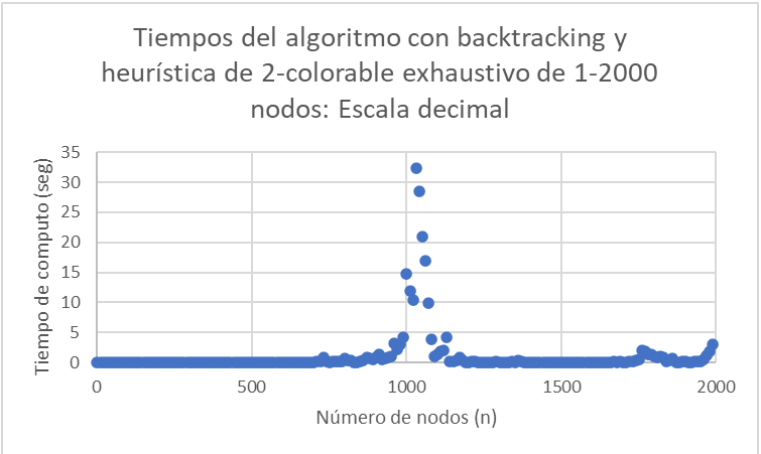

Figura 46. Representación del tiempo de cómputo para problema asociado al problema de colorabilidad k=2 frente al número de nodos.

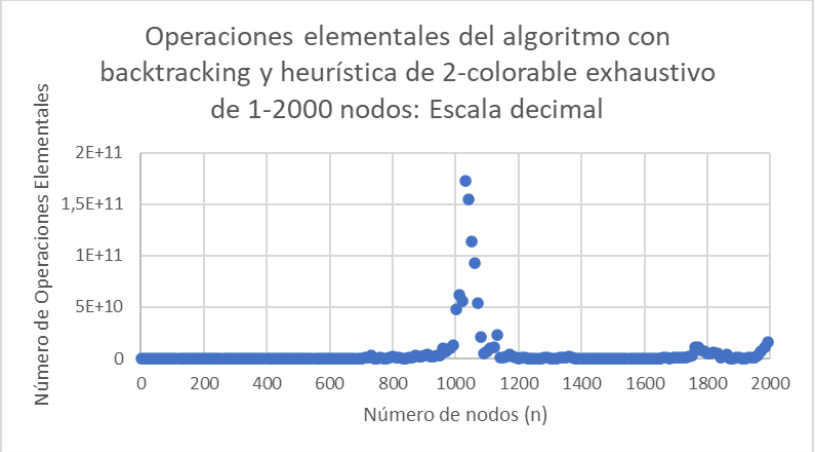

Figura 47. Representación del número de operaciones elementales para problema asociado al problema de colorabilidad k=2 con backtracking frente al número de nodos.

Podemos ver que aproximadamente a la mitad de las gráficas hay una subida brusca de las medidas. Este se debe a que el grafo en las sucesivas ejecuciones se va descubriendo nuevas aristas y llegado ese punto por la aleatoriedad de la generación de estas, se prueban nuevos caminos que son candidatos a ser solución, pero tiempo después 1100 nodos se abren otras nuevas que permiten descartarlas más rápidamente.

Se exponen ahora las versiones logarítmicas de las gráficas anteriores, para poder ver claramente que tendencia siguen sin perder información importante.

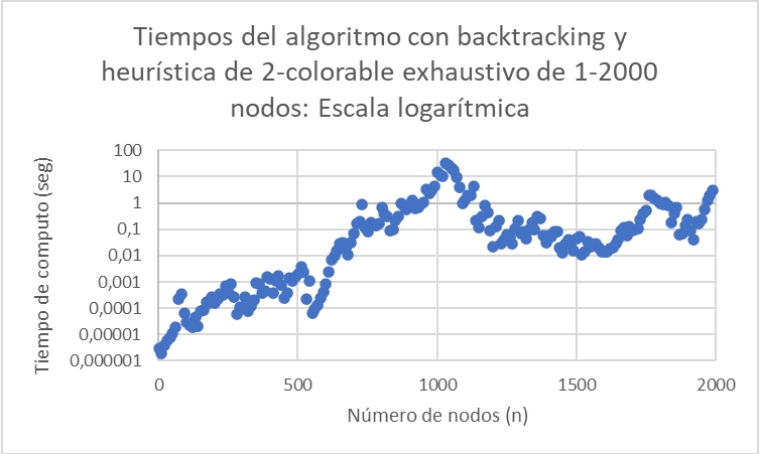

Figura 48. Representación del tiempo de cómputo para problema asociado al problema de colorabilidad k=2 con backtracking frente al número de nodos en escala logarítmica.

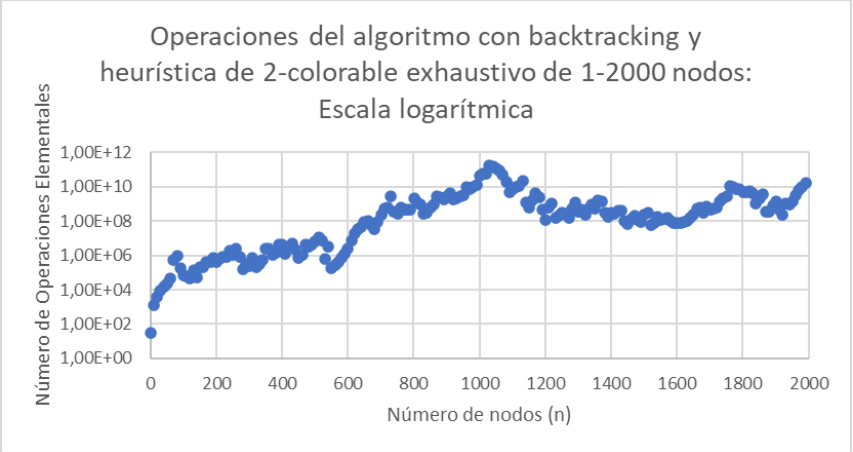

Figura 49. Representación del número de operaciones elementales para problema asociado al problema de colorabilidad k=2 con backtracking frente al número de nodos en escala logarítmica.

Se debe mencionar que llegados los 60 nodos, ya no se encontraran soluciones en las ejecuciones.

Una vez dicho, esto se puede ver que al comienzo, hasta alrededor de 600 nodos, la tendencia que siguen los datos es polinómica que se ve por la forma logarítmica que toman estas últimas gráficas. Una vez pasado este punto comienza a aumentar exponencialmente hasta los 1010 nodos, punto crítico de la ejecución, se aprecia por la linealidad de la logarítmica. Pasado ese punto se vuelve a la forma polinómica con un pequeño repunte exponencial al final, en torno a 1800 nodos.

Una vez analizado el comportamiento de los datos, se puede concluir que la complejidad de este algoritmo, a pesar de ser polinómico en muchos puntos, será exponencial.

Para mejorar la complejidad del algoritmo implementado, sería interesante utilizar BFS (búsqueda en amplitud) debido a que analíticamente tendría una complejidad de $O(n^2)$. Aunque esto iría en contra de la aplicación de backtracking en la situación, por lo que no es una opción en cuanto a lo que se pide en la realización de este proyecto.

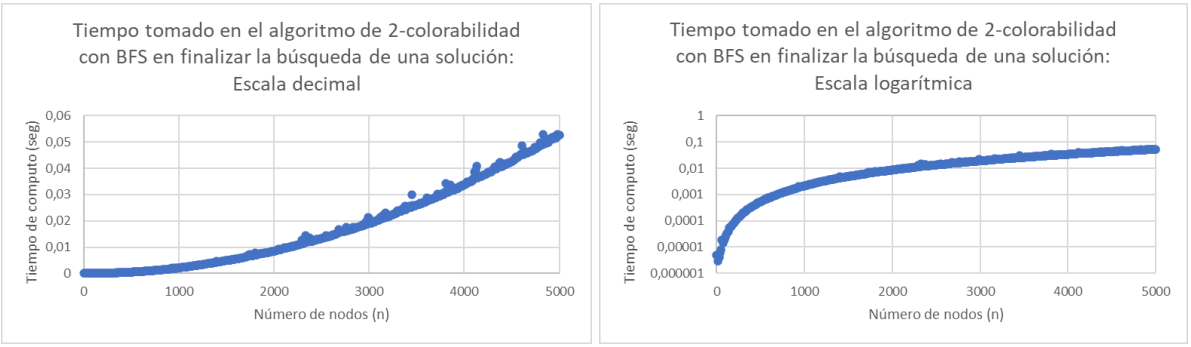

Figura 50. Representación del tiempo de cómputo para problema asociado al problema de colorabilidad k=2 con búsqueda en amplitud frente al número de nodos en escala decimal y logarítmica.

5. Ejercicio 4. Reducción K-Colorabilidad al de Clique

a. Nuevo método

Para este ejercicio se aborda el problema de k+1-clique o max-clique, este problema se ve relacionado con el problema de colorabilidad de manera que al encontrar un k-clique, es decir un subgrafo que tenga k nodos conectados entre ellos se puede garantizar que el grafo no es k-1 coloreable, ya que tiene que haber al menos k colores diferentes para que el clique fuera coloreable.

El método que se ha utilizado es el propuesto en clase, este método recorre todos los nodos comprobando que no hayan sido descartados. Primeramente, antes de considerar un nodo se comprueba si se cumple la heurística implementada, esta heurística utiliza un array rellenado previamente con el número de conexiones de cada nodo, si un nodo cuenta con más de k-2 conexiones se considera para formar un clique. Cuando se considera un nodo se añade a una lista, cada vez que se añade un nodo se comprueba si puede o no formar un clique con los demás de la lista, esto es que esté conectado con los demás nodos de la lista. Si se encuentra un clique de tamaño k+1, se llega al caso base y se imprime la información de dicho clique. Esto se hace de forma recursiva hasta la profundidad k, cuando un nodo ya ha sido considerado, pero no forma un clique, se descarta/poda y no se vuelve a considerar.

b. Estudio analítico

Al realizar el análisis analítico del algoritmo se puede determinar que el mejor caso sucede cuando los k+1 primeros nodos forman un clique. Por otra parte, en el peor caso se deben recorrer todos los subgrafos de G, al estar utilizando un algoritmo de fuerza bruta el tiempo que se tarda es $O(n^k k^2)$ dado que hay n^k subgrafos y cada uno tiene k^2 aristas. Cuando k es constante como en nuestro problema podemos resolver el problema en tiempo polinómico O(n^k), previamente estos resultados ya fueron obtenidos en la bibliografía de la que han sido obtenidos [3].

Aun así a efectos prácticos, se desarrollará la obtención de esta complejidad analíticamente a continuación resolviendo la recursividad.

Para empezar, el primer bucle de nuestro método *encontrarK_clique* se ejecuta n veces, después en el caso de que todas las iteraciones superen la heurística implementada (más de k-2 conexiones), se procede a comprobar si los nodos que hay en nuestra lista forman un clique con la función *esClique* que tiene una complejidad de $T(n,k) = k^2 - k$ ya que el bucle exterior se ejecuta un máximo de k veces al recorrer desde 0 hasta la longitud de la lista que sirve para comprobar si los nodos que representan los índices dentro de ella forman un k-clique. El bucle interior como máximo se ejecuta k-1 veces ya que la variable de control se inicializa con la misma condición pero con una unidad más que la variable de control del bucle exterior. Por simplicidad ya que solo se busca la mayor complejidad tomaremos la complejidad de *esClique* como $T(n,k) = k^2$.

Después de comprobar si la lista forma un clique se encuentra la recursividad, hasta ahora la ecuación de la recursividad se resumiría en:

$$T(n,k) = n * [k^2 + T(n,k-1)]$$

Si se desarrolla la fórmula:

$$T(n,k) = n * [k^2 * n * [k^2 + T(n,k-2)]$$

$$T(n,k) = n * k^2 + n^2 * k^2 + n^2 * T(n,k-2)$$

$$T(n,k) = k^2 * (n + n^2 + n^3) + n^3 * T(n,k-3)$$

Si llevamos hasta el infinito:dado que nuestro caso base devuelve 1 y viendo el patrón:

$$T(n,k) = k^2 * (n^k + n^{k-1} + \ldots + n^2 + n) + n^k * 1$$

$$T(n,k) = k^2 * (n^k + n^{k-1} + \ldots + n^2 + n) + n^k$$

Por tanto si obtenemos la complejidad mayor vemos que el método concordando con la bibliografía tiene una complejidad de $O(n^k * k^2) => O(n^k)$.

c. Estudio empírico

Para el estudio empírico se ejecuta el método y se mide el tiempo que tarda en obtener la solución para el clique k+1 además de las operaciones elementales necesarias por medio de testigos.

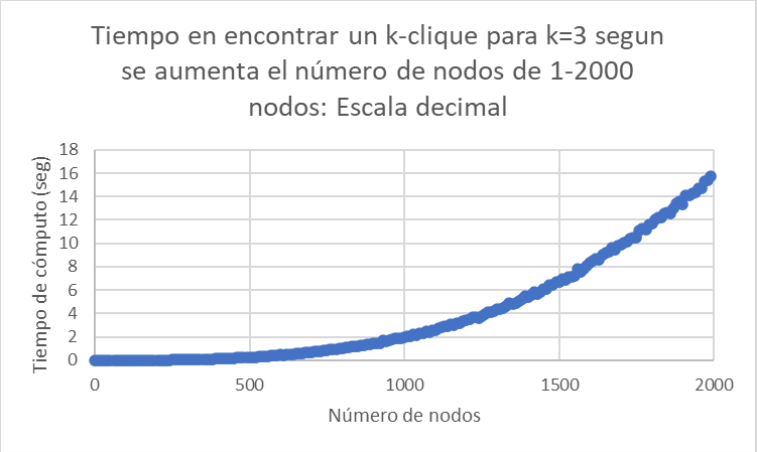

Figura 51. Gráfica comparativa del tiempo necesario para encontrar un k-clique según el número de nodos en escala líneal.

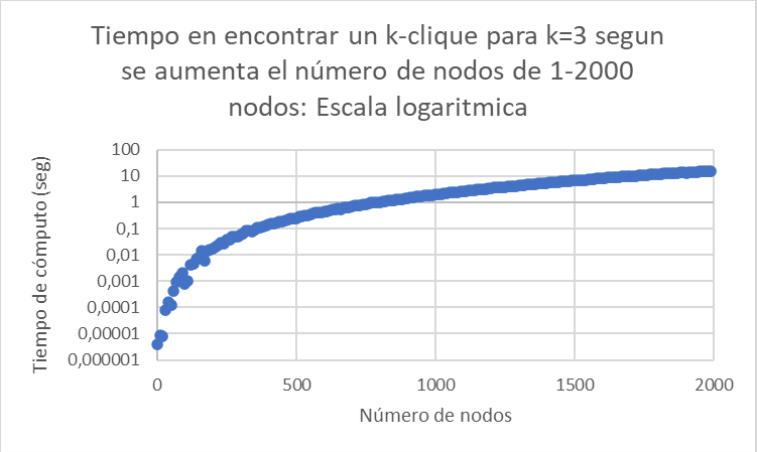

Figura 52. Gráfica comparativa del tiempo necesario para encontrar un k-clique según el número de nodos en escala logarítmica.

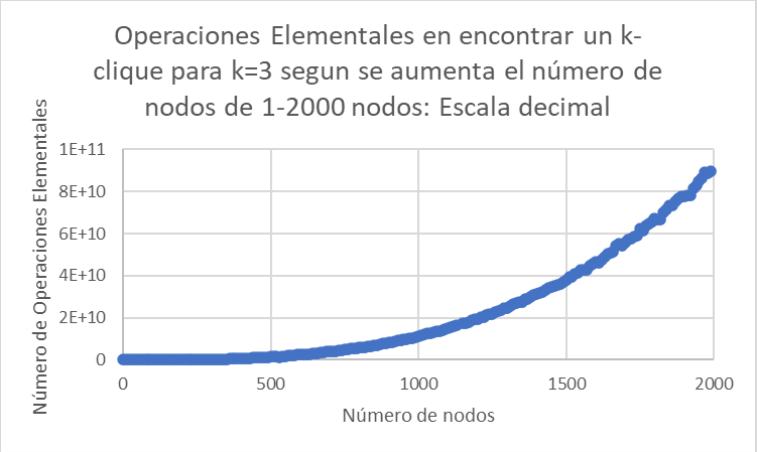

Figura 53. Gráfica comparativa del número de operaciones elementales necesarias para encontrar un k-clique según el número de nodos en escala decimal.

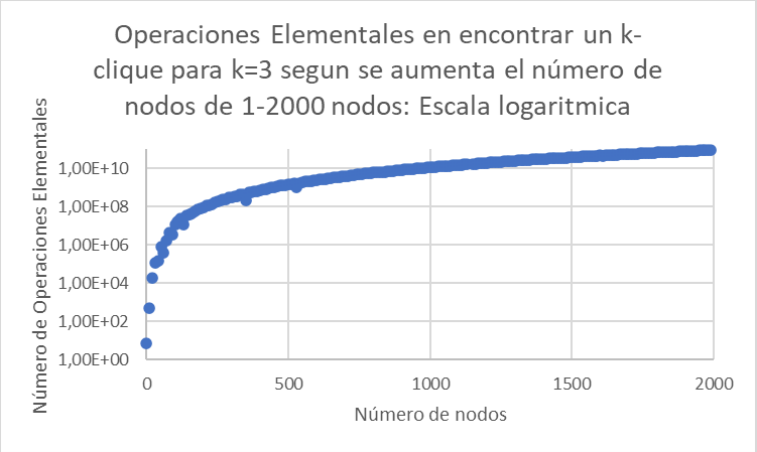

Figura 54. Gráfica comparativa del número de operaciones elementales necesarias para encontrar un k-clique según el número de nodos en escala logarítmica..

En la Figura 51 y Figura 53 se puede observar como el tiempo crece de manera parecida a una exponencial pero para comprobarlo se han realizado la Figura 52 y la Figura 53 en las cuales al tener escala logarítmica y no encontrarse una línea se puede descartar la exponencialidad y asumir que el tiempo que se tarda es polinómico concordando con nuestro análisis analítico.

6. Ejercicio 5. Estudio teórico

a. Clase K-Coloreado y K-Clique

Para poder clasificar la clase de cada problema debemos entender las definiciones de cada una de ellas:

- **Clase P**: A esta clase pertenecen los problemas de decisión que puedan ser resueltos en tiempo polinómico sobre una máquina de Turing determinista.
- **Clase NP**: Es el mismo caso que en los problemas de clase P, solo que en este caso la máquina de Turing determinista pasa a ser no determinista. Debido a esto, los problemas que nos encontramos en la clase P también se encuentran contenidos en esta clase.
- **Clase NP-Completo**: Este tipo de complejidad son aquellos problemas de decisión que pertenecen a NP y que cualquier otro problema NP pueda ser reducible a este.

El primer problema que debemos clasificar es el problema de k-colorabilidad. Podemos asegurar que el problema no es clase P para cualquier k mayor que 2. Esto es debido a que para k=2 es posible encontrar un método con tiempo polinómico si intenta encontrar un grafo bipartito. Para cualquier otra k, el problema pasa a ser NP debido a su imposibilidad de resolver el problema en tiempo polinómico con una máquina de Turing determinista.

La duda que surge en este momento es diferenciar si este problema es NP o NP-completo. Podemos confirmar es completo reduciendo el problema 3-SAT a 3-Colorable, dado que 3-SAT es conocido por ser NP-Completo. Para conseguir esto, cada literal del SAT pasa a ser un par de nodos (x, x') en un grafo. Estos nodos pasan a estar conectados entre sí y a un mismo nodo base. De esta manera, tendremos un grafo 3-colorable si y solo sí se cumple la ecuación del 3-SAT. Todo este proceso se muestra de manera gráfica en la bibliografía [2]. Es muy importante recalcar que esta transformación se realiza en tiempo polinómico.

Se concluye, por tanto, que el problema de K-Colorabilidad pertenece a la clase NP-Completo.

El mismo caso se tiene con el problema de K-Clique. Sabemos que no pertenece a clase P, pero se debe comprobar si es posible su pertenencia a los problemas NP-Completo. Para esto podemos realizar la reducción del problema k-coloreable a k-clique. Esta transformación se plantea en el capítulo 4, y es que si sabemos que un grafo tiene un clique k obtenido mediante k-clique, dicho grafo no es coloreable para el rango [1, k-1]. Dicho esto, se puede afirmar que el problema k-clique es, por tanto, un problema de clase NP-completo.

b. Implicaciones

El hecho de poder utilizar algoritmos independientemente del problema (todo esto debido a que pertenecen a la clase NP-Completo como se concluye en la sección anterior) permite que se puedan encontrar algoritmos para resolverlos de una menor complejidad, al utilizar otro enfoque. La transformación en tiempo polinómico de los problemas también permite poder representar el problema de una manera más sencilla (como es este caso, o la reducción del problema 3-SAT a 3-colorable), de esta manera es posible encontrar heurísticas de manera más efectiva, al simplificar el razonamiento que implican, cosa que puede afectar a la complejidad del programa.

7. Conclusión

El trabajo realizado representa el estudio de los algoritmos NP-C relacionados con la colorabilidad, en concreto, el problema de k-colorabilidad y el problema de k-clique o max-clique.

Con este estudio se muestra la posibilidad de transformar diferentes tipos de problemas NP-C entre ellos de manera polinómica. Además, se demuestra con análisis analíticos y empíricos la complejidad de los mismos.

El trabajo a realizar ha sido en gran parte solventado por las aportaciones de código en Aula Global que han reducido el tiempo de implementación del código para que el trabajo se centrara en estudiar la bibliografía, analizar y ejecutar con diferentes parámetros los problemas planteados además de probar diferentes heurísticas. Todo el código que se ha utilizado, con sus respectivas modificaciones, se encuentra en la carpeta Código.

Por otra parte, se han aprendido conceptos nuevos como las transiciones de fase y su importancia en este tipo de problemas, ya que explican cómo los diferentes parámetros afectan de forma sustancial a los resultados obtenidos. Se pueden visualizar con más detalle estos resultados en la hoja de cálculo de la carpeta Gráficas.

Gracias a esta práctica hemos podido poner en práctica los conocimientos adquiridos durante la lectura de la bibliografía y hemos tenido la posibilidad de asimilarlos mediante ejemplos prácticos.

8. Referencias

[1] B. Hayes: “Computing Science: On the Threshold”. American Scientist, vol. 91, No. 1, January-February 2003, pp.12-171.

[2] L. Mouatadid, «Introduction to Complexity Theory: 3-Colouring is NP-complete», p. 3.

[3]Downey, R. G.; Fellows, M. R. (1995), "Fixed-parameter tractability and completeness. II. On completeness for W[1]", Theoretical Computer Science, 141 (1–2): 109–131, doi:10.1016/0304-3975(94)00097-3.