Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

Integrante	Libreta Universitaria	Correo electrónico
Collasius, Federico	164/20	fede.collasius@gmail.com
Donzis, Tomás	443/20	donzis.tomy@gmail.com
Totaro, Facundo Ariel	43/20	facutotaro@gmail.com
Venturini, Julia	159/20	juliaventurini00@gmail.com

Primer cuatrimestre - 2022

Keywords: Grafos Geodésicos, Union Find, Algoritmo de Johnson, Bertossi.

Grafo Geodésico

Presentación del problema:

Queremos determinar, dado un grafo conexo G, si el mismo es geodésico en tiempo O(nm). En caso afirmativo, el algoritmo debe devolver una matriz de i*j donde cada posición i,j representa el predecesor del j-ésimo vértice en el único camino de i a j. En caso negativo deberá devolver dos caminos mínimos distintos entre algún par de vértices del grafo.

Ahora bien, la idea del algoritmo es correr una implementación de BFS modificada (llamemosla BFS') desde cada nodo del grafo, para encontrar todos los caminos mínimos entre todos los vértices, determinar si el grafo es geodésico y a su vez ir generando la salida (matriz con predecesores o vector de caminos mínimos), sea geodésico o no. Veamos en detalle cómo se logra esto.

Por cada vértice del grafo, la función solver va a correr BFS', que me devuelve true si encontró dos o más caminos mínimos entre el vértice de origen y algún otro vértice del grafo. Si es true significa que el grafo no es geodésico, entonces se modificará la variable global esGeodésico para que sea false. Si al finalizar la ejecución de los n BFS' la variable esGeodésico es true, se imprimirá por pantalla la matriz de predecesores. En caso contrario, se imprimirán dos caminos mínimos distintos entre algún par de vértices del grafo. En ambos casos, la función BFS' es la que se encarga de completar las estructuras que se usan para imprimir la salida por pantalla. Veamos qué atributos le damos a cada vértice para entender qué es lo que hace BFS'.

Cada vértice tiene los siguientes atributos:

- Número: identifica a cada vértice.
- Distancia: indica la distancia del vértice al origen.
- Color: varia entre blanco, gris y negro y se usa para identificar el estado del vertice en la ejecucion de BFS'.
- Padre: identifica al antecesor del vértice en la búsqueda.
- Caminos Mínimos: la cantidad de caminos mínimos que llegan a ese nodo desde un origen en una búsqueda BFS' particular.

La función BFS' comienza seteando todos los vértices del grafo con distancia infinito, color blanco, padre -1 y caminos minimos 0. Luego toma el vértice de origen, le asigna color gris ya que lo "descubrió", distancia 0 y caminos mínimos 1. A continuación lo encola en una queue. Mientras esta queue no esté vacía, el algoritmo va a ir desencolando vértices y va a recorrer a sus vecinos. Si son blancos, es decir no los descubrió todavía, los pinta de gris, les asigna como padre al vértice que desencolé y los encola en la queue. Luego chequea si la distancia de este vértice "vecino" es infinito (esto quiere decir que lo descubrí por primera vez). En caso afirmativo, le asigna distancia igual a la distancia del padre + 1 y le asigna la misma cantidad de caminos mínimos que tenga el padre. Si la distancia no es infinito, es decir que ya descubrí al vértice en algún momento de la ejecución de BFS', me fijo si la distancia del "vecino" es igual a la distancia del vértice que desencolé + 1. Si esto es cierto, significa que hay dos caminos mínimos desde el origen al "vecino", ya que su padre será distinto (porque nunca recorro dos vértices dos veces) pero puedo llegar mediante el vértice que desencole con un camino de igual longitud (que es mínimo ya que estamos recorriendo con BFS). De modo que incremento el atributo caminos mínimos del vértice "vecino" en 1. Luego, chequeo si la cantidad de caminos mínimos para el vértice "vecino" son más de 1, lo que indicaría que el grafo no es geodésico. Si es asi, le asigno false a la variable esGeodesico y reconstruyo dos caminos mínimos que vayan desde el vecino hasta algún vértice en común, agarrando por un lado a los antecesores del vecino y por el otro a los antecesores del vértice que desencolé. Esta información queda guardada en un vector de 2 posiciones que representa los dos caminos mínimos con mismo origen y destino. En caso contrario, es decir que la cantidad de caminos mínimos del "vecino" es 1, voy a construir la salida que debería imprimir el

algoritmo si el grafo fuese geodésico. Todavía no se si esto se va a cumplir pero en caso de que se cumpla, ya voy a tener la matriz armada lista para imprimir. Para construir la salida, en cada corrida de BFS' voy a completar solo una fila de la matriz de predecesores, que corresponde a la fila del vértice de origen. Es decir, si el vértice de origen es el 2, completare solo la fila 2 de la matriz en la ejecución de BFS'(2). De este modo, si el grafo es geodésico, al finalizar la ejecución de los n BFS', la matriz quedará completa. Ahora bien, cada fila de la matriz se completa mirando al padre del j-esimo vértice en cada iteración. Es decir, si yo estoy iterando sobre el j-esimo vértice, a la posición (s, j) de la matriz (s siendo el origen) le asignare el valor del padre de j. Una vez que miramos a todos los vecinos del vertice que desencole, lo pintamos de negro. Al quedar la cola vacía, es decir que no quedan vértices por recorrer, devolvemos el valor de esGeodesico.

En este punto es donde la función solver se fija si la variable esGeodesico es true, imprime un 1 seguido de la matriz de predecesores. Si es false, imprime un 0 seguido de los dos caminos mínimos guardados en el vector caminosMinimos. Esto concluye la ejecución del algoritmo.

El Algoritmo:

El algoritmo devuelve true si para todo nodo del grafo el siguiente algoritmo da true:

```
PSEUDOCODIGO ALGORITMO BFS:
esGeodesico = true
Para cada vértice del grafo:
    Inicializo su color en blanco, su distancia en infinito y su padre en -1.
    Marco el vértice de origen s con color gris, distancia 0, padre -1 y
    caminos mínimos en 0.

    Mientras la cola de vértices no esté vacía:
        Desencolo un vértice u y me fijo para cada uno de sus vecinos v:
            Si no fue descubierto todavía (su color es blanco)
                Lo marco en gris, le asigno a u como su padre y lo encolo.
            Si la distancia del vértice es infinito:
                Le asigno distancia = distancia de u + 1
                Le asigno caminos mínimos = caminos mínimos de u
            Sino:
                Si su distancia es la distancia de u + 1 (hay más de un camino
                mínimo hasta este vértice).
                    Le asigno caminos mínimos = caminos mínimos de u + 1 +
                    caminos mínimos de v.
                    Cambio esGeodesico a false.
            Si la cantidad de caminos mínimos de v es > 1:
                Armo dos caminos mínimos y los agrego al vector de caminos
                mínimos agarrando los predecesores de u por un lado y los
                de v por el otro hasta llegar a un nodo en común.
            Sino:
                Para la fila s de la matriz de predecesores le asigno a la posición (s, u)
                el valor del padre de u. (Salvo a (s,s) que le asigno a si mismo)
    Marco u de color negro
Devuelvo esGeodesico
```

Veamos que se cumple la complejidad O(mn) pedida:
El costo de construir el grafo es $O(n*m)+O(n^2)$ ya que estamos creando la lista de adyacencias y la matriz de predecesores respectivamente. El costo de cada BFS' es O(n+m) ya que se recorre cada vertice una sola vez y para cada uno de ellos se mira a todos sus vecinos, recorriendo todas las listas de adyacencias. Esto lo hago desde cada vertice por lo que tengo n BFS's. Para la sumatoria de todos los BFS' el costo combinado de construir el output es a lo sumo $O(n^2)$. Luego, el costo total del algoritmo que determina si un grafo es geodesico es $O(n*m)+O(n^2) + O((n+m)n) + O(n^2) = O(2n(n+m) + 3(n^2)) = O(n+m) + n^2)$. Ahora bien, la cantidad máxima de aristas que puede tener un grafo conexo no dirigido es $n * (n-1)/2$, lo que significa que en

peor caso m es del orden de n^2, que es equivalente a decir que $O(m) = O(n^2)$. Entonces, la complejidad pedida es $O(n(n+m) + n^2) = O(n(n+n^2) + n^2) = O(n^3)$. De este modo obtenemos que la complejidad total del algoritmo es $O(n*m + n^2) = O(n^3+n^2) = O(n^3)$ y cumple con la complejidad pedida.

Cantidad de Componentes Conexas:

Presentación del problema

El segundo ejercicio consta de encontrar la cantidad de componentes conexas en un grafo. El input que se toma para el problema es un grafo representado como una matriz de ceros y unos de dimensión *m* x *n*, donde los unos representan la presencia de un vértice y los ceros la ausencia de uno. Según esta representación, dos vértices son adyacentes si, justamente, están adyacentes en la matriz.

Este problema es similar al problema de hallar la cantidad de islas en una matriz, donde hay tierra si el valor en una posición es 1 y agua si la posición es 0, el cual podría resolverse de forma intuitiva y sencilla recorriendo la matriz, armando un grafo mediante las adyacencias y aplicarle DFS hasta que todos los nodos hayan sido visitados, en cuyo caso el resultado sería la cantidad de veces en las que se tuvo que aplicar DFS. Sin embargo, nosotros buscamos poder resolverlo con ciertas limitaciones. Por ejemplo, la complejidad espacial debe ser de O(n), donde n es la cantidad de columnas de la matriz y la complejidad temporal debe ser de $O(mn*\alpha(mn))$, donde α es la función inversa de Ackermann. Por estas restricciones deberíamos analizar solo una fila nueva por iteración y encontrar una estructura de datos acorde para conseguir la complejidad deseada.

Union Find (o Disjoint Set)

Como mencionamos previamente, querríamos identificar una estructura de datos que nos ayude a conseguir la complejidad temporal de $O(mn*\alpha(mn))$.
En principio notemos que *mn* es la cantidad total de posiciones que tendrá nuestra matriz y por lo tanto la cantidad máxima que podríamos tener de vértices. Por este motivo, podemos determinar que vamos a operar sobre cada vértice una cantidad constante de veces de forma tal que obtengamos luego la expresión de la inversa de Ackermann.
Afortunadamente, tanto *Introduction to Algorithms* como las clases prácticas nos dieron una noción de una estructura de datos que consigue esa cota con la sumatoria del costo de sus operaciones. Esta estructura es el Disjoint Set que tal como indica su nombre puede representar un conjunto disjunto y sus respectivas operaciones:

- **Crear un set** (makeSet), es decir darle una etiqueta distintiva a un conjunto.
- **Encontrar el set** (findSet), que determina cuál es *el elemento representante* del set al que pertenece cierto elemento y que justamente dictamina la pertenencia de dos elementos en un mismo conjunto.
- La función **Union** (∪), que une dos conjuntos disjuntos, agregando según algún criterio los elementos de un conjunto en el otro mediante la redefinición del *elemento representante* de los elementos del set dominado.

Esta estructura de datos presenta una representación de árbol, donde el árbol de mayor altura es el que domina sobre aquellos de menor altura y el encontrar el set consiste en subir hasta el nodo raíz. Sin embargo, nosotros lo implementamos sobre un arreglo de *parents* (o tags) de n posiciones que indica cuál es el *elemento representante* del set al que pertenece el vértice de cada columna de una fila, que es más estático y no requiere puntualmente el manejo de punteros, haciendo el algoritmo algo más accesible a la

comprensión. En cuanto a los rasgos generales de la implementación usamos los algoritmos standard, planteados por la bibliografía de la materia para cada operación.

Sin embargo, cabe aclarar algunos detalles de nuestra implementación. Por ejemplo, nuestro Disjoint Set tiene dos variantes fundamentales para obtener la complejidad deseada. La primera es el **path compression**, en la que al hacer findSet sobre una posición de la matriz, directamente modificamos el parent de la posición en el arreglo por el *representante*, llamando recursivamente a la función y asignando el resultado a la posición, para no tener que "subir" a la raíz del árbol en cada llamada, sino simplemente devolver el *elemento representante*. La segunda es el Union by Rank, que justamente determina que el conjunto dominante en la unión es la que más veces dominó previamente, para lo cual mantenemos un arreglo de ranks por posición del arreglo parents.

Por último, agregamos dos métodos adicionales a la clase implementada cuyo uso explicaremos al presentar el algoritmo diseñado.

El algoritmo

Primero presentaremos una versión en pseudocódigo del algoritmo propuesto y después justificaremos su correctitud, además de presentar las nuevas operaciones que agregamos al Disjoint Set como anticipamos en la subsección anterior.

```
1   Algoritmo cantidadComponentesConexas(in M: N^(m×n) (una fila a la vez)) → out res: N
2     Crear un UnionFind vacío (UF).
3
4     Para cada i = 0, …, m:
5       Traer la fila i del input.
6
7       Para cada j = 0, …, n:
8
9         Si M[i][j] == 1:
10          UF.makeSet(j), res++ y
11          si el vértice actual tiene precedencia en la fila anterior y aún no cambié la raíz de la componente
12          a la que pertenece en esta fila:
13            UF.cambiarRaiz(j), res--;
14
15        Si el vértice actual tiene un adyacente a su izquierda:
16          UF.union(j, j-1), res--;
17
18        Si el vértice actual es adyacente hacia arriba con otra componente conexa
19        (es decir con distinto parent):
20          UF.union(j, j + n), res--;
21
22      Reestablecer las estructuras de datos de forma tal que la fila actual, procesada en esta iteración,
23      se convierta en la fila anterior.
24
25      Reestablecer el UF.
26
27    Retornar res.
```

Antes de analizar el algoritmo, notemos qué estructuras de datos precisaríamos para implementarlo.
Nos es de conveniencia analizar una fila por vez pero aún así manteniendo constancia de la fila anterior, al menos de los parents (o tags) presentes en la última iteración del ciclo, por lo tanto nuestro Union Find tiene dos niveles en su arreglo de parents, indicando en la primera fila aquellos de las posiciones en la iteración anterior y en la segunda los de la iteración actual. Además, por fines prácticos, para verificar que un vértice presente en la fila actual extiende a una componente conexa de la fila anterior, nos serviría guardarnos en sí la última fila procesada, por lo que el arreglo en el que se guardan los valores de la fila actual consta de dos filas, una para los vértices de la última iteración y otra para la actual. Esto no excede la cota requerida para la complejidad espacial ya que $2n \in O(n)$.

Además, para determinar si la *raíz* de una componente conexa ya fue modificada en la fila actual, nos sirve tener un arreglo de *n* booleanos indicando si el tag del parent de la columna $0 \leq j < n$ de la fila anterior ya fue redefinido. Habiendo presentado las estructuras utilizadas y mostrado que las cotas espaciales fueron cumplidas, podemos explicar el algoritmo.

Notemos que en cada iteración, si tenemos un vértice en la columna actual (j) de la fila en proceso, creamos un set en el Union Find con el tag j + *n*, ya que consideramos que un vértice antes de ser procesado puede considerarse una potencial nueva componente conexa. Por lo tanto, lo primero que verificamos es si en la fila anterior tiene un vértice adyacente, inductivamente perteneciente a otra componente conexa, y si esa componente conexa ya fue, valga la redundancia, conectada en la fila actual. En el caso negativo, esta redefinición del tag de la componente la implementamos como **cambiarRaíz** en el Union Find, de forma tal que la anterior *raíz* apunte a la posición j + *n* del arreglo de parents, para así al hacer findSet sobre cualquier miembro de la componente conexa en la fila anterior apunte directamente a la *raíz* redefinida. Si se modifica la raíz, decrementamos la cantidad de componentes halladas, ya que el vértice actual pertenecía a otra componente *verticalmente* y más aún, es el nuevo vértice representante de la misma. Además asignamos "true" en la entrada j del arreglo de booleanos, ya que la posición actual fue asignada como raíz de una componente conexa preexistente.

Luego hacemos las verificaciones de si el vértice de la columna j es adyacente tanto a izquierda como hacia arriba con componente conexas preexistentes, en caso que ocurra, se hace Unión entre j y el parent anterior para que la nueva componente "absorba" al nuevo vértice, de forma tal que el parent del nuevo vértice pasa a unificarse con la componente conexa recién unida. Cabe destacar, nuevamente que si logré conectar el vértice actual a una componente conexa, entonces este deja de ser una potencial componente y en ambas verificaciones de unión, en caso de ser positivas, se decrementa el resultado y por tanto las componentes conexas del grafo.

Por último, reestablecemos tanto el arreglo de *2 x n* como el de booleanos y las estructuras internas del Union Find (con un método agregado **reestablecer**) de forma tal que en la siguiente iteración, las filas procesadas en la iteración actual pasen a comportarse como filas anteriores, de las que se heredan las componentes y con las cuales hay adyacencias.

En cuanto a la complejidad del algoritmo, podemos ver fácilmente que estamos iterando sobre la cantidad de filas (*m)* cantidad de columnas (*n*) veces para hacer el input y procesar los vértices independientemente. Como todas las operaciones para los vértices ocurren en el Union Find, su complejidad, según la bibliografía, se basa en la complejidad de la **unión** y el **findSet** y la cantidad de llamadas a las mismas, la cuál se reduce a la expresión de la función inversa de Ackermann evaluada en la cantidad de llamadas. Por lo tanto conseguimos la cota que deseabamos de $O(mn * \alpha(mn))$.

Luego, la complejidad de reestablecer los arreglos y el Union Find es de $O(n)$ ya que solo se recorre una única vez cada arreglo por cada fila para preparar las estructuras para la próxima iteración. Luego el algoritmo tiene complejidad $O(mn * \alpha(mn))$. En el apéndice se puede ver la performance y resultados del algoritmo para instancias de grafos de distintas dimensiones y densidad de vértices.

Algoritmo de Johnson

Presentación del problema

En un grafo, para encontrar el camino mínimo entre un nodo y todos los otros del grafo existen varios algoritmos. Uno de ellos es el algoritmo de Dijkstra. El problema del algoritmo de Dijkstra es que no se puede garantizar su correctitud si el algoritmo tiene aristas de longitud negativa. Dado un grafo G(V,E), su

complejidad temporal si lo implementamos sobre una cola de prioridad usando una lista de adyacencias es de O(log |V| * (|V|+|E|)) == O(log |V| * |E|). También existe el algoritmo de Bellman-Ford, que permite lo mismo que Dijkstra, pero para grafos con pesos negativos. Hay un inconveniente que si un grafo tiene aristas con pesos negativos, puede tener ciclos con longitud negativa. Y al tener ciclos con longitud negativa, siempre que se relajen las aristas, se encontrará que existe un camino de menor longitud para, al menos, a los nodos que pertenecen a ese ciclo de longitud negativa, ya que al dar una vuelta por el ciclo y sumar los pesos del ciclo, se tendrá una distancia negativa y entonces, si se da vueltas por el ciclo, siempre llegar al nodo costará menos que antes. Sin embargo con Bellman-Ford, podemos detectar estos ciclos y no dar un resultado si existen.
La complejidad temporal de este algoritmo es O(|V||E|). Notemos que log |V| < |V| asintóticamente, por lo que, el algoritmo de Dijkstra es más eficiente que el de Bellman-Ford en tiempo.
Sin embargo, existe una forma, para transformar parcialmente el grafo en uno en el que sus pesos sean todos positivos, aplicar Dijkstra y tener la longitud original del camino. Esto es mediante el algoritmo de Johnson.
Primero lo que se hace es agregar un nuevo nodo s y |V| aristas dirigidas desde el hacia el resto de los nodos con peso 0. La idea para redefinir los pesos consiste en definir un nuevo peso ŵ(u,v) tal que ŵ(u,v) = w(u,v) + h(u) - h(v), siendo h : V -> R una función que no depende del camino y queremos que ŵ(u,v) >= 0 para todo par de vértices (u,v).
Sin embargo, un grafo contiene ciclos negativos si y solo si esta redefinición de los pesos tiene ciclos de pesos negativos. Por lo que al aplicar Bellman-Ford también se los encontrará
Demostración.
Sea c un camino <v_{inicio}, $v_{inicio+1}$,$v_{inicio+2}$, … ,v_{fin}> y ŵ(c) la suma de los pesos de un camino. Queremos probar que ~w(c) = w(c)+ h(inicio) - h(fin)
Luego

$$\hat{w}(c) = \sum_{i\,=\,inicio}^{fin} \hat{w}(v_{i-1}, v_i) = \sum_{i\,=\,inicio}^{fin} w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) =$$

$= (\sum_{i\,=\,inicio}^{fin} w(v_{i-1}, v_i)) + h(v_{inicio}) - h(v_{fin})$ porque es una suma telescópica =

$$= w(c) + h(v_{inicio}) - h(v_{fin})$$

Ahora bien, si tenemos un ciclo, $v_{incio} = v_{fin}$. Por lo que ŵ(c) = w(c) + h(v_{incio}) - h(v_{fin}) = w(c) + h(v_{incio}) - h(v_{incio}) = w(c). Y si w(c) tiene peso negativo, entonces ŵ(c) también lo tendrá. Por lo que Bellman-Ford también lo detectará

Ahora bien, queda definir la función h. Acá es donde vuelve a aparecer el nodo s. h(k) = dist(s,k).
Luego, $\hat{w}(u,v) = w(u,v) + dist(s,u) - dist(s,v)$ y por desigualdad triangular sabemos que $dist(s,v) \leq dist(s,u) + w(u,v)$ y entonces $0 \leq dist(s,u) + w(u,v) - dist(s,v) = \hat{w}(u,v)$

Al aplicar Bellman-Ford sobre el nodo s, si no existen ciclos de pesos negativos, este encontrará todas las distancias desde s a cualquiera de los vértices. Una vez hecho esto, hay que redefinir el peso de las aristas u→v como dist(s,u) + w(u,v) - dist(s,v). Ahora podemos aplicar Dijkstra, pero hay que tener en cuenta que la longitud del camino mínimo entre u,v que obtuvimos es ŵ(u,v) que equivale a w(u,v)+dist(s,u)-dist(s,v) por lo que, para obtener la longitud w(c) tendremos restarle dist(s,u) y sumarle dist(s,v) al resultado obtenido de cada camino antes de devolverlo.

Si Bellman-Ford encontró un ciclo de longitud negativa, para devolverlo lo que se puede hacer es lo siguiente. Al relajar las aristas por |V|-ésima vez, al menos dos de estas van a tener un peso menor al que tenian en la iteración |V|-1, ya que un ciclo tiene al menos dos aristas. Luego, al hacer esta relajación, al encontrar la primera arista u→v que tiene al relajarla la distancia hasta el vértice v tiene una menor distancia que tenía en la iteración |V|-1, podemos afirmar que hay un camino desde alguno de los vértices

de un ciclo negativo que llega hasta ella y nos guardamos v. Podemos afirmar que v es un vértice que pertenece a un ciclo negativo ya que se encontró la primera arista la cual se puede volver a relajar y esto se da solamente si pertenece a un ciclo negativo, ya que si no perteneciera a un ciclo negativo y al ser la primera que se reduce, la distancia se mantendría igual. Lo que podemos plantear es un sistema de parentescos. Como el grafo es dirigido, u es el padre de v en este caso. Lo que se hace es crear un vector booleanos B de |V| posiciones inicializadas en falsa donde cada posición corresponde un nodo. Y un vector de |V| posiciones P de enteros inicializadas en -infinito. Luego lo que se va a hacer es, desde v (que sabemos que pertenece a un ciclo negativo), en B marcamos la posición correspondiente como true y el número de iteración en P y lo guardamos en un nuevo vector de soluciones S al final. Vamos repitiendo lo mismo iterativamente sobre los padres de todos los vértices y cuando encontramos a uno que ya había visitado en B, devolvemos el vector solución desde la posición P que le corresponde a este vértice hasta el final. El vector P lo tenemos porque puede ocurrir que haya dos ciclos "anidados" de peso negativo que contengan a un vértice, y al recorrerlos con el sistema de parentesco, el recorrido siga hacia el otro ciclo. A lo sumo recorreremos una vez cada vértice, por lo que la complejidad de esto es O(|V|).
El algoritmo sería el siguiente:

```
Johnson(G(V,E))
Agregar a G un nodo s que tiene |V| aristas salientes, una a cada nodo de peso 0 //O(|V|)
Aplicar Bellman-Ford desde s //O(|V+1|*|E|) = O(|V||E|)
Si G tiene ciclos de longitud negativa devolver el ciclo //O(|V|)
Si G no tiene ciclos de longitud negativa:
    Cambiar el peso de todas las aristas por w(u,v)-h(v)+h(u) //O(|V|+|E|)
    Eliminar las aristas y el nodo que se agregaron al principio //O(|V|)
    Para cada nodo aplicar Dijkstra y a la longitud restarle h(u) y sumarle h(v) y devolver
    //O(|V|*log |V|*|E|)
```

El costo total del algoritmo en peor caso es O(|V||E|)+O(|V|*log |V|*|E|) = O(|V||E|+|V|*|E|*log |V|) = O(|V|*|E|*log |V|)

Complejidad espacial

El algoritmo trabaja con un solo grafo, cargado con lista de adyacencias. Su complejidad espacial es O(|V|+|E|). Sin embargo, como el grafo es conexo, |E| > |V| por lo que la complejidad espacial será de O(|E|)

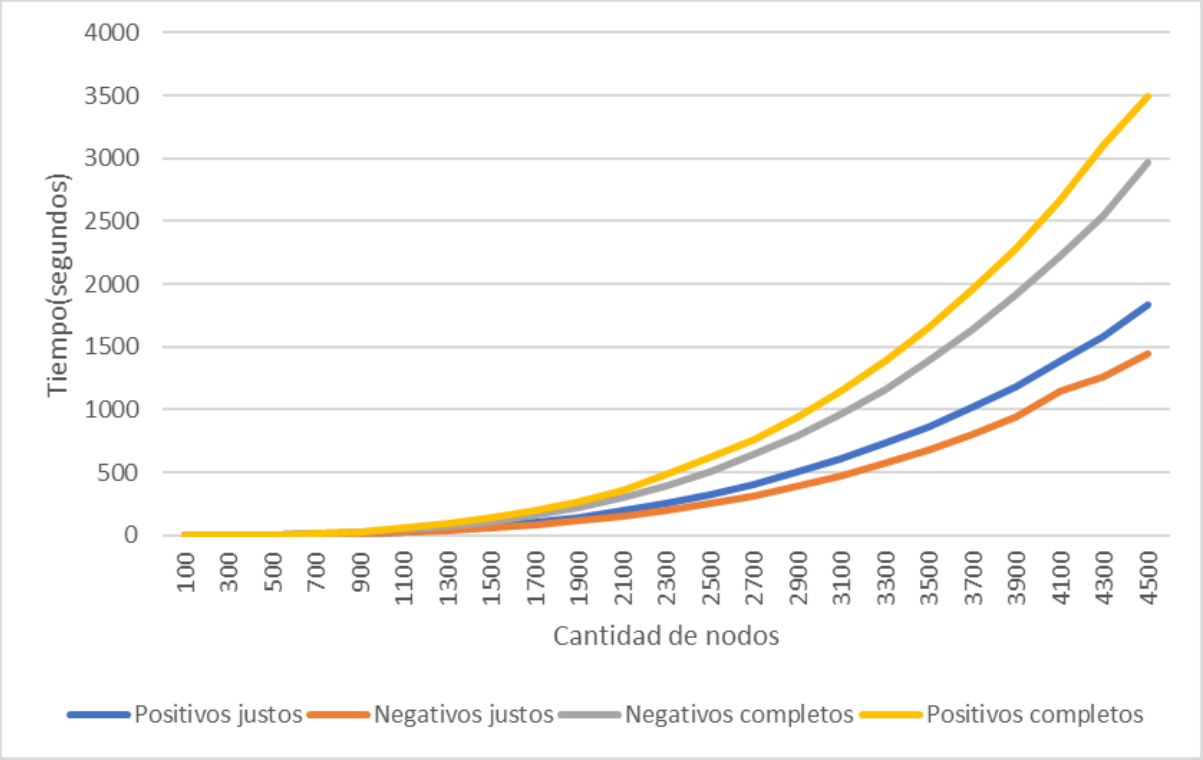
Experimentación

Complejidad Temporal

En este informe se trabajará con grafos conexos. Lo primero que decidimos experimentar fue con los tiempos. Notar que las complejidades entre el caso en el que hay ciclos negativos contra el caso en el que no los hay es distinta, por lo que se midieron los tiempos de ambos por separado. Generamos casos de test aleatorios pero notamos que en la mayoría de los casos, si el grafo tenía aristas negativas, aparecía un ciclo de peso negativo (esto se va a detallar más adelante). En nuestra implementación, para los dos casos, el algoritmo realiza las mismas operaciones tanto si el grafo tiene aristas de peso negativo como si no las tiene, por lo que, para experimentar con los tiempos en el caso de mayor complejidad temporal, decidimos que estos grafos tengan todos aristas positivas, para no tener que buscar exhaustivamente casos de test que lo cumplan. Para observar la complejidad se realizaron 4 experimentos:

- Aristas justas(con una cantidad de hasta |V| aristas demás) para un grafo conexo sin ciclos negativos
- Aristas justas(con una cantidad de hasta |V| aristas demás) para un grafo conexo con ciclos negativos
- Grafo completo sin ciclos negativos
- Grafo completo con ciclos negativos

Y se pudo observar lo siguiente:



Experimentos ejecutados con un CPU Ryzen3 1300X con 8Gb de memoria RAM

Con negativos nos referimos a que el grafo contiene ciclos de longitud negativa y con positivos a que no contiene ciclos de longitud negativa. Justos significa que tiene una cantidad de aristas |E| tal que (|V|-2) * (|V|-1)/2 <= |E| <= (|V|-2) * (|V|-1)/2 + |V| y completos significa que se tiene una cantidad de aristas |E| = (|V|) * (|V|-1).

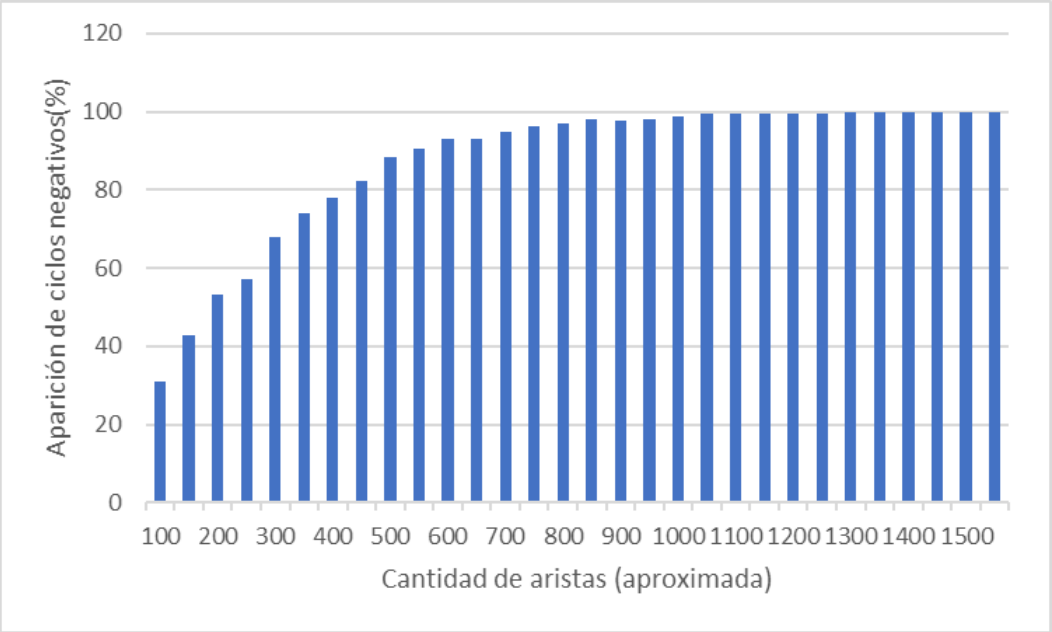
En ambos casos, $|E| = O(|V|^2)$

Podemos observar que dentro de grafos con una cantidad similar de aristas, si se encuentran ciclos negativos, el algoritmo tarda menos que si no se encontraran. Sin embargo, si comparamos los tiempos de los grafos completos contra los de los grafos justos, obtenemos que los completos tardan aproximadamente el doble que sus respectivos grafos justos. Incluso se puede notar que el caso del grafo completo que tiene ciclos negativos, tarda más en ejecutarse para cualquier cantidad de nodos que el que no tiene ciclos negativos para una cantidad de aristas justas. Y aunque la cota nos dice que asintóticamente debería ser mayor para el caso en el que se encuentra un ciclo negativo, podemos observar que estamos en un caso en el que se tienen 4500 nodos(alrededor de 10 millones de aristas para el caso justo y 20 millones para el caso completo), estamos en un caso el cual el tamaño de la entrada es muy grande, lleva mucho tiempo de procesamiento (prácticamente 1 hora para el caso del grafo de 4500 nodos, sin ciclos negativos y completo) y si tomáramos dos grafos con la misma cantidad de nodos cualesquiera (en este caso estamos tomando los extremos pero podría pasar con casos intermedios), nos encontraríamos en una situación en la cual en entradas de gran tamaño, en las que uno espera observar que la complejidad asintótica da una posible guía para ver cuál algoritmo tarda más o cual tarda menos, al ejecutarse nos podríamos llevar una sorpresa.

Generación de grafos

A la hora de la experimentación se quisieron generar grafos aleatorios, pero aparecieron ciertos problemas. La primera aproximación fue que a la hora de decidir qué valor asignar como peso a las aristas, se tenía un

parámetro de probabilidad de que sea negativa Pn, que la decidíamos nosotros, y se generaba un número aleatorio. Si el número aleatorio generado era menor a Pn, entonces el número era negativo. Luego, generamos un entero al azar positivo entre 0 y 50. Lo que nos sucedía con este método es que, incluso para valores de Pn bajo, para grafos con más de 20 nodos, siempre el resultado era negativo, ya que aparecía una arista de gran valor, negativa, en al menos un ciclo, la cual al sumarse con los otros pesos de las aristas de los ciclos a los que pertenecía, daba un resultado negativo. Por lo que se decidió por otro enfoque para generarlos. Ahora la generación de las aristas seguía teniendo el mismo concepto de probabilidad negativa, pero los pesos se generaban con una distribución normal con media 100 y desviación estándar 5, teniendo así una probabilidad del 99,7% de que los valores que se generen tengan módulo entre 85 y 115.
Para ver cómo variaba la aparición de ciclos negativos se realizó el siguiente experimento.
Se generaron 1000 grafos aleatorios con Pn = 1%, media = 100 y desviación estándar = 5, variando la cantidad de nodos y observamos que porcentaje encontraba un ciclo negativo. Sorprendentemente obtuvimos el siguiente resultado:



Notar que a mayor cantidad de aristas, más es el porcentaje de la cantidad de ciclos negativos que aparecen para una misma cantidad de aristas. Además, por lo que se puede observar en el gráfico, esta tendencia parece ser de forma logarítmica.

Demostración de cumplimiento de los tiempos del algoritmo en relación a la cota teórica

Cotas teóricas

Grafos densos

En grafos densos, podemos considerar a $|E| \in O(|V|^2)$. Luego el caso en el que se encuentran ciclos negativos, la complejidad es de $O(|V||E|+|V|) = O(|V||E|)$ (ejecutar Bellman-Ford y encontrar el ciclo negativo). Reemplazando $|E|$ por $|V|^2$ obtenemos que la complejidad del algoritmo para este caso es de $O(|V||V|^2) = O(|V|^3)$. En el caso en el que no se encuentran ciclos negativos, vimos que la complejidad en peor caso es de $O(|V|*|E|*\log |V|)$. Y reemplazando $|E|$ por $|V|^2$ obtenemos que la complejidad es $O(|V|*|V|^2 * \log |V|) = O(|V|^3 * \log |V|)$. Notemos entonces que para este tipo de grafos, la complejidad es peor que usando el algoritmo de Floyd-Warshall.

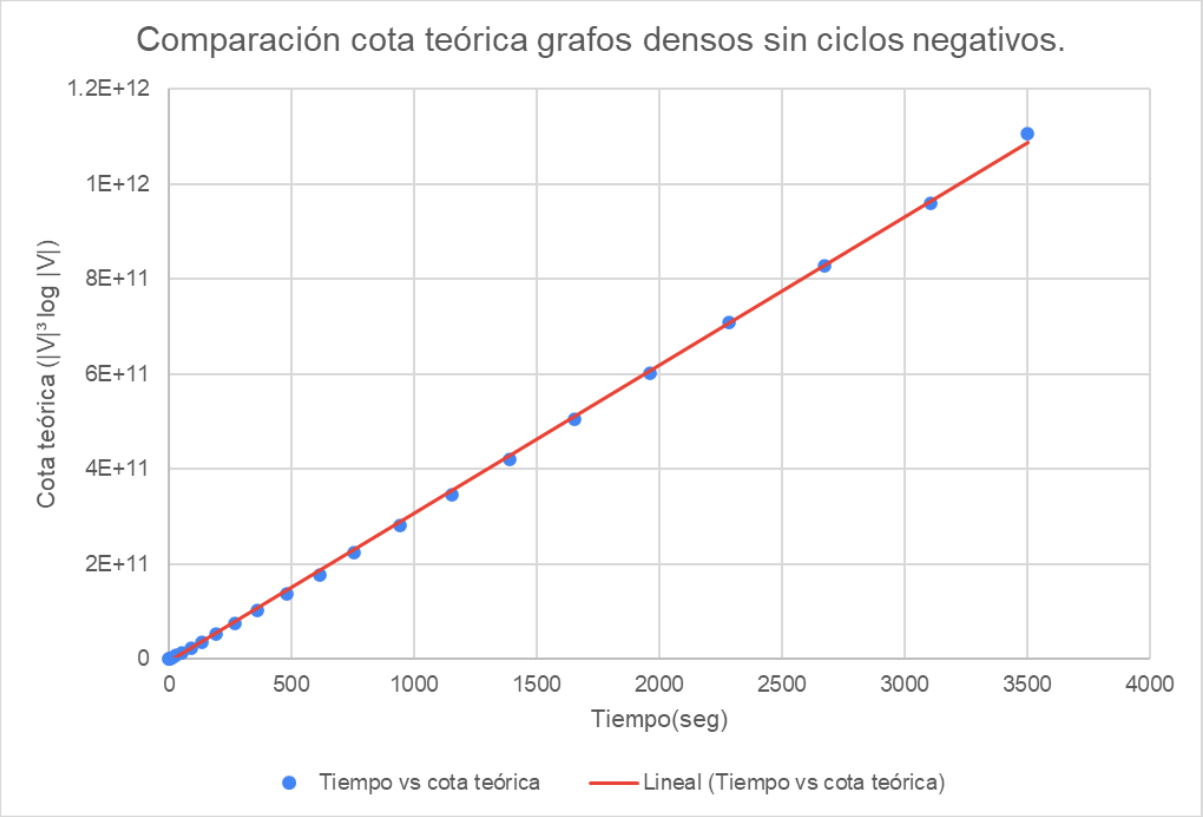
Grafos ralos

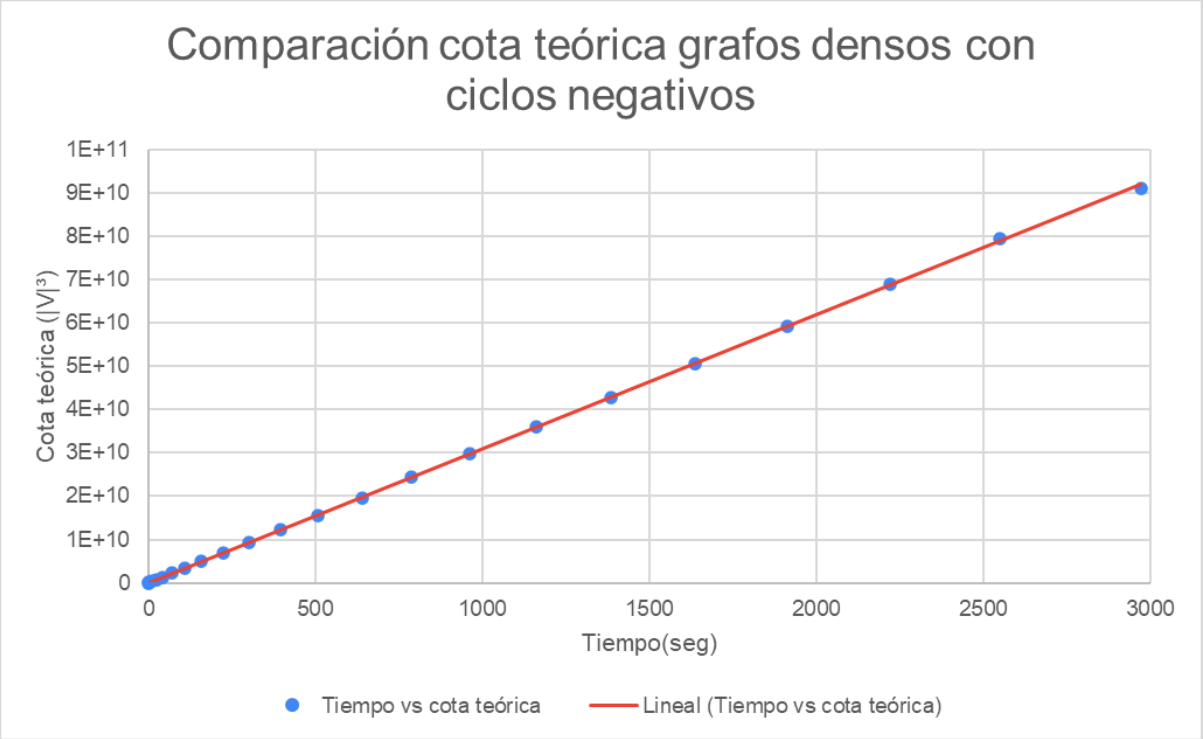
En grafos ralos, podemos considerar a $|E| \in O(|V|)$. Para el caso en el que se encuentra algún ciclo de peso negativo, sabemos que la complejidad temporal es $O(|V||E|)$. Luego la complejidad temporal en este caso es de $O(|V|*|V|) = O(|V|^2)$. Para el caso en el que no se encuentran ciclos negativos, la complejidad es de $O(|V|*|E|*\log |V|)$. En este caso reemplazamos a $|E|$ por $|V|$ y obtenemos que la misma es $O(|V|*|V|*\log|V|) = O(|V|^2*\log|V|)$. Mejorando así la complejidad del algoritmo de Floyd-Warshall.

Experimentación

Para comparar la cota teórica contra los resultados experimentales lo que se hizo fue ejecutar los experimentos con dos tipos de grafos, densos y ralos, variando los nodos, y observar si existía una correlación entre la cota teórica y los tiempos obtenidos para cada uno de los casos y se obtuvieron los siguientes resultados.

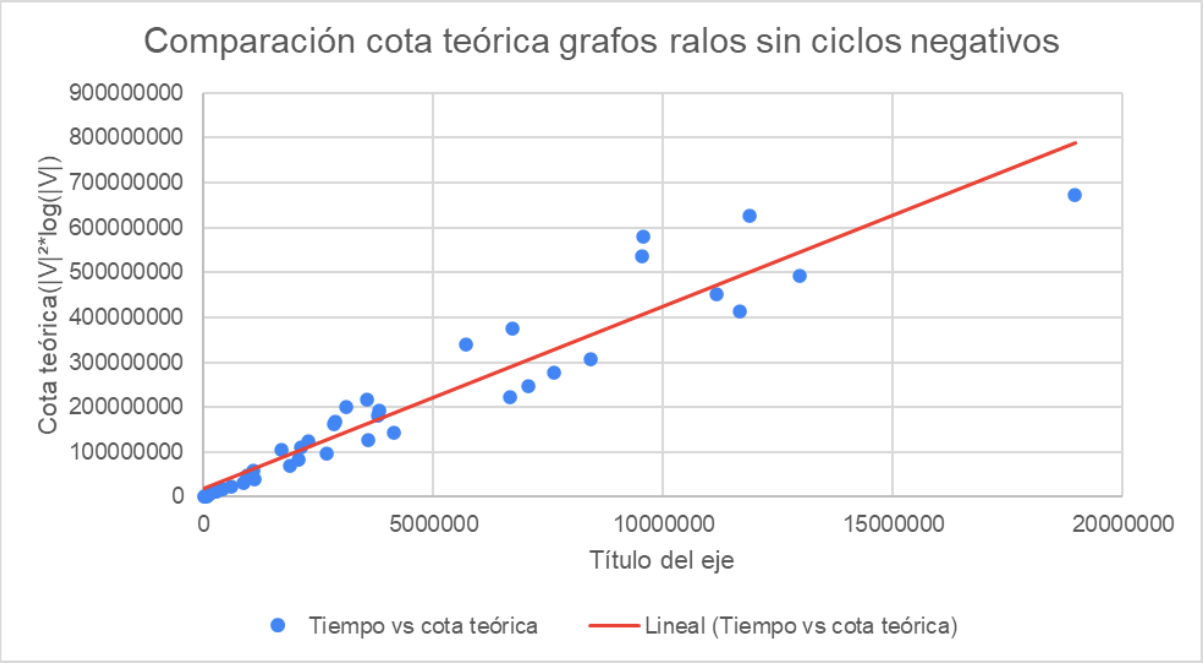
Grafos densos

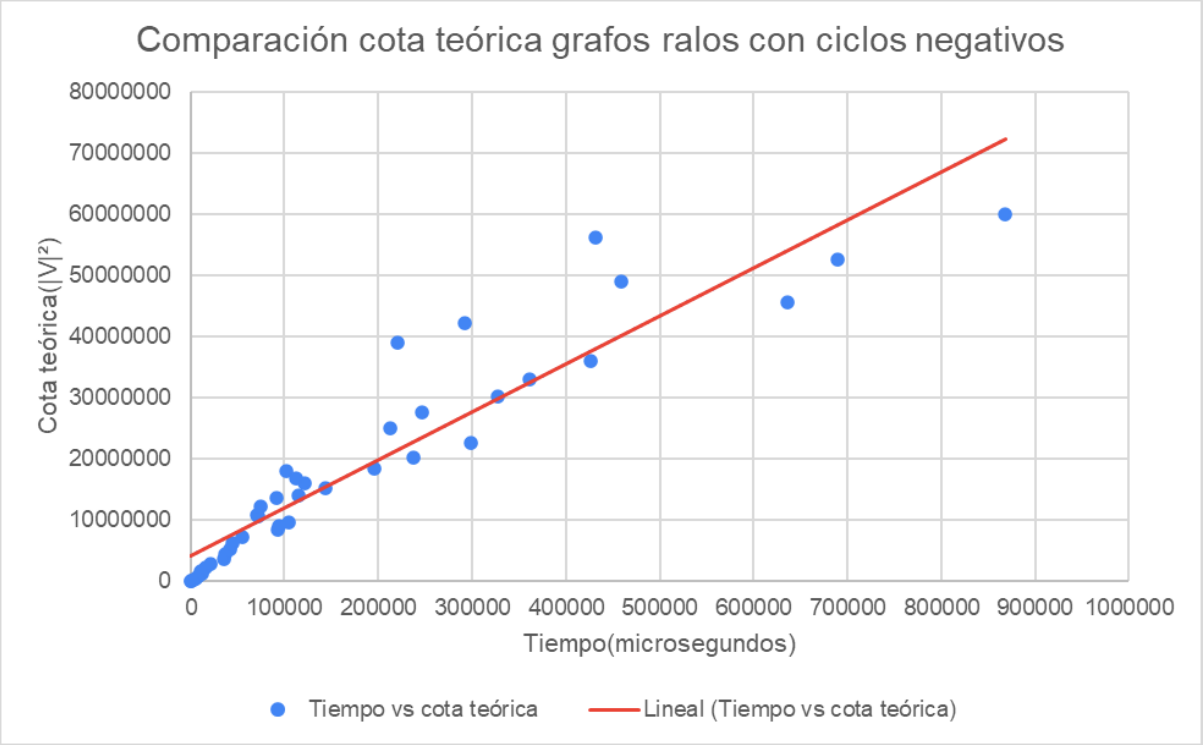




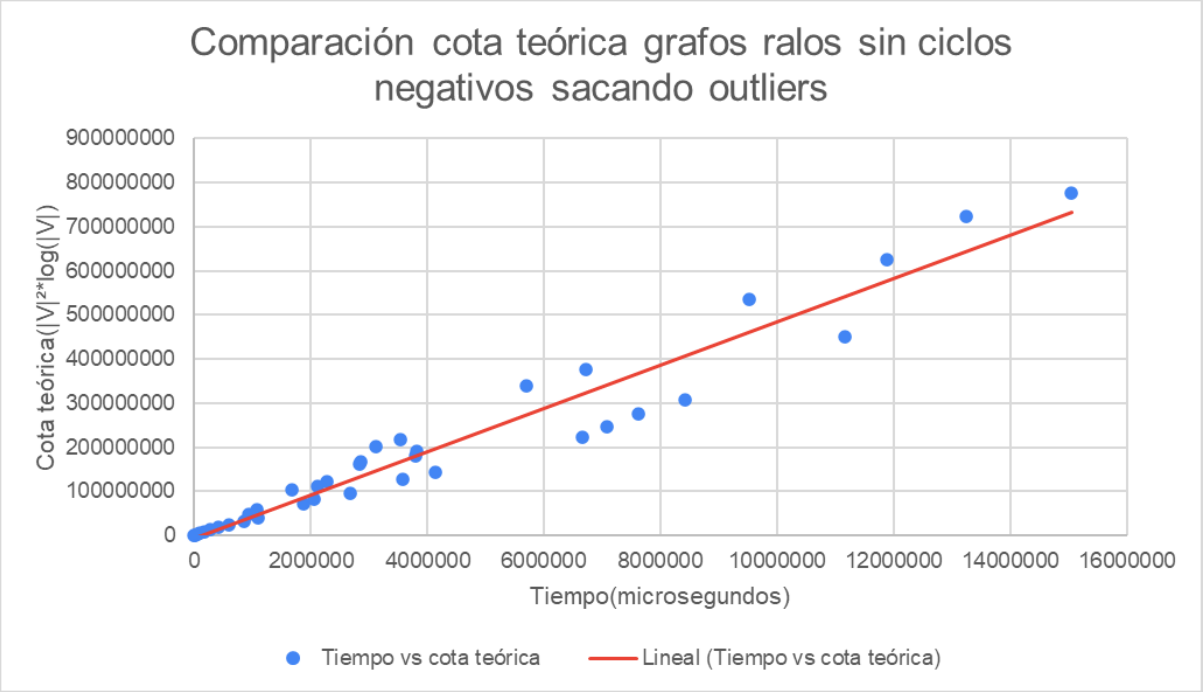
Para los grafos densos, utilizamos los resultados que ya teníamos de los grafos completos. Se puede observar que en todos los casos la regresión lineal se adecúa muy bien a la ubicación de los puntos. Incluso, el coeficiente de correlación de Pearson para el caso sin ciclos negativos es de 0,9998 y para el caso con ciclos negativos es de 0.9999, mostrando así que la correlación casi perfecta entre la cota teórica y los tiempos obtenidos.

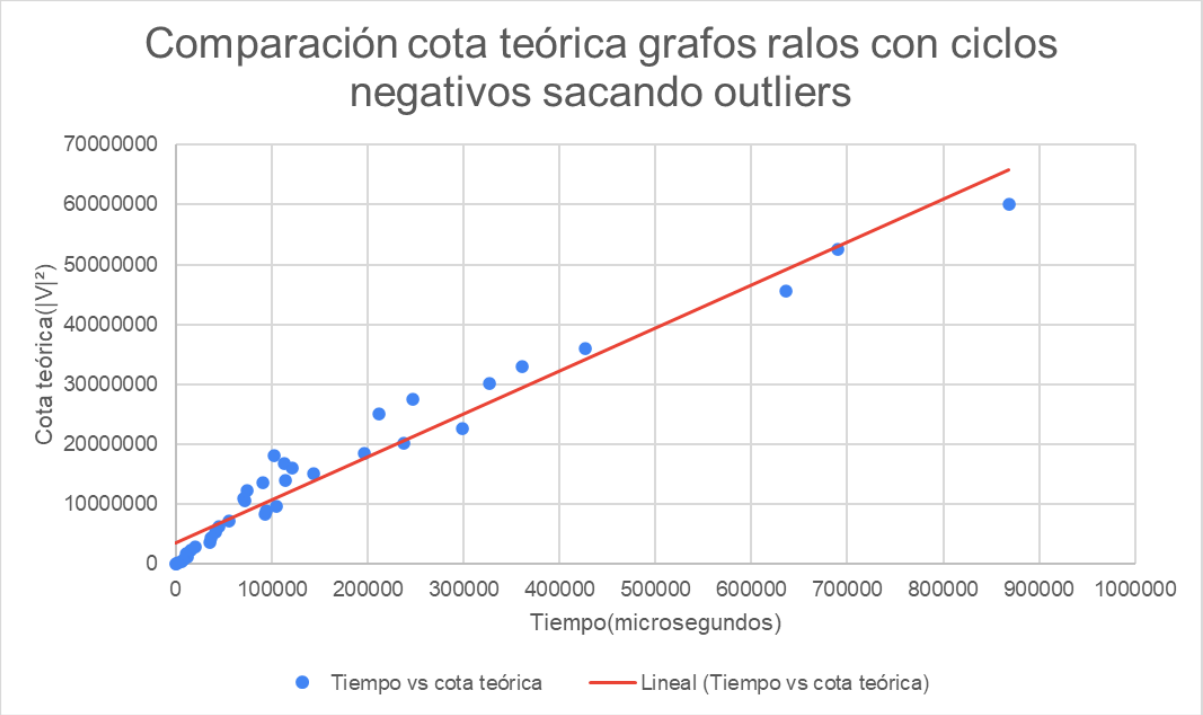
Grafos ralos





Para los grafos ralos, a simple vista no se ve una correlación tan perfecta como la que existía para los grafos densos. Para experimentar con estos, lo que hicimos fue generar grafos ralos de manera aleatoria. Lo que sucedió fue que el algoritmo para generar grafos ralos tardaba demasiado y no se pudieron obtener instancias de una gran cantidad de aristas, como las que se podían obtener en el caso de los grafos densos, de modo que la cota asintótica, en términos generales, no parece estar tan correlacionada con los resultados de los experimentos como en los grafos densos. Sin embargo, el coeficiente de Pearson para el caso de los grafos sin ciclos negativos que obtuvimos fue de 0,9558 y para el caso con ciclos negativos fue de 0,9338, un resultado bastante aceptable. Podemos observar que incluso, en ambos casos hay ciertos outliers que los podemos remover y conseguimos una mejor aproximación lineal. Por ejemplo, a los siguientes gráficos se le removieron 4 puntos a cada uno.





Podemos observar que la aproximación lineal mejora significativamente. Los coeficientes de Pearson que obtenemos para estos casos son de 0,9716 para el caso sin ciclos negativos y 0.9753 para el casos con ciclos de peso negativo, notando así una mejora frente a lo que teníamos antes.

Conclusión

En los gráficos se puede ver entonces la existencia de una relación lineal entre el valor de la cota teórica y el resultado de los experimentos evaluados sobre una misma cantidad de nodos. A medida que aumentamos esta variable se puede ver como aumenta el tiempo de ejecución del algoritmo a la par con la función de complejidad teórica. Esta fuerte correlación entre los resultados del algoritmo y valor teórico de la función de complejidad evidencian que el algoritmo cumple con la cota teórica esperada.

Algoritmo de Bertossi

Presentación del problema

El paper de Alan Bertossi "Total Domination in Interval Graphs" presenta un algoritmo para encontrar conjuntos dominantes totales de cardinalidad mínima en un grafo intervalo. En general el problema de encontrar un CDT (conjunto dominante total) de mínima cardinalidad en la mayoría de los grafos es NP-completo. Bertossi introduce un algoritmo $O(n^2)$ para encontrar un CDT en un tipo particular de grafos: los grafos intervalos.

Los grafos intervalos son aquellos construidos en base a una familia de intervalos $I = \{I_1, \ldots, I_{n+1}\}$. Un intervalo $I_i \in I$ está definido por sus extremos a, b. En nuestro caso los intervalos estarán incluidos en $[0, 2n)$ con todos los intervalos de distinto valor. Luego un grafo G es un **grafo intervalo** si y sólo si hay una relación uno a uno entre los intervalos de I y los nodos del grafo tal que dos nodos son adyacentes si y sólo si sus intervalos se intersecan.

Un **CDT** de un grafo intervalo $G(I)$ es un subconjunto S de intervalos en I tal que todo intervalo en I se interseque con algún intervalo en S. Esto también debe cumplirse para los intervalos en S. Para este algoritmo solo se tendrán en cuenta grafos conexos ya que si el grafo es disconexo el problema se reduce a encontrar un CDT en cada una de las componentes conexas con el procedimiento que será explicado a continuación.

La idea de Bertossi es reducir el problema a uno de camino mínimo. Es decir, se presenta una manera de modelar un grafo intervalo como un grafo acíclico tal que si se ejecutan cualquier algoritmo de camino mínimo los nodos que formen parte de este serán los intervalos que también son parte del CDT.

Estructuras de Representación

Los **intervalos** son representados por una estructura que guarda sus extremos y además un índice que corresponderá al orden en el que fue pasado por parámetro y luego, en caso de formar parte del grafo, a su índice dentro del conjunto de nodos de $G(I)$.

Un **vecino** de un nodo del grafo está definido con un par (Intervalo, peso).

Una **arista** de $G(I)$ está representada por una estructura tal que su primer elemento será la cola, el segundo la cabeza y por último el peso de la arista.

Finalmente el **grafo** está representado como una lista de adyacencia en la que la i-esima lista va a coincidir con el índice del i-ésimo intervalo en el conjunto de nodos del grafo.

El Algoritmo

```
Algoritmo Bertossi(in n: N, in I: conj(Intervalos)) -> out CDT: conj(Nat)
    // Caso Trivial:
        Si el mínimo a y el máximo b coinciden en intervalo. // O(n)
            Devolver el índice de ese intervalo con cualquier otro como CDT. // O(1)
    // Caso No Trivial:
        Ordenar(I). // O(nlog(n))

        Crear el conjunto N de nodos de D. // O(1)
        N U {I_0, I_n+1}. // O(1)
        Para cada Ii ∈ I: // O(n) * O(n) = O(n²)
            Si Ii no está completamente contenido en un intervalo Ij en I: // O(n) * O(1) = O(n)
                N U {Ii} // O(1)

        Crear el conjunto intermedio de aristas B y C. // O(1)
        Para cada Ii ∈ N: // O(n) * 2O(n) = 2O(n²) = O(n²)
            Si se interseca Ii con un intervalo Ij  N: // O(n)
                B U {(Ii, Ij)}. // O(1)
            Si no se intersecan Ii con un intervalo Ij  N y no hay Ih  I tq Ii.a < Ih.a < Ih.b < Ij.b: // O(n)
                C U {(Ii, Ij)}.

        Divido los nodos en N que pertenecen también a I en dos, I_in e I_out. // O(n)

        Crear el conjunto A de aristas de D. // O(1)
        Para cada nodo Ii ∈ I: // O(n)
            A U {(Ii_in, Ii_out)} con peso 0.
        Para cada arista (u, v)  B: // O(m) = O(n²)
            la actualizamos por (u_out, v_in) con peso 1.
        Para cada arista (u, v)  C: // O(m) = O(n²)
            Si u = I_0 // O(1)
                la actualizamos por (I_0, v_out) con peso 0.
            Si v = I_n+1 // O(1)
                la actualizamos por (u, I_n+1) con peso 1.
            Sino  // O(1)
                la actualizamos por (u_in, v_out) con peso 1.
        A U (B U C) // O(1)

        D <- Grafo(N, A). // O(n²)

        Caminos <- DAGSSP(D, I_0). // O(n + m) = O(n²)
        CDT <- ArmarCaminoMinimo(I_0, I_n+1, Caminos). // O(n + n*log(n)) = O(n)

        Devolver CDT. // O(1)
```

El algoritmo se puede dividir en dos casos: el trivial y el no trivial. Comencemos por el primero.

Caso trivial: hay un intervalo que contiene completamente a todo el resto.
Llamamos trivial al caso en el que existe un intervalo $I_i \in I$ tal que contiene a todos los demás. Luego el CDT de cardinalidad mínima asociado a ese $G(I)$ será el conjunto que contenga al intervalo I_i y a cualquier otro intervalo de I. Este caso se puede detectar en $O(n)$ viendo si el $a_i = min_{1 \leq k \leq n}(a_k)$ y $b_j = max_{1 \leq k \leq n}(b_k)$ coinciden en su intervalo.

Caso no trivial: no hay un intervalo que contiene completamente a todo el resto.
Por otro lado, el caso no trivial es en el que I no contiene un intervalo que contiene a todos los otros. Veamos cómo se resuelve. Primero ordenamos I según a creciente en $O(nlog(n))$. Creamos un conjunto de intervalos N que va a representar al conjunto de vértices de D, nuestro grafo intervalo dirigido. Agregamos en N dos intervalos de "mentira": I_0, I_{n+1}. La idea es que estos intervalos delimiten el principio y fin del camino mínimo pero no formarán parte del CDT. El extremo b de I_0 tiene que ser menor a todo extremo a de los intervalos en I y el extremo a de I_{n+1} tiene que ser mayor a todo extremo b de los intervalos en I. Como los $I_i \in I$ están entre $[0, 2n)$, se puede establecer que $I_0 = [-2, -1]$ y $I_{n+1} = [2n, 2n + 1]$. Además tendrán índices *0* y $2K + 1$ respectivamente. Como siguiente paso iteramos sobre I separando los intervalos que están contenidos completamente dentro de otro de los que no, agregando estos últimos a N. Esto toma $O(n^2)$. Cada vez que agregue un intervalo a N se va a actualizar su índice con el valor actual de K indicando que es el k-esimo intervalo que se agregó a N. Además voy a agregar su

índice original a un vector P tal que en la posición $K - 1$ se encuentre el índice original. Esto se debe a que se debe devolver el índice que corresponde al i-ésimo intervalo pasado por parámetro que forme parte del CDT de mínima cardinalidad. Esta manipulación de vector toma $O(1)$.
Lo siguiente es armar los conjuntos de aristas B y C. B representa el conjunto de intervalos en N que se intersecan. Es decir $(i,j) \in B \Leftrightarrow a_i < a_j < b_i < b_j$. C contiene a los intervalos en N que no se intersecan y además no hay un intervalo completo entre dos. Es decir, $(i,j) \in C \Leftrightarrow b_i < a_j \wedge \nexists\ I_h \in I\ tq'\ b_i < a_h < b_h < a_j$. El algoritmo "naive" para construir C tiene costo $O(n^3)$ que arruina la complejidad descrita por Bertossi. Para eso construimos al vector $intervaloAsociado$ que va tener en su i-ésima posición al primer $I_a \in I$ que empiece y termine luego del fin de un intervalo $I_i \in N - \{I_{n+1}\}$. No tomamos en cuenta al I_{n+1} ya que por definición ningún intervalo finaliza después de este. Construir este vector toma $O(n^2)$ y acceder a las posiciones dentro del mismo $O(1)$. Por lo tanto para construir ambos conjuntos no queda más que para cada intervalo $I_i \in N$ recorrer N fijándose si para algún $I_j \in N$ vale que (I_i, I_j) pertenece a alguno de B, C o ninguno de los dos. Esto toma $O(n^2)$.
El siguiente paso es introducir para todo intervalo $I \in N$ los nodos I_{in}, I_{out} . Es decir, el algoritmo va a dividir cada nodo $I_i \in N - \{I_0, I_{n+1}\}$ en dos tal que los extremos de $I_{i_{in}}, I_{i_{out}}$ y el índice de I_{in} van a coincidir con los valores de I_i. No obstante el índice de I_{out} va a ser igual a $K + indice_{i_{in}}$. Esto se debe a que en D queremos que los vértices estén ordenados de la siguiente forma: primero I_0, luego los nodos I_{in}, los nodos I_{out} y por último I_{n+1}. A su vez, por cada nodo $I_{i_{in}}, I_{i_{out}}$ voy a introducir una arista $(I_{i_{in}}, I_{i_{out}})$ de peso 0. Además vamos a actualizar a P agregando en la posición que coinciden con el índice de $I_{i_{out}}$el mismo valor que hay en la posición igual al índice de $I_{i_{in}}$en P. Es decir quiero que ambos devuelvan el índice del I_i original. Esto se debe a que el CDT que quedará definido al ejecutar el algoritmo de camino mínimo contendrá a un intervalo $I_i \in I$ si y sólo si el camino pasa por alguno de $I_{i_{in}}, I_{i_{out}}$ o ambos. Esta sección tiene una complejidad $O(n)$ ya que requiere duplicar la cantidad de intervalos. Actualizar los valores en P tiene costo $O(1)$.
Luego actualizamos los conjuntos B, C definidos anteriormente. Para cada arista $(i,j) \in B$ voy a agregar en la lista de adyacencia del vértice i_{out} a j_{in} con peso 1. En cambio para cada arista $(i,j) \in C$ agregamos en la lista de adyacencia del vértice i_{in} a j_{out} con peso 1. Por último, para cada arista $(I_0, i) \in C$ agregamos en la lista de adyacencia del nodo I_0 al vértice i_{out} con peso 0 mientras que para las aristas (i, I_{n+1}) agregamos en la lista de adyacencia del nodo i_{in} a I_{n+1} con peso 1. Al ser un grafo conectado dirigido tenemos que la cota superior para la cantidad de arista en D es $(n-1) * n \in O(n^2)$. Luego actualizar las aristas pertenece al orden de $O(n^2)$.
Al tener construido el grafo intervalo D lo único que queda es ejecutar un algoritmo de camino mínimo sobre el mismo con inicio en el intervalo I_0. En nuestro caso elegimos el $DAGSP$ ya que los intervalos en $I \cup \{I_0, I_{n+1}\}$ están ordenados según a creciente y por cada arista $(i,j) \in A$ tenemos que $a_i < a_j$. Luego D es acíclico. Ejecutar el algoritmo de camino mínimo toma $O(n+m)$ pero dado que m está acotado por n^2 el $DAGSP$ termina perteneciendo a $O(n+n^2) = O(n^2)$. El camino que nos interesa es el que finaliza en el nodo I_{n+1}. Luego generamos ese recorrido a partir de la función $ArmarCaminoMinimo$ que recorre el arreglo de padres asociado a la ejecución del $DAGSP$ previa. Esta función tiene complejidad $O(n)$.
Finalmente creamos el CDT a partir de los índices asociados a los intervalos que forman el recorrido del camino mínimo entre I_0 e I_{n+1}. Para eso recurrimos al vector P creado previamente donde tenemos los índices originales. El acceso al mismo es constante. En caso de que el camino contenga tanto al I_{in} como el I_{out} de un mismo intervalo el CDT al ser un conjunto se quedará con solo uno de ellos por lo tanto no habrá intervalos repetidos en la solución. Debido a esto el CDT está implementado como un conjunto y no como un vector. Por lo tanto construir agregando los índices de los intervalos del camino mínimo tiene complejidad $O(nlog(n))$.
Es sencillo ver que la construcción del grafo intervalo D es $O(n^2)$ como también lo es ejecutar el algoritmo de camino mínimo. Construir el CDT toma $O(nlog(n))$ al estar implementado sobre un conjunto. Luego la complejidad total de la implementación del Algoritmo de Bertossi toma $O(n^2)$ como se detalla en el paper.

Apéndice

Las siguientes son las tablas que representan la performance temporal y el resultado de los algoritmos propuestos ante distintos escenarios. Algunos de los tests utilizados provienen del repositorio '*la-plebe/1C2022-AED3-TP2-tests*', un conjunto de tests comunitarios compartidos por compañeros de la materia.

Test	Geodésico	Nodos	Aristas	Tiempo de ejecución (seg)
test_NOG_50.txt	Si	50	53	0.000235s
test_NOG_100.txt	No	100	104	0.000384s
test_NOG_200.txt	No	200	203	0.000776s
test_NOG_300.txt	No	300	303	0.001209s
test_NOG_400.txt	No	400	402	0.001639s
test_NOG_500.txt	No	500	504	0.002239s
test_completo_50.txt	Si	50	1225	0.006930s
test_completo_100.txt	Si	100	4950	0.048019s
test_completo_200.txt	Si	200	19900	0.310262s
test_completo_500.txt	Si	500	124750	4.455196s
test_tree_50.txt	Si	50	49	0.001207s
test_tree_100.txt	Si	100	99	0.004506s
test_tree_200.txt	Si	200	199	0.016495s
test_tree_500.txt	Si	500	499	0.102786s
pentatopeG.txt	Si	5	10	0.000120s
trivial0G.txt	Si	1	0	0.000100s
trivial1.txt	Si	2	1	0.000087s
lollipopNOG.txt	No	5	6	0.000108s
test3NOG.txt	No	24	42	0.000608s

Tabla 1: Subconjunto de tests representativos para el Ejercicio 1. Este conjunto de tests fue corrido con un procesador Intel i5-9400F con 16 GB de memoria RAM.

Nombre de Test	M	N	Óptimo	Resultado	Tiempo (seg.)
frantst2.in	19	20	30	30	0.00
frantst10.in	89	28	191	191	0.00
frantst17.in	99	46	326	326	0.00
frantst18.in	85	89	535	535	0.00
solido-10.in	10	10	1	1	0.00
stripes-v-10.in	10	10	5	5	0.00
T-80.in	80	100	1	1	0.00
sus.in	25	25	16	16	0.00
big-frantst1.in	1419	377	36049	36049	0.07
big-frantst2.in	2437	1285	207162	207162	0.36
big-frantst6.in	2032	1713	229833	229833	0.40
big-frantst3.in	3935	938	244342	244342	0.44
big-frantst17.in	1142	3361	253044	253044	0.46
big-frantst4.in	3903	1663	427408	427408	0.73
big-frantst7.in	4165	4556	1249500	1249500	2.13

Tabla 2: Subconjunto representativo de resultados de tests para el Ejercicio 2, para probar el algoritmo se utilizaron todos los grafos en el repositorio de tests adjuntado. Los casos fueron elegidos en función de la variabilidad de los parámetros *m* y *n*. Los tests fueron ejecutados con un procesador AMD Ryzen 5 3500U.

Para el algoritmo de Johnson generamos nuestros propios test. Los resultados que se obtuvieron fueron los siguientes para medir los tiempos:

Cantidad de nodos	Tiempo positivos justos(seg)	Tiempo negativos justos(seg)	Tiempo positivos completos (seg)	Tiempo negativos completos (seg)
100	0.05	0.02	0.08	0.06
300	0.75	0.49	1.23	0.91
500	3.05	2.02	5.3	4.11
700	8.01	5.5	14.22	11.25
900	16.41	11.85	29.41	23.74
1100	29.24	21.72	53.5	43
1300	48.34	34.76	90.72	71.28
1500	71.67	53.56	133.21	108.48
1700	102.94	78.23	193.2	158.33
1900	143.09	109.98	268.51	222.32
2100	192.2	145.99	361.55	299.77
2300	252.79	194.38	478.99	393.59
2500	321.39	250.65	616.56	505.32
2700	399.58	310.64	755.08	639.35
2900	501.16	385.48	944.61	787.6
3100	610.31	473.11	1151.85	961.52
3300	729.64	577.68	1391.24	1160.62
3500	860.59	676.87	1653.36	1385.62
3700	1,017.32	806.35	1960.11	1637.47
3900	1,178.33	936.51	2283.35	1912.53
4100	1,384.52	1149.48	2672.52	2221.09
4300	1,579.37	1257.69	3103.77	2547.37
4500	1,833.36	1442.67	3500.1	2971.6

Tabla 3: Casos de test para el algoritmo de Johnson. Se ejecutaron con un CPU Ryzen 3 y 8GB de memoria RAM. En pos de mantener la prolijidad en la tabla, omitimos el nombre de los archivos. Estos se encuentran en la carpeta … y ahí se detalla el formato del título de los casos.

Nombre de Test	N	Tiempo(seg)
contained-2.in	6	0.000037
contained-1.in	4	0.000039
big-100.in	100	0.000125
big-sg-100.in	100	0.000213
big-500.in	500	0.000901
big-sg-500.in	500	0.001163
big-1500.in	1500	0.003822
big-sg-1000.in	1000	0.004957
big-1000.in	1000	0.008620
big-sg-1500.in	1500	0.010713
big-10000.in	10000	0.063318
big-sg-10000.in	10000	0.193517
big-20000.in	20000	0.198699
big-25000.in	25000	0.262755
big-sg-25000.in	25000	0.859191
big-sg-20000.in	20000	0.873566

Tabla 4: Casos de test para el algoritmo de Bertossi. Se ejecutaron con un CPU Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz y 8GB de memoria RAM.

Tablas de tiempos

Grafos completos sin ciclos negativos		
Cantidad de Nodos	**Tiempo (seg.)**	**Cota teórica($\|V\|^3 * \log\|V\|$)**
100	0.08	6643856.19
300	1.23	222178104.6
500	5.3	1120723036
700	14.22	3241765411
900	29.41	7154246488

1100	53.5	13447476073
1300	90.72	22726418110
1500	133.21	35608770401
1700	193.2	52722970399
1900	268.51	74706744420
2100	361.55	1.02206E+11
2300	478.99	1.35874E+11
2500	616.56	1.76371E+11
2700	755.08	2.24361E+11
2900	944.61	2.80518E+11
3100	1151.85	3.45518E+11
3300	1391.24	4.20041E+11
3500	1653.36	5.04773E+11
3700	1960.11	6.00406E+11
3900	2283.35	7.07632E+11
4100	2672.52	8.27149E+11
4300	3103.77	9.59659E+11
4500	3500.1	1.10587E+12

Grafos completos con ciclos negativos		
Cantidad de Nodos	**Tiempo (seg.)**	**Cota teórica($\|V\|^3$)**
100	0.06	1000000
300	0.91	27000000
500	4.11	125000000
700	11.25	343000000
900	23.74	729000000
1100	43	1331000000
1300	71.28	2197000000

1500	108.48	3375000000
1700	158.33	4913000000
1900	222.32	6859000000
2100	299.77	9261000000
2300	393.59	12167000000
2500	505.32	15625000000
2700	639.35	19683000000
2900	787.6	24389000000
3100	961.52	29791000000
3300	1160.62	35937000000
3500	1385.62	42875000000
3700	1637.47	50653000000
3900	1912.53	59319000000
4100	2221.09	68921000000
4300	2547.37	79507000000
4500	2971.6	91125000000

Grafos ralos sin ciclos negativos		
Cantidad de Nodos	**Tiempo (ms)**	**Cota teórica (|V|²*log|V|)**
100	4382	66438.5619
300	22933	740593.6821
500	59317	2241446.071
700	102258	4631093.445
900	170035	7949162.765
1100	272128	12224978.25
1300	419727	17481860.08
1500	594329	23739180.27

1700	862780	31013512
1900	1096005	39319339.17
2100	942268	48669525.63
2300	1077341	59075641.99
2500	1876713	70548202.37
2700	2059683	83096841.51
2900	2689970	96730450.73
3000	1687381	103956721.1
3100	2124107	111457284.5
3250	2279416	123224491
3300	3585081	127285045.9
3500	4141929	144220955.3
3700	2837246	162271807.8
3750	2863440	166959490.5
3900	3797099	181444020.4
4000	3813231	191452548.6
4100	3112524	201743671.7
4250	3548031	217711776.2
4300	6659534	223176536.3
4500	7075374	245748113
4750	7620532	275571872.4
5000	8422408	307192809.5
5250	5705924	340620178.3
5500	6715353	375862781.1
5750	11678802	412929010.1
6000	11162694	451826884.3
6250	12961767	492564081
6500	9527987	535147964.1
6750	9553907	579585608.3
7000	11883896	625883821.1
7250	18956050	674049162.5
7500	13244245	724087962

7750	15050211	776006334.5

Grafos ralos sin ciclos negativos		
Cantidad de Nodos	**Tiempo (ms)**	**Cota teórica ($\|V\|^2$)**
100	154	10000
300	864	90000
500	3012	250000
700	5769	490000
900	8065	810000
1100	12115	1210000
1300	10584	1690000
1500	16202	2250000
1700	20726	2890000
1900	35225	3610000
2100	36743	4410000
2300	42031	5290000
2500	44867	6250000
2700	55155	7290000
2900	93409	8410000
3000	94147	9000000
3100	105109	9610000
3250	71929	10562500
3300	71304	10890000
3500	74474	12250000
3700	91427	13690000
3750	114761	14062500
3900	144102	15210000
4000	121499	16000000
4100	112949	16810000

4250	102462	18062500
4300	196262	18490000
4500	237409	20250000
4750	298262	22562500
5000	212410	25000000
5250	246849	27562500
5500	327512	30250000
5750	361080	33062500
6000	426541	36000000
6250	221054	39062500
6500	292302	42250000
6750	636111	45562500
7000	458908	49000000
7250	689455	52562500
7500	431701	56250000
7750	868281	60062500

Las filas marcadas en rojo corresponden a los elementos que se quitaron para mejorar la aproximación lineal.