DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio de Trabajo Práctico II

Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Ituarte, Joaquin	457/13	joaquinituarte@gmail.com
Oller, Luca	667/13	ollerrr@live.com
Otero, Fernando	424/11	fergabot@gmail.com
Arroyo, Luis	913/13	luis.arroyo.90@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Contents

1 Introducción

En este informe explicaremos el desarrollo del código realizado para resolver los problemas pertenecientes al Trabajo Práctico 2 y la justificación de la complejidad obtenida de cada ejercicio.

Los problemas serán resueltos utilizando las técnicas algorítmicas para el modelado de grafos y resolución de problemas de arboles generadores mínimos, y de caminos mínimos.

El código está desarrollado en C++ y los gráficos los generamos a partir de diversas librerías para la creación de graficos en lenguaje python.

Es importante destacar además que a medida que se resolvieron los problemas, las soluciones fueron probadas para verificar el funcionamiento del código que serán presentadas como tests.

Para justificar la complejidad de los algoritmos que resuelven los problemas se utiliza pseudocódigo y luego una justificación de la complejidad del algoritmo. Por último, se realizará una experimentación para comparar el funcionamiento del algoritmo con la complejidad estimada.

2 Problema 1: Delivery óptimo

2.1 Introducción

Se nos presenta un problema en el que una empresa de logística tiene que transportar mercadería en una provincia cuyas ciudades están conectadas mediante rutas, algunas de las cuales son rutas de categoría "premium". Por resolución de la provincia, sólo se puede utilizar un número determinado de rutas premium.

El problema consiste luego de encontrar el camino más corto entre dos ciudades si se tiene permiso para utilizar a lo sumo "k" rutas premium.

2.2 Resolución del problema

Para resolver este problema decidimos modelarlo como un problema de grafos. Consideramos las ciudades como nodos y las rutas van a ser las aristas con un peso asociado a la distancia entre las ciudades. Luego nuestro problema consiste en encontrar el camino más corto desde un nodo a otro, teniendo en cuenta que sólo se pueden utilizar un número determinado de rutas premium.

Para resolver el problema de usar un número fijo de rutas premium, decidimos realizar lo siguiente: Primero armamos el grafo correspondiente a las ciudades y las rutas que nos plantean. Luego, si nuestro límite de usar las rutas premium es "k", copiamos el grafo resultante k veces, obteniendo k+1 grafos iguales. Finalmente vamos a eliminar las rutas premium de los grafos y para cada ruta premium que va desde la ciudad "a" hasta la ciudad "b" vamos a conectar con una arista dirigida del mismo peso al nodo que representa a la ciudad a de cada i-ésimo grafo con el nodo que representa a la ciudad "b" en el i+1-ésimo grafo, y lo mismo con un eje dirigido que va desde "b" hacia "a". Para el k-ésimo grafo no realizamos estas conexiones pero si removemos las rutas premium.

Una vez realizado esto, nos va a quedar un grafo con n$*$ (k+1) nodos, en donde, partiendo desde un nodo de origen del 0-ésimo grafo, podemos utilizar el algoritmo de dijkstra para calcular la distancia mínima hacia todos los demás nodos, teniendo el siguiente significado: para un nodo de destino cualquiera, la menor distancia obtenida es la menor distancia posible de forma que si el nodo pertenece al i-ésimo grafo, entonces se utilizaron "i" rutas premium. Luego para encontrar la distancia mínima para ir a la ciudad que representa ese nodo, debemos buscar el valor mínimo entre los k+1 nodos que representan a esa ciudad, uno por cada copia del grafo que se realizó.

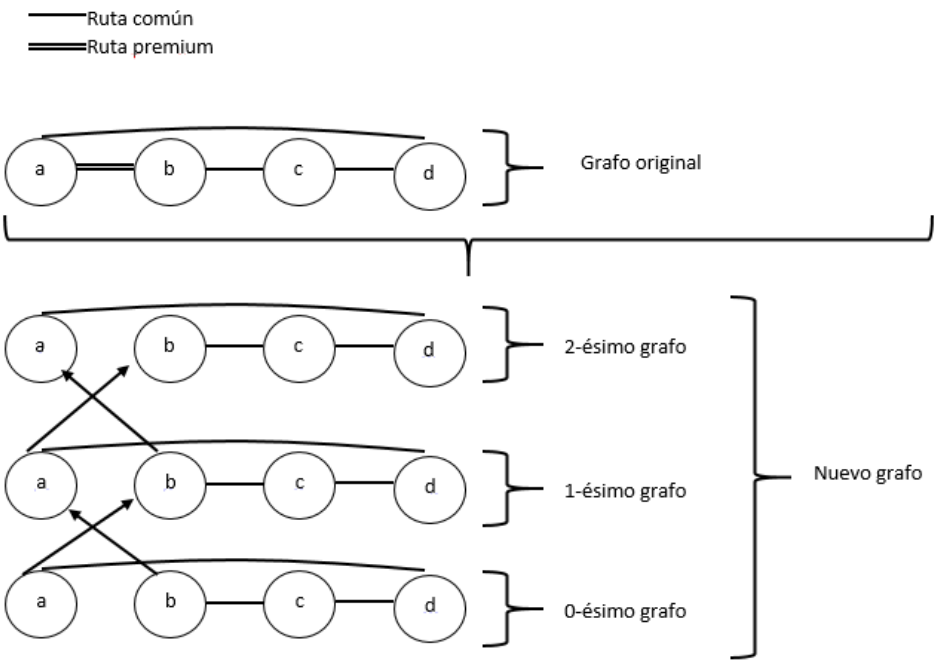


Figure 1: Ejemplo de transformación de un grafo con $k = 2$.

2.3 Pseudocódigo

Algorithm 1 DistanciaMinima

```
1:  procedure DistanciaMinima(graph provincia, int origen, int destino ) → int
2:      List<graph> grafos = copiar_k+1_veces(provincia)
3:      borrar_rutas_premiun(grafos)
4:      for Cada ruta premium entre ciudades a y b con peso p do
5:          for  Cada i-ésima copia del grafo provincia de la lista "grafos", excepto la k-ésima copia  do
6:              nodo_a_piso_i = nodo_que_representa_a(grafos[i], a)
7:              nodo_a_piso_i+1 = nodo_que_representa_a(grafos[i+1], a)
8:
9:              nodo_b_piso_i = nodo_que_representa_a(grafos[i], b)
10:             nodo_b_piso_i+1 = nodo_que_representa_a(grafos[i+1], b)
11:
12:             nodo_a_piso_i.agregar_eje_dirigido_hacia(nodo_b_piso_i+1, p)
13:             nodo_b_piso_i.agregar_eje_dirigido_hacia(nodo_a_piso_i+1, p)
14:         end for
15:     end for
            ▷ la lista grafos ahora representa a un grafo. Ahora aplicamos Dijkstra comenzando desde el
    nodo origen del 0-ésimo grafo.
16:
17:     dijkstra(grafos, origen)
18:     int resultado = min_{0≤i≤k}(distancia_minima_obtenida(nodo_que_representa_a(grafos[i], destino))
19:
20:     return resultado
21: end procedure
```

2.4 Correctitud

A continuación vamos a explicar por qué el algoritmo es correcto.

Realizamos un grafo que representa a la provincia, sus ciudades y a las rutas entre ciudades. A este grafo lo copiamos k+1 veces. A los grafos les reemplazamos las aristas premium como se ha explicado anteriormente, y obtenemos un grafo dirigido, donde el i-ésimo grafo se conecta con el i+1-ésimo grafo solamente por los ejes dirigidos que representan a las rutas premium.

Con este nuevo grafo, obtenemos que cada ciudad es representada por k+1 nodos, con un único nodo que la representa en cada copia del grafo provincia. Como cada i-ésima copia del grafo provincia se conecta con la (i+1)-ésima copia (mientras ésta exista, es decir hasta la k-ésima copia) con únicamente ejes dirigidos desde el primer grafo hasta el segundo, cualquier camino que vaya desde del i-ésimo hasta el (i+1)-ésimo deberá pasar necesariamente por alguno de estos ejes dirigidos. Como no hay ejes que vayan desde el (i+1)-ésimo grafo hasta el i-ésimo, cualquier camino desde el i-ésimo hasta el (i+1)-ésimo no solamente utilizará una ruta premium, si no que es la única que utiliza, debido a que no se puede volver al primer grafo para pasar por otro eje que vaya hacia el segundo grafo.

De esta forma, al comenzar un camino desde un nodo del 0-ésimo grafo, representamos el inicio de un camino desde la ciudad que ese nodo representa, sin haber utilizado, particularmente, ninguna ruta premium. Como todo eje de un mismo i-ésimo grafo representa a una ruta común, todo camino entre dos nodos que se mantenga en el mismo i-ésimo grafo no utiliza rutas premium.

Utilizando esta lógica, un camino desde un nodo del 0-ésimo grafo hasta un nodo del i-ésimo grafo puede expresarse como la unión de subcaminos los cuales son caminos sin rutas premium mientras los nodos del subcamino pertenezcan al mismo j-ésimo grafo, y caminos de un eje conectando el j-ésimo grafo con el (j+1)-ésimo grafo, representando el uso de una ruta premium.

Luego, si el nodo de origen pertenece al 0-ésimo grafo y el nodo destino pertenece al i-ésimo grafo, sabemos que debe pasar por una ruta premium por cada salto desde un j-ésimo grafo a otro h-ésimo grafo al usar sus nodos para el camino, y que por construcción, este h-ésimo grafo va a ser el (j+1)-ésimo grafo, es decir, se incremente el índice en 1, obteniendo así un camino que utiliza exactamente i rutas premium, una por cada salto comenzando desde el 0-ésimo grafo, hasta llegar al i-ésimo.

Desde el punto de vista de la provincia siendo representado por nuestra construcción del grafo, al comenzar un camino desde una ciudad determinada, se comienza un camino desde el nodo que la representa. Por cada ruta común que se utilice se generará el camino en nuestro grafo usando los ejes que representan a dicha ruta común y por ende manteniéndose en el mismo j-ésimo grafo en el que se esté actualmente el último nodo del subcamino recorrido. Por cada vez que se utiliza una ruta premium entre dos ciudades a y b, significa que en nuestro grafo hay un eje dirigido desde el nodo que representa a la ciudad a en el j-ésimo grafo en el que se esté actualmente en el subcamino, y ese eje dirigido llega hasta el nodo destino que representa a la ciudad b en el (j+1)-ésimo grafo.

De esta forma se puede obtener un camino entre dos nodos cualesquiera mientras se utilicen a lo sumo k rutas premium, y nuestro grafo construido representará adecuadamente a este camino.

Concluyendo, todo camino desde un nodo "origen" del 0-ésimo grafo hasta cualquier nodo "destino", es un camino en el cual se utilizan "j" aristas premium si el nodo "destino" pertenece a la j-ésimo copia del grafo. Luego, aplicando Dijkstra desde el nodo "origen" obtenemos el camino más corto a todos los nodos con la condición de que si un nodo pertenece al i-ésimo grafo, el camino utiliza exactamente "i" aristas premium.

Para obtener el camino mínimo desde un nodo "origen" hasta un nodo "destino" utilizando a lo sumo k aristas premium, al aplicar Dijkstra sobre el digrafo obtenido y con origen en el nodo que representa a la ciudad de origen en el 0-ésimo grafo, obtenemos el camino mínimo hasta los k+1 nodos que representan a "destino", uno por cada copia del grafo original que se realizó. Obteniendo el valor mínimo entre

todos estos nodos (sin considerar los valores indefinidos en caso de que el camino no exista), se obtiene la distancia más corta que se puede hacer para llegar al nodo "destino" utilizando a lo sumo "k" aristas premium.

2.5 Complejidad del algoritmo

Para obtener la complejidad temporal del algoritmo, vamos a analizar todas las partes del mismo guiándonos con el pseudocódigo. Notemos "n" la cantidad de nodos, "m" la cantidad de ejes (incluye premium), y "k" la cantidad permitida para usar rutas premium. Decidimos utilizar un vector de nodos para representar a un grafo dirigido donde los nodos contienen una lista con las aristas, y las aristas son punteros a los nodos vecinos y un peso fijo.

De esta forma procedemos a analizar las complejidades:

- **Copiar el grafo k+1 veces y agregar los ejes premium:** En el pseudocódigo, se tiene como parámetro de entrada a un grafo que ya representa a la provincia. Como nosotros tenemos que armar el grafo, vamos a armar un grafo con n nodos, luego lo copiamos k+1 veces, obteniendo un vector por cada grafo copiado y luego agregamos los ejes. Hacer un grafo con n nodos a complejidad constante cada nodo es $O(n)$. Copiar esto tiene complejidad $O(n * k)$. Luego agregar las aristas tiene complejidad constante por cada una (sea premium o no, ya que consiste en apuntar a nodos identificables por sus índices es un vector), pero cada arista se agrega en los k+1 grafos, se tiene complejidad $O(k)$ por cada ruta y como se tienen m rutas, la complejidad es $O(m * k)$. Sumando las complejidades, se obtiene que la complejidad es $O((n + m) * k)$
- **Aplicar Dijkstra:** Nuestra implementación de Dijkstra consiste en utilizar una cola a la cual extraer el nodo de menor distancia calculada tiene tiempo lineal. De esta forma el algoritmo de Dijkstra según nuestra implementación usando una cola simple tiene complejidad $O(h^2)$, donde h es la cantidad de nodos del grafo. Como nuestro grafo tiene $O(n * k)$ nodos, la complejidad de Dijkstra es de $O((n * k)^2)$.
- **Buscar el mejor resultado para el destino:** Como se tienen k+1 nodos que representan a el destino, obtener el mínimo entre éstos tiene complejidad $O(k)$, ya que comparar dos números tiene complejidad constante.

Luego sumando todas las complejidades y acotando la cantidad de ejes por n^2, obtenemos la complejidad: $O((n+m)*k+(n*k)^2+k) = O((n+n^2)*k+(n*k)^2+k) = O(n^2*k+n^2*k^2+k) = O(n^2*k^2)$

2.6 Experimentación

Como la complejidad depende de dos variables, la cantidad de nodos (N) y la cantidad de rutas premium que se pueden utilizar (K), decidimos experimentar fijando N en un valor 100 y variando K y luego fijando K en un valor 100 y variando N. Los experimentos realizados son los casos descriptos a continuación, en los cuales se han analizando cada instancia 10 veces y obteniendo su tiempo promedio.

- **Caso malo:** Un mal caso se da cuando el grafo es un grafo completo con solución y se utilizan todas las rutas premium posibles (ya que nuestra implementación hace que al usar una ruta premium mas se analice un grafo nuevo y se accedan a más nodos). De esta forma, como nuestro algoritmo no termina al encontrar la solución, sino que se analizan todos los nodos en todos los grafos que se realizaron para representar el uso de las K rutas premium.
- **Caso bueno:** Un buen caso se da con un grafo sin aristas. De esta forma la construcción del grafo de "n" nodos se construye de la forma más rapida al no haber ejes que generar, y buscar la solución es analizar un sólo nodo, ya que al no haber ejes, nuestra implementación de Dijsktra termina al analizar únicamente al nodo de origen.
- **Casos randoms:** Para los casos randoms generamos para cada instancia de "n" nodos, una cantidad de ejes entre 0 y $n*(n-1)/2$ donde los valores en ese rango son equiprobables de obtener. Cada eje también es premium o no decidido de forma aleatoria, también equiprobablemente. Los ejes de origen y de destino son respectivamente el nodo 1 y el nodo n para cada instancia y el grafo generado puede no ser conexo. En estos casos, para cada tamaño de entrada, se crearon 10 grafos aleatorios distintos y cada uno fue promediado en tiempo 10 veces, siendo este resultado el tiempo para cada tamaño de entrada.

Los resultados obtenidos para los malos casos y los casos random cuando N es variable y para cuando K es variable son expuestos a continuación.

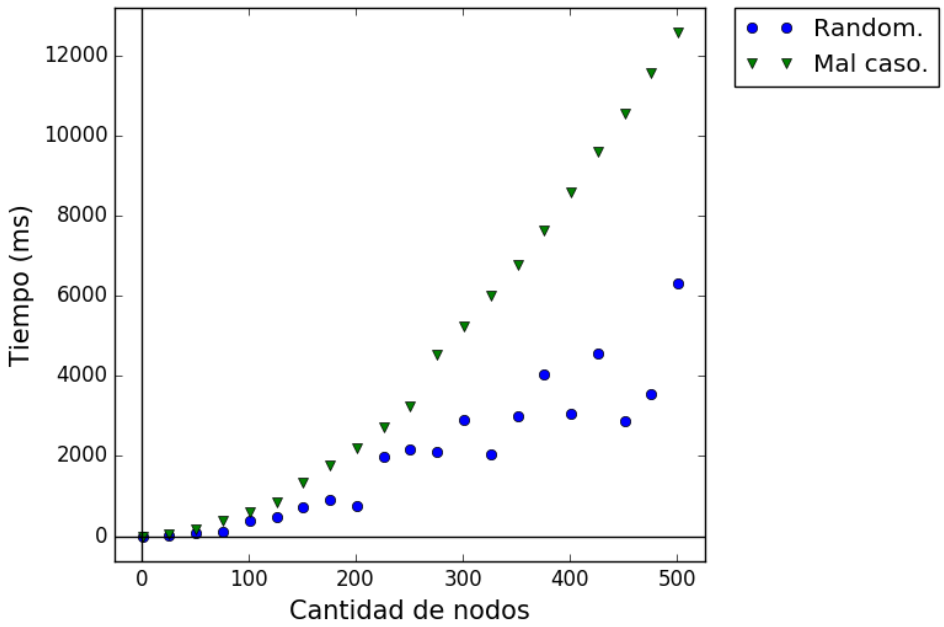


Figure 2: Malos casos y casos randoms. N variable.

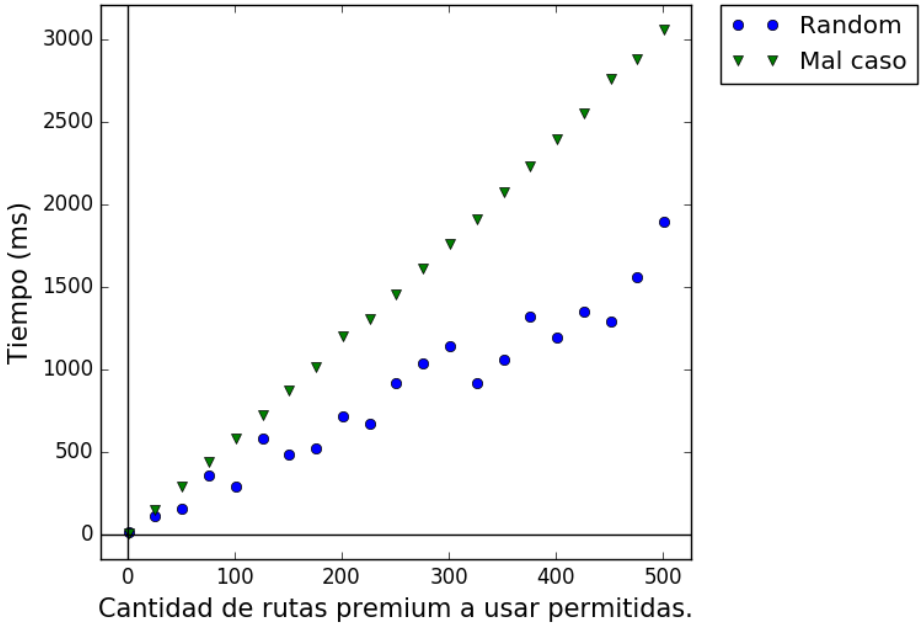


Figure 3: Malos casos y casos randoms. K variable.

En estos resultados puede observarse como los casos random se encuentran por debajo de nuestros malos casos. Esto indica que nuestros malos casos dan una buena idea de que tan mal puede comportarse en performance nuestro algoritmo. Los casos random tienen cierta variabilidad incluso luego de ser promediados entre distintas instancias. Esto puede deberse a que al generarse de forma equiprobable la cantidad de ejes y a que nodo pertenecen, puede ocurrir que se generen grafos donde la ciudad de origen pueda alcanzar solamente unas pocas ciudades, haciendo que el algoritmo termine al poco tiempo de haber iniciado, y haciendo que el tiempo promedio se reduzca. Debido a que no se generó una cantidad alta de grafos para cada tamaño de instancia, pudo ocurrir que en algunos tamaños no haya este tipo de grafos "rápidos" y en otras puede que hayan mas de uno, haciendo que entre tamaños de instancia haya una variabilidad de performance.

Otro resultado que puede deducirse a partir de los gráficos, es que la cantidad de nodos (N) tiene más peso a nivel de performance que la variable K. Esto puede deberse a que si no se varía el grafo, la cantidad de nodos y aristas es proporcional a la variable K por las copias que se realizan, y si bien la complejidad es $O(N^2 * K^2)$, debido a que nuestra implementación de Dijkstra compara los nodos vecinos y no todos los nodos del grafo, incrementar el valor de K tiene efectos similares a aplicar Dijkstra al grafo original K veces, siendo el tiempo percibido $O(N^2 * k)$ en donde N^2 al ser un valor fijo, se percibe en realidad una complejidad $O(K)$.

Los tiempos de los casos buenos fijando ambas variables fueron:

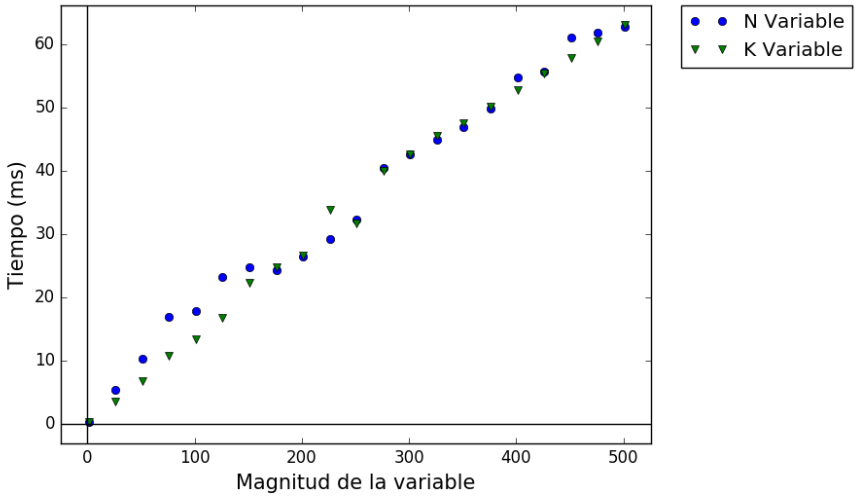


Figure 4: Casos buenos.

Las conclusiones que podemos obtener con estos resultados son que comparando las escalas de los gráficos, los casos buenos tienen mucha diferencia de performance con los malos, y no se puede apreciar alguna diferencia de importancia entre qué variable tiene más peso a nivel de performance.

Finalmente, para corroborar que la complejidad del algoritmo sea la propuesta, dividimos los tiempos de los malos casos por la complejidad temporal, y de esta forma obtuvimos el siguiente gráfico:

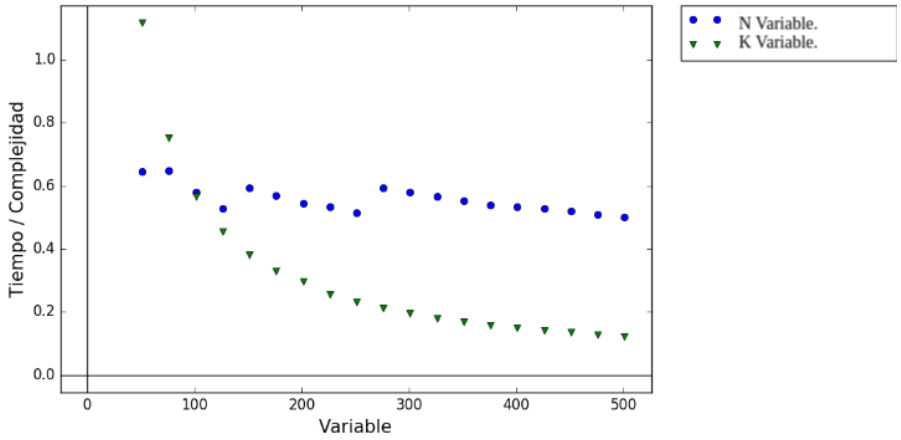


Figure 5: Malos casos divididos su complejidad temporal.

Como puede observarse, el comportamiento en ambos casos es decreciente, mostrando que la complejidad no supera a la complejidad pedida de $O(n^2 * k^2)$. También puede observarse como el caso cuando se fijan la cantidad de nodos tiene un comportamiento decreciente mayor al caso en donde se fijan la cantidad de rutas premium a usar. Esto muestra que la cantidad de nodos tiene un peso mayor al considerar los factores que hacen que el algoritmo sea más rápido o más lento.

Otra herramienta que utilizamos para corroborar que nuestra complejidad es adecuada, es calcular el coeficiente de pearson. Para los datos de nuestras instancias malas, el coeficiente obtenido fue de 0,97 comparándolos con la función n^2, indicando una gran similitud entre ésta y los datos; y calculando

el coeficiente al compararlo con la función $n^1, 4$, obtuvimos el valor 0,99, lo cual indica que nuestra complejidad temporal en estos casos es menor a $O(n^2)$.

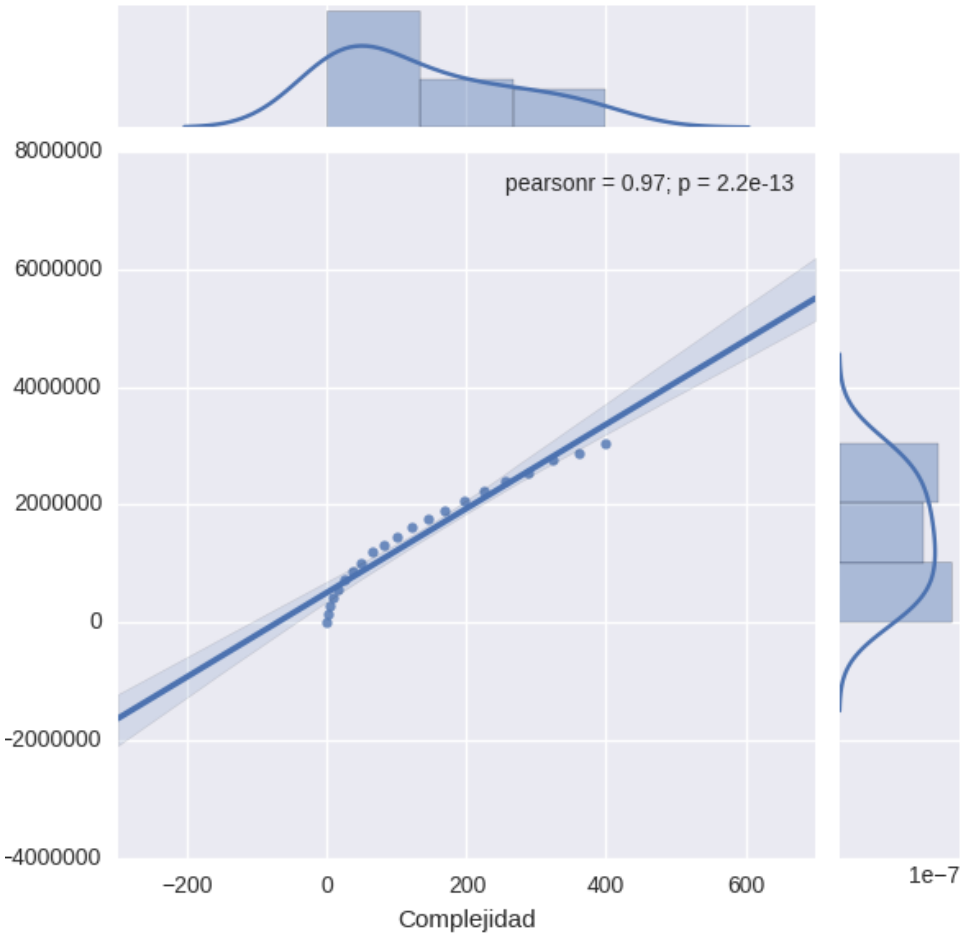


Figure 6: Pearson comparado con la función n^2 para N variable.

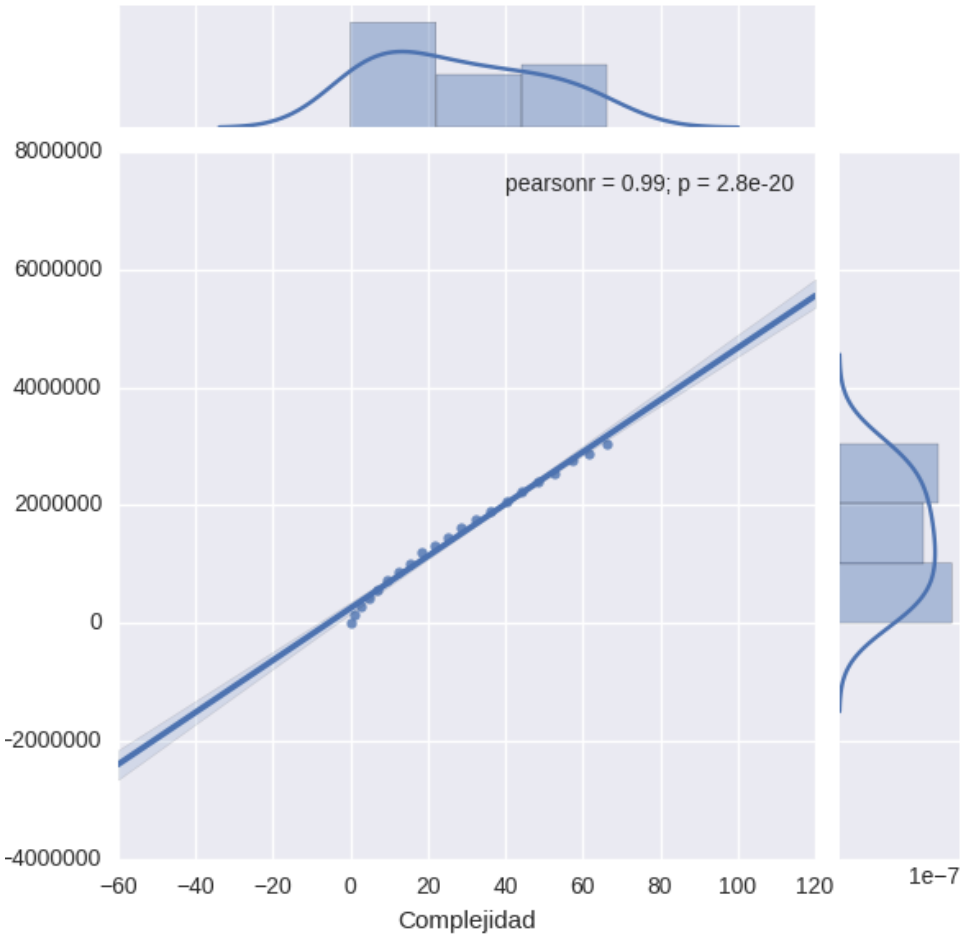


Figure 7: Pearson comparado con la función $n^{1,4}$ para N variable.

De la misma forma, calculamos el coeficiente de Pearson para los casos malos cuando fijamos a N en 100, siendo K la variable. Comparándo los datos obtenidos con la curva $k^1, 45$, obtuvimos un coeficiente de 1, indicando si similiaridad con dicha fórmula, y por lo tanto corroborando que nuestra complejidad no supera a $O(k^2)$ al fijar N.

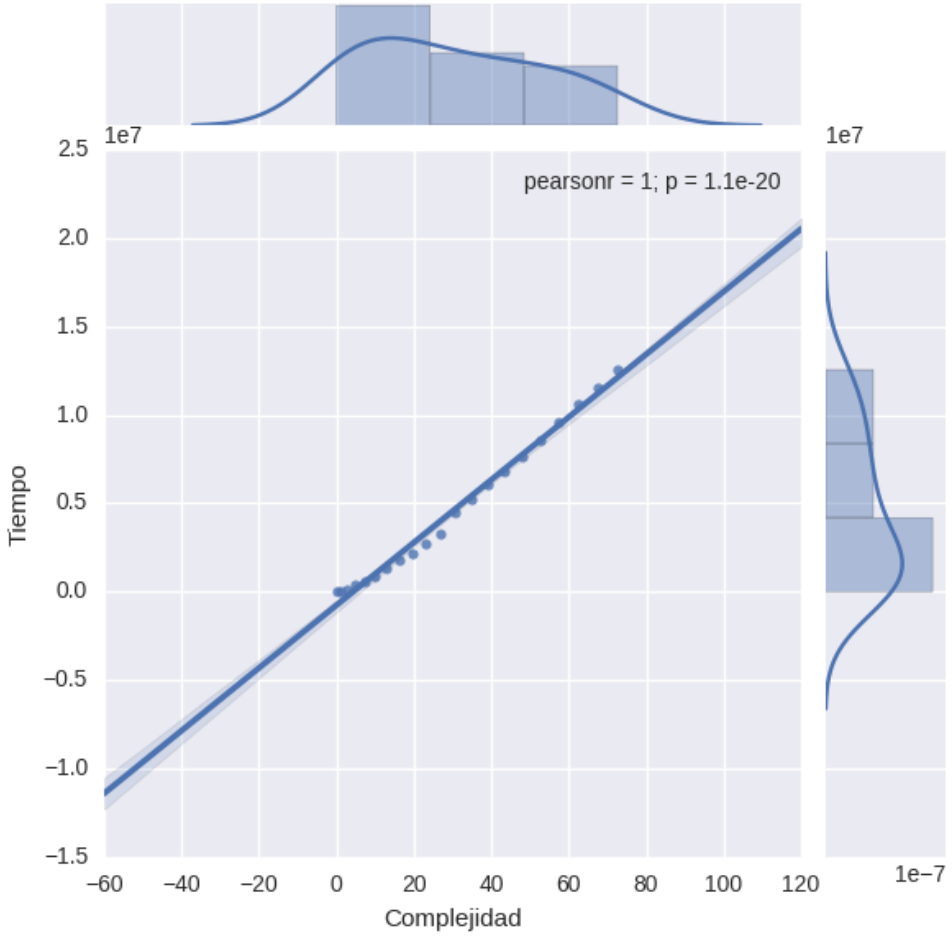


Figure 8: Pearson comparado con la función $n^{1,45}$ para K variable.

3 Problema 2: Subsidiando el transporte

3.1 Introducción

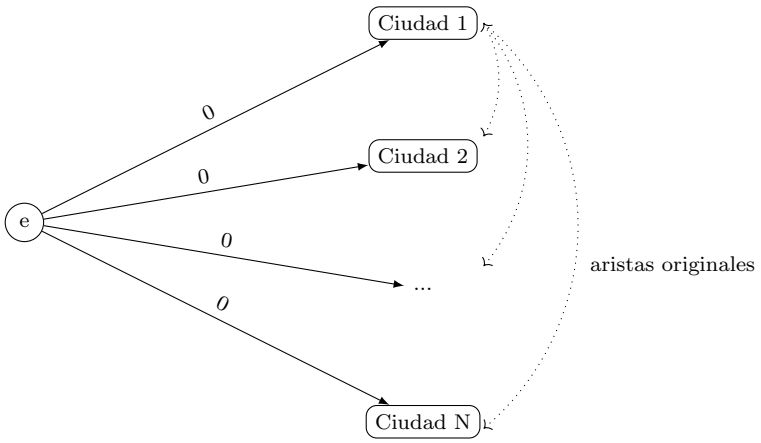
En éste problema se nos pasa por entrada las distintas rutas entre ciudades con el costo de transitarlas. Se nos pide encontrar un valor único para restar al costo de todos los peajes, siendo éste el máximo valor que se puede restar sin que una persona pueda salir de una ciudad y volver a la misma ganando plata.

El problema fue modelado con un grafo, donde los vértices son las ciudades y las aristas son las rutas, siendo el peso asociado a las mismas el costo del peaje. La condición de salir de una ciudad y volver ganando plata es equivalente a encontrar un ciclo negativo, ya que el peso de las aristas es el gasto de recorrerlas.

3.2 Resolución del problema

Sabemos que el valor C a restar en todos los pesos de las aristas se encuentra entre el mínimo y el máximo peso, dado que si restamos el mínimo quedarían todos los valores positivos, y si restamos el máximo quedarían todos los pesos no-positivos (≤ 0). Por lo tanto realizamos una búsqueda binaria entre esos valores, tomando un potencial C (llamado v en el algoritmo) y luego verificando que no cree ciclos negativos. Para verificar que el v propuesto no genere ciclos negativos, optamos por usar la verificación del algoritmo de Bellman-Ford. Es importante notar que en ningún momento utilizamos dicho algoritmo para encontrar caminos.

Para que el algoritmo de Bellman-Ford verifique que no hay ciclos negativos en todo el grafo, debemos buscar los caminos mínimos desde cada nodo. Para realizar las N búsquedas en una sola ejecución del algoritmo podemos pensar que hay un nodo extra e que se conecta a todos los nodos del grafo por medio de aristas de peso 0. De esta forma, el algoritmo llega a todos los nodo con costo 0, y desde ese estado se revisan todas las aristas. Como todas las nuevas aristas tienen como origen el nuevo nodo e, no se agregan ni quitan caminos entre cada par de nodos del grafo original. De esta forma, los invariantes del algoritmo de Bellman-Ford se mantienen, y si e es nuestro nodo de inicio, se revisan todos los caminos que inician en cada nodo del grafo original. Verificar si hay un ciclo de peso negativo desde e seria equivalente a verificar que haya algún ciclo negativo en todo el grafo.



El costo del Bellman-Ford con la verificación de los ciclos negativos es $\mathcal{O}(EV)$. Si bien esta solución se puede utilizar y haría uso de un Bellman-Ford sin ninguna modificación, como estoy agregando 1 nodo y V aristas, su costo seria de $\mathcal{O}(V+1*(V+E)) = \mathcal{O}((V+1)*V + (V+1)*E) = \mathcal{O}(VE + V^2)$

Sin embargo, podemos observar que en el grafo propuesto todos los nodos tienen como mínimo peso 0, y el nodo e no aporta al funcionamiento del algoritmo (es solo para seguir calculando todos los caminos desde un nodo particular). Por lo tanto, el algoritmo tendría el mismo funcionamiento y resultado (siempre hablando de la busqueda de ciclos negativos, no de caminos) si en vez de agregar el nodo e y las E aristas desde e hasta cada nodo del grafo original, simplemente inicialicemos el peso de cada nodo en 0. De ésta forma, el algoritmo vuelve a costar $\mathcal{O}(EV)$ y sigue calculando si existe algún ciclo de peso negativo en el grafo.

3.3 Pseudocódigo

Arista es una clase que solamente tiene 3 enteros públicos: origen, destino y peso.

Algorithm 2 Bellman-Ford modificado

```
1: procedure BellmanFordModificado(int n, int m, vector<Arista> &aristas, int resta) → bool
2:     vector< int > distancia(n,0)                          ▷ e se une a todos con peso 0
3:     for i = 0 to n do
4:         for j = 0 to m do
5:             p ← aristas[j].origen-1;      ▷ Las ciudades se enumeran de 1 a N, distancia es de 0 a N-1
6:             q ← aristas[j].destino-1;
7:             w ← aristas[j].peso;
8:             if distancia[p] + w - resta < distancia[q] then
9:                 distancia[q] ← distancia[p] + w - resta
10:            end if
11:        end for
12:    end for
13:    for j = 0 to m do
14:        p ← aristas[j].origen-1;
15:        q ← aristas[j].destino-1;
16:        w ← aristas[j].peso;
17:        if distancia[p] + w - resta < distancia[q] then return false      ▷ hay un ciclo negativo
18:        end if
19:    end for
        return true
20: end procedure
```

Algorithm 3 Main del ejercicio2

```
1: procedure EJERCICIO2(int n, int m, int cmin, int cmax, vector<Arista> &aristas) → int
2:     vini ← cmin                          ▷ cmin es un C valido (no deja ciclos negativos)
3:     vfin ← cmax
4:     repeat
5:         v ← ⌊ (vfin + vini)/2 ⌋
6:         sinCiclosNeg ← BellmanFordModificado(n,m,aristas,v)
7:         if sinCiclosNeg then
8:             vini ← v                     ▷ vini siempre tiene un valor valido para C
9:         else
10:            vfin ← v                     ▷ Como hay ciclos negativos, tomo un v más chico
11:        end if
12:    until vfin-vini < 2
       return vini;
13: end procedure
```

3.4 Correctitud

El algoritmo hace búsqueda binaria sobre los posibles valores de C, que se encuentran entre [mínimo peso, máximo peso) salvo cuando el mínimo y el máximo son iguales. Como se reducen los pesos de todas las aristas por el mismo valor, al encontrar un valor para el que se genere un ciclo negativo, sabemos que los valores mayores a esos también van a generar ciclos negativos porque las aristas involucradas van a ser aún menores. De la misma forma, si encontramos un valor para el cual no se generen valores negativos, los valores menores a este no van a tener ciclos negativos debido a que sus ejes van a tener un peso mayor. De esta forma, usando búsqueda binaria vamos a hallar el valor para el cual no haya ciclos negativos y que además el valor sucesor da ciclos negativos.

Como se observa en el pseudocódigo, el invariante del ciclo Repeat-Until es que en vini siempre hay un valor valido. Al salir del ciclo en vini se encuentra el máximo valor válido para C, por lo que devolvemos el valor requerido.

Verificar que v sea un valor válido (que no haya ciclos negativos) se realiza con el Bellman-Ford modificado que fue explicado previamente. Como el algoritmo es equivalente a agregar el nodo e y las aristas del nodo e hacia el resto de los nodos y aplicar Bellman-Ford con el nodo e de nodo inicial, podemos concluir que el Bellman-Ford modificado devuelve correctamente si hay un ciclo negativo o no.

Luego, si el valor v (el cual es siempre el punto medio del intervalo de los posibles candidatos) es válido, significa que v es un valor posible C y como buscamos el C máximo, podemos descartar todos los valores inferiores, por lo que sabemos que la solución se encuentra en el intervalo [v, vfin]. Si el valor es inválido, entonces v no puede ser una solución, y por ende tampoco todos los valores mayores a v, descartándolos y obteniendo que la solución se encuentra ahora en el intervalo [vini, v]. Una vez terminado esto, obtenemos un valor v que será nuestra solución ya con este valor es el mayor valor para el que no se forman ciclos negativos, ya que sabemos que todos los valores superiores forman un ciclo negativo.

3.5 Complejidad

La complejidad del Bellman-Ford modificado es la misma que el algoritmo original, por lo que cuesta $\mathcal{O}(nm)$. Este es llamado $\lfloor log_2(cmax - cmin) \rfloor$ veces, que es $\mathcal{O}(log(c))$.

Concluimos entonces que el algoritmo cuesta $\mathcal{O}(nm * log(c))$

3.6 Experimentación

Para los experimentos armamos casos que varían la cantidad de nodos, y toman el tiempo promedio de 50 repeticiones de cada instancia. En los casos con grafos aleatorios, al realizar las repeticiones para cada tamaño de instancia, realizamos un nuevo grafo y tomamos el tiempo a éste grafo (una vez), promediando luego los tiempos obtenidos para todos los grafos del mismo tamaño de instancia. Decidimos medir los siguientes casos en función de la cantidad de nodos (N), en donde los pesos de los ejes son valores aleatorios entre 1 y 100:

- **Grafos completos:** Estos grafos, al tener la mayor cantidad de ejes posibles, serán grafos en los que su performance va a ser mala.
- **Grafos mínimos:** Estos grafos son grafos que únicamente tienen un eje de salida en cada nodo, respetando la condición del problema de que cada ciudad puede dirigirse a al menos una ciudad. Al tener la menor cantidad de ejes posibles, estos casos van a tener tiempos de ejecución rápidos.
- **Grafos aleatorios:** Partiendo de grafos mínimos, se decidió generar de forma equiproblable una cantidad de ejes cuyos nodos de origen y destinos son aleatorios y también equiprobable, verificando que no se repitan ejes.

Los resultados obtenidos para estos 3 casos fueron:

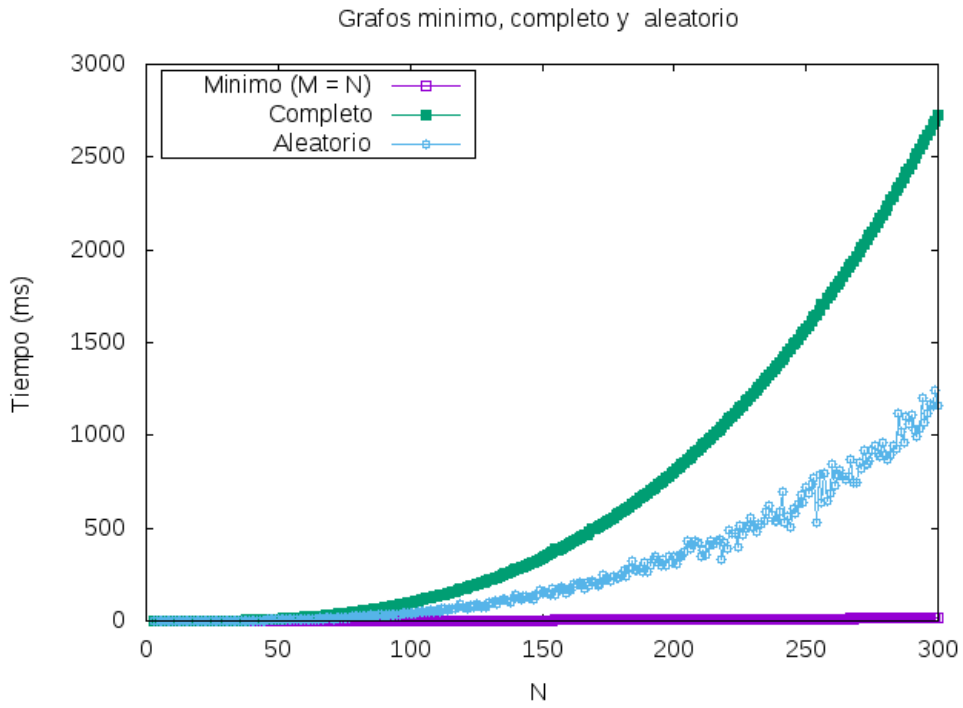


Figure 9: Tiempos de ejecución de los 3 casos.

Debido a que por la escala no se pueden observar los datos obtenidos para los grafos mínimos, éstos se muestran en el siguiente gráfico:

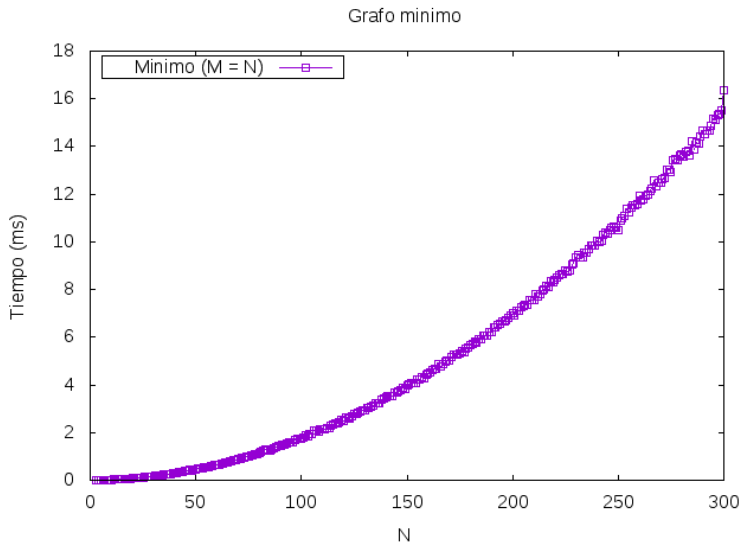


Figure 10: Tiempos de ejecución de los grafos mínimos.

Observando los gráficos podemos deducir que hay mucha diferencia de performance dependiendo de los tipos de grafos que se resuelvan. Los grafos aleatorios se resuelven en promedio a la mitad del tiempo que les toma a los grafos completos, y los grafos mínimos se resuelven con tiempos de ejecución muy cortos.

Para corroborar que los tiempos obtenidos cumplen con la complejidad del problema, decidimos además calcular el coeficiente de Pearson comparando los resultados obtenidos de los grafos completos con la función $N * M * log_2(c)$. El coeficiente obtenido fue de 1, lo que indica que la cercanía de los datos corresponde con la complejidad adecuada del problema.

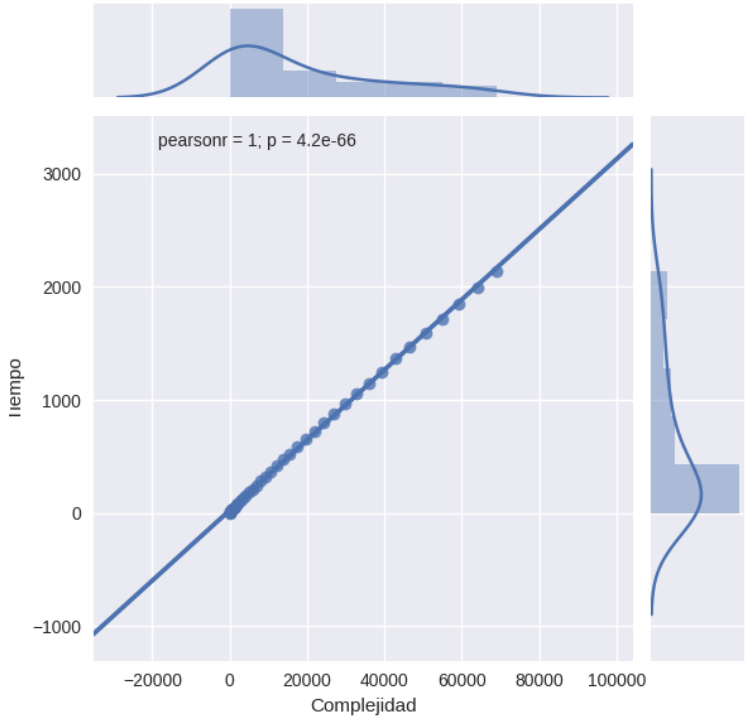


Figure 11: Pearson para grafos completos.

Para observar también el peso que tiene el valor C en la complejidad, medimos tiempos tomando un grafo mínimo de nodos n = 150 y calculando el tiempo del algoritmo variando la diferencia entre el mínimo peso del grafo y el máximo, obtuvimos los siguientes resultados:

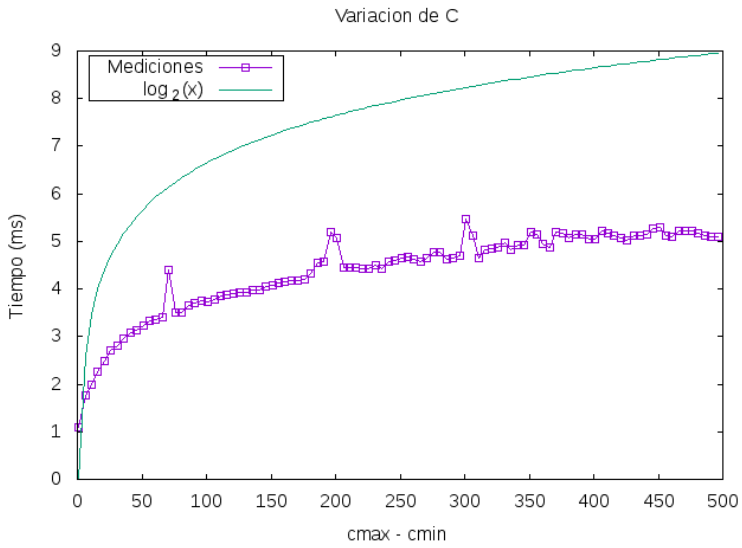


Figure 12: Influencia del valor C en la complejidad.

En este caso el gráfico muestra un crecimiento logarítmico, como lo indica la complejidad, y para corroborarlo calculamos su coeficiente de Pearson comparando los datos con la función $log_2(C)$, obteniendo un coeficiente de 0,98, indicando una gran similitud entre las curvas y verificando nuevamente que la complejidad es adecuada.

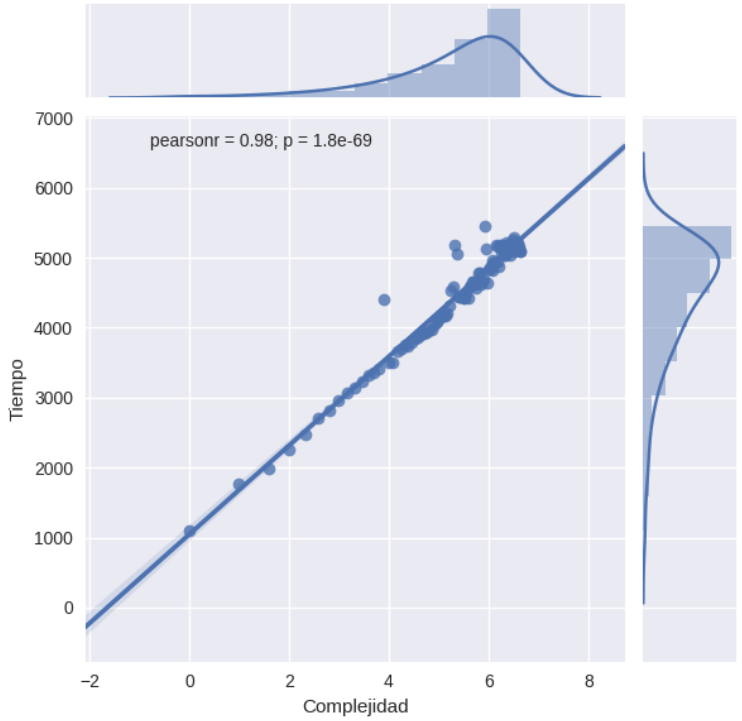


Figure 13: Pearson para grafos mínimos con 150 nodos y C variable.

4 Problema 3: Reconfiguraciónon de rutas

4.1 Introducción

Se tiene una ciudad con rutas construidas y rutas sin construir, cada ruta tiene un costo de construcción o de destrucción, se quiere poder conectar todas las ciudades con solo una ruta cada dos ciudades de manera tal que se minimice el costo de construir y destruir dichas rutas.

Obs: En la salida del algoritmo lo consultamos y la salida va a estar compuesta por cada una tupla donde cada conecta la ciudad i con la j

4.2 Resolución del problema

Se decidió modelar el problema con un grafo $G = (V, X)$ y una función $l : X \to \mathbb{Z}$ que asigna pesos a las aristas.

Nuestro grafo será un grafo completo, porque tenemos de entrada, los costos de construcción o destrucción de rutas para cada par de ciudades.

Las aristas con peso positivo son las rutas que no están construidas, mientras que las de peso negativo son las rutas que estaban construidas al recibir la entrada, esto lo hacemos porque en nuestro algoritmo vamos a "destruir" todas las rutas ya existente su respectivo costo total guardado y por último buscaremos un AGM utilizando el algoritmo de Kruskal, pues elegimos este y no Prim porque este algoritmo acepta distancias negativas y a medida que el algoritmo va agregando rutas, cuando se queda con una que es negativa, le restamos este peso al costo total que vamos acumulando ya que esto implicaría que no debemos destruirla.

De esta forma, el problema se reduce a conseguir el peso de un árbol generador mínimo (AGM) de dicho grafo.

Para asegurarnos que la complejidad sea la pedida, vamos a implementar el algoritmo de Kruskal sobre una estructura de UNionDisjoinSet (UDS), de esta forma, por lo visto en el taller, nos asegura que la complejidad será de $\mathcal{O}(n + n^2 log(n))$

4.3 Pseudocódigo

USD es una estructura que tiene los atributo *padre* y *componente* y representa a los UnionDisjointSet.

Peso es un entero que almacena el costo de construcción o destrucción de una ruta.

Como el grafo modelado tiene aristas con peso negativo, el peso del árbol generador mínimo puede llegar a ser negativo, entonces ignoramos ese peso.

Para calcular el costo, se suman todas las aristas del árbol que eran positivas, es decir que se construyen y por otro lado se suman todas las aristas con pesos negativos que no pertenecían al árbol, es decir las que se demolieron.

Algorithm 4 Conseguir Peso del AGM

```
procedure GETAGMWEIGHT(ListaAdyacencia E)
    sets ← UDS[n];                                  ▷ O(n)
    treeWeight ← kruskal(E, sets);                  ▷ O(mlog(n))
     return treeWeight;
end procedure
```

En *getAGMWeight* se genera un array de UDS, uno por cada nodo del grafo. Despues se llama al algoritmo de Kruskal que toma como parámetros al grafo y al array creado.

El algoritmo de Kruskal modifica el array de UDS y devuelve un entero que es asignado a *treeWeight*.

Algorithm 5 Kruskal

```
 1: procedure KRUSKAL(ListaAdyacencia E, UDS[] sets)
 2:     treeWeight ← 0
 3:     sort(G)
 4:     for e in E do
 5:         if find(e.inicio, sets) ≠ fin(e.fin, sets) then
 6:             treeWeight ← treeWeight + e.peso
 7:             calles++
 8:             if e.peso>0 then
 9:                 peso = peso +e.peso
10:             end if
11:             join(e.inicio, e.set, sets)
12:         else
13:             if e.peso<0 then
14:                 peso = peso -e.peso
15:             end if
16:         end if
17:     end for
18: end procedure
```

Algorithm 6 USD Find

```
1: procedure FIND(Entero n1, UDS[] sets) return sets[n1].padre
2: end procedure
```

Algorithm 7 USD Join

```
 1: procedure JOIN(Entero x, Entero y, UDS[] sets)
 2:     x ← find(x)
 3:     y ← find(x,sets)
 4:     if |sets[x].componente| > |sets[y].componente| then
 5:         swap(x,y)
 6:     end if
 7:     for z in sets[x].componente do
 8:         sets[z].padre = y
 9:         Agregar z a sets[y].componente
10:     end for
11:     Vaciar sets[x].componente
12: end procedure
```

4.4 Correctitud

Nuestro problema al ser modelado por medio de un grafo, tendrá la siguiente representación: las ciudades están representadas como los vértices del grafo, las rutas entre ciudades son representadas como los ejes del grafo con pesos en los ejes, donde cuando la ciudad está construida, implica que el peso asignado a la arista es negativo, en cambio cuando no está construido el peso es positivo. De esta forma el problema se reduce a encontrar un árbol generador mínimo entre todas las ciudades, porque necesitamos que estén unidas todas las ciudades (vértices) por exactamente solo una ruta (arista) entre cada par de ciudades. Luego como queremos conseguir el costo mínimo posible, nosotros tenemos un grafo completo de aristas con ejes positivos y negativos, entonces nosotros podemos decir que el grafo simula nuestra provincia completamente vacía, sin rutas creadas entre dos ciudades, asumiendo un costo base (el de la suma de todas las rutas que podemos destruir).

Por lo tanto, encontrar el AGM significa encontrar las rutas cuya suma de pesos sea la menor posible para reducir costos, dado que si para cada eje su peso es negativo, no deben ser destruidas (restándose al costo asumido de destruir todas las rutas al comienzo del algoritmo), y si es positivo, deben ser construidas (agregándose al costo total). Al aplicar Kruskal encontramos el AGM y de esta forma es suficiente para resolver el problema.

Como fue explicado en la teórica , el Algoritmo de Kruskal es correcto para grafos conexos con pesos negativos en las aristas, y como la entrada del problema es un grafo completo que además es conexo, entonces se tiene que nuestro algoritmo es correcto.

4.5 Complejidad del algoritmo

La estructura de nuestro algoritmo está determinado por un array de n elementos que representa cada ciudad y con una lista de adyacencia que utilizamos para representar el grafo.

La complejidad de la solución está dada por la complejidad del algoritmo *getAGMWeight* que realiza las siguientes operaciones:

1. Crea un array de n elementos. ($\mathcal{O}(n)$)
2. Corre Kruskal. ($\mathcal{O}(mlog(n))$)

Crear el array cuesta $\mathcal{O}(n)$ porque son n operaciones $\mathcal{O}(1)$

Luego, porque el grafo es completo entonces se tiene que $\mathcal{O}(m) = \mathcal{O}(n * (n-1)/2) = \mathcal{O}(n^2)$

por ultimo, Kruskal, al ser implementado sobre las dos estructuras, lista de adyacencia y UnionDisjoinSet, se ve que la complejidad del algoritmo es $\mathcal{O}(n + mlog(n)) = \mathcal{O}(n + n^2log(n))$.

4.6 Experimentación

4.7 Metodología

Para la realización de los experimentos, vamos a tomar como entrada grafos de tamaño completo con una cantidad de vertices que va a ir desde 2 hasta 1750 aumentando el tamaño de entrada de a un paso.

Para cada tamaño de entrada vamos a correr el algoritmo 10 veces y vamos a tomar el promedio de tiempos entre estos resultados.

El tiempo estará definido por microsegundos y solamente evalua el algoritmo en cuestión, sin medir tiempo de salida/entrada o algún otro proceso que sea ajeno al algoritmo.

4.8 Casos de test

Como este algoritmo es muy particular, no sabemos a priori cuales podrían ser los casos buenos o malos a la hora de correrlo, por lo tanto vamos a generar diversas instancias de entrada variando solamente el peso de las aristas, ya que el grafo de entrada siempre es completo, y si la ciudad existe o no existe no varía en nada porque nuestro algoritmo evalúa a las aristas que existen y las que no existen de todas formas.

Las instancias que creamos fueron:

- Aristas Iguales: Todas las aristas en este grafo tienen exactamente el mismo peso que será el de 10
- Creciente: Las aristas a la hora de ser agregadas, usando el imput establecido para la entrada de los elementos, será de forma creciente desde 1 a m donde m es la cantidad de aristas que tiene el grafo.
- Decreciente: Muy idéntico al anterior pero en forma inversa, desde m hasta 1.
- Alternada: Esta instancia es particularmente una combinación entre la instancia creciente y decreciente, para la posición i de la entrada donde $i \equiv 0(2)$ el valor asignado es creciente mientras que en la posicion $i \equiv 1(2)$ es creciente. Ambas aumentando o disminuyendo en 1 por cada iteración La secuencia decreciente empezará en 500 y la creciente en 0, teniendo una secuencia de la siguiente forma: 0,500,1,499 ..... 500,0,501,-1... etc.
- Random La instancia random le asignamos al arista un peso que generamos a partir de la función rand() de c++ de la siguiente manera: $w(e) \leftarrow rand() \equiv 0(500)$ donde $w(e)$ es el peso de cada arista.

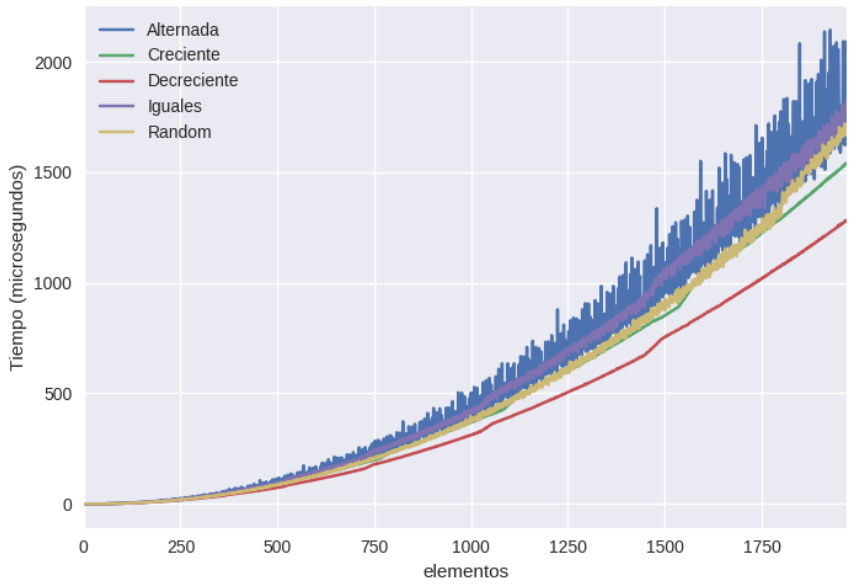


Figure 14: Tiempos de los distintos casos testeados

Por lo que se puede observar en el gráfico, el mejor caso sería el de una secuencia decreciente, el random se mantendría en una media, Pero por ultimo el pero caso sería el de la secuencia alternada, que resulta raro el porqué de este comportamiento ya que en cada iteración oscila entre dos extremos que se van agrandando a medida que el tamaño de entrada va siendo cada vez mas grande, suponemos que tendrá algo que ver relacionado al tamaño, porque el pico mas alto se da cuando el programa tiene una entrada de tamaño impar.

4.9 Fitting

Para demostrar que la complejidad es la pedida lo que hemos hecho fue, tomar cualquiera de las curvas anteriores y compararlas contra una función que incrementa su valor según la función $n^2log(n)$ y calcular el coeficiente de Pearson resultando al comparar estas dos funciones. Como se puede ver en la gráfica de mas abajo, el coeficiente de correlatividad entre las dos curvas es 1 y así se mantiene para todas los otros casos de tests que hemos hecho, lo cual implica que que la complejidad es la correcta. En el caso de la gráfica a continuación estamos usando los datos del con entrada de peso de aristas alternado.

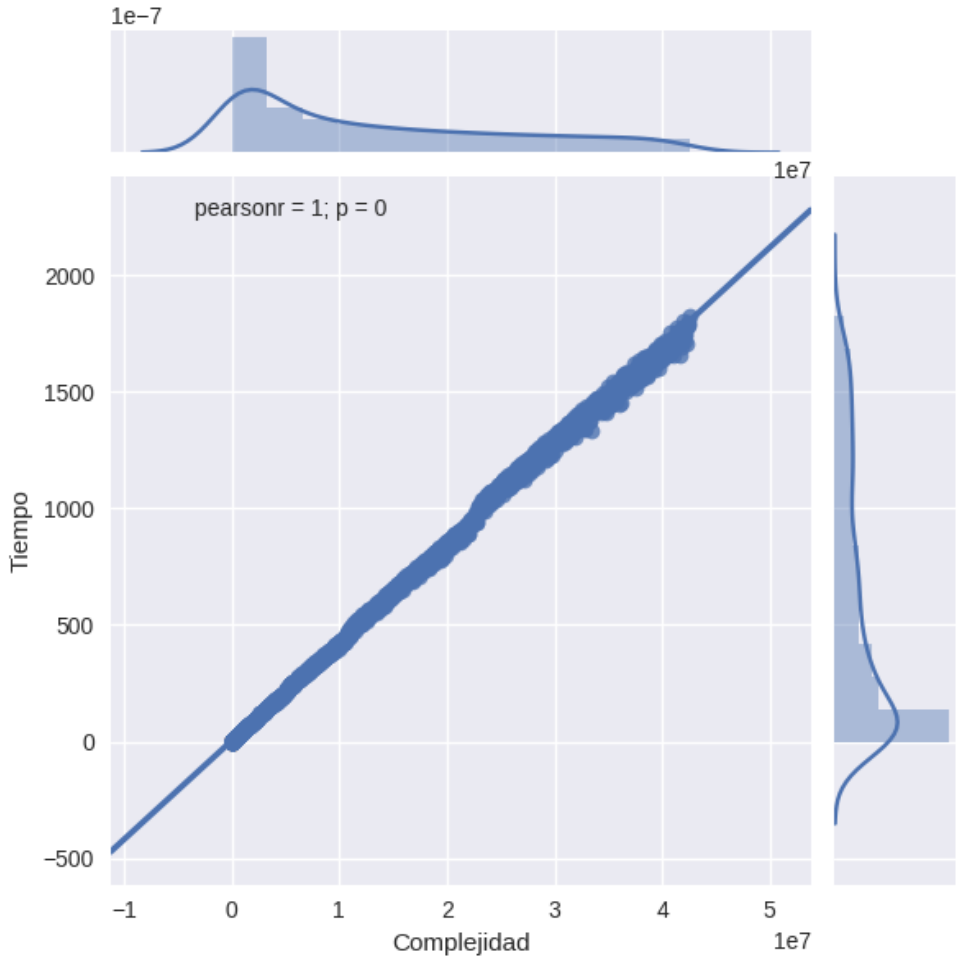


Figure 15: Pearson para una curva experimentada contra $n^2log(n)$

5 Modificaciones realizadas

5.1 Ejericico 1:

- Eliminada la parte implementativa del pseudocódigo del método DistanciaMinima.
- Eliminado el pseudocódigo de Dijkstra por ser un algoritmo conocido.
- Se explicó el mapeo entre nuestra construcción del grafo y la provincia para resolver el problema.
- Expresado en la complejidad del algoritmo que nuestra implementación de dijkstra utiliza una cola simple en vez de una cola de prioridad.
- Se reorganizo la sección de experimentación, manteniendo los mismos experimentos, pero mejorando las explicaciones y desarrollando y deduciendo mejor las justificaciones y separando los gráficos.
- Mejor explicados los casos randoms, buenos y malos.
- Agregados coeficientes de Pearson.
- Se mejoro la explicación de la metodología y se cambió al comienzo de la sección.

5.2 Ejercicio 2:

- Corregida la complejidad temporal.
- Experimentación mejorada y mejor expuesta y explicada.
- Pseudocódigo corregido detalle menor.
- Explicada mejor la correctitud del algoritmo.

5.3 Ejercicio 3:

- Mejorada la explicación de correctitud.
- Agregada la metodología de la experimentación.
- Cambio de gráfico conclusiones y experimentos de tiempo y fitting.