DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

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.

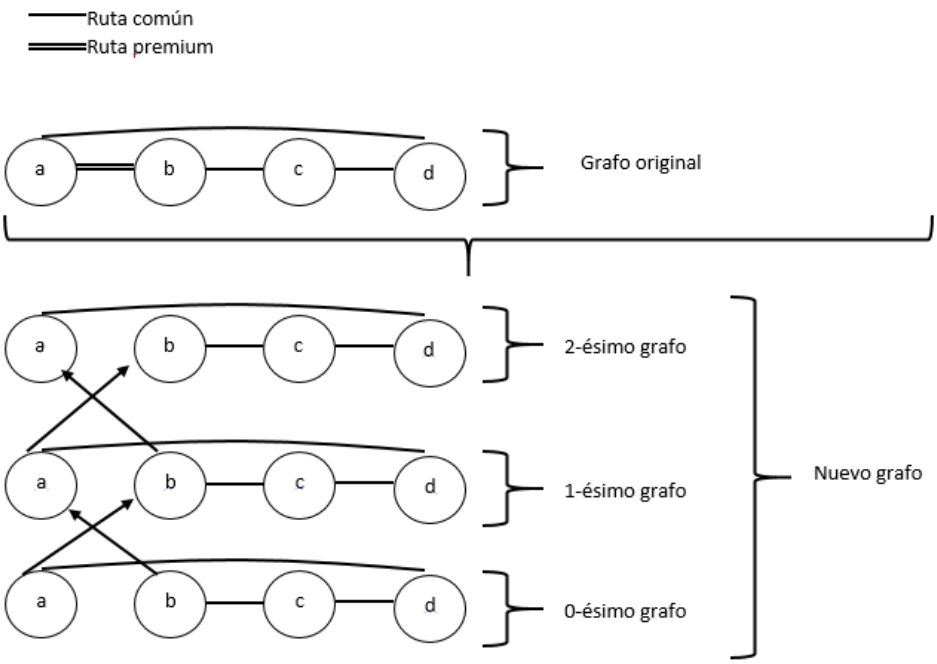


Figure 1: Ejemplo de transformación de un grafo con $k = 2$.

2.3 Pseudocódigo

Algorithm 1 DistanciaMinima

```
1:  procedure DistanciaMinima(graph ciudad, int origen, int destino ) → int
2:      List<graph> grafos = copiar_k+1_veces(ciudad)
3:      borrar_rutas_premiun(grafos)
4:      for Cada ruta premium entre ciudades a y b con peso p do
5:          for (int i = 0; i < k: i++) 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
```

Algorithm 2 Dijkstra

```
1: procedure DIJKSTRA(graph grafo, nodo inicial )
       ▷ Creo una cola de prioridad, donde el nodo con menor distancia_minima tiene mayor prioridad
2:     cola_prioridad ← vacia()
3:     distancia_minima(inicial) ← 0
4:     cola_prioridad.add(inicial)
5:
6:     while cola_prioridad no este vacía do
7:         actual ← cola_prioridad.desencolar()
8:         dist ← distancia_minima(actual)
9:
10:        for all w ∈ {vecinos de actual} do
11:            peso_eje ← peso_eje(actual, w)
12:            peso_nuevo ← dist + peso_eje
13:            peso_viejo ← distancia_minima(w)
14:
15:            if (peso viejo es indefinido) then
16:                cola_prioridad.add(w)
17:            end if
18:            if (peso viejo es indefinido) or (peso_nuevo > peso_viejo) then
19:                definir_distancia_minima(w, peso_nuevo)
20:            end if
21:        end for
22:    end while
23: end procedure
```

2.4 Correctitud

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

Realizamos un grafo que representa a la ciudad y a las rutas. 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 las rutas premium. De esta forma, 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.

De esta forma, 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 ciudad. 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 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: $O((n + m) * k + (n * k)^2 + k) = O((n * k)^2)$

2.6 Experimentación

Para la experimentación hemos considerado los siguientes casos:

- **Peor caso:** El peor 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 mas se analice un grafo nuevo). De esta forma se analizan todos los nodos en todos los grafos que se realizaron para representar el uso de las K rutas premium.
- **Mejor caso:** El mejor caso se da con un grafo sin aristas. De esta forma se construye el grafo que representa al problema y buscar la solución es analizar un sólo nodo, ya que al no haber ejes, nuestra implementación de Dijsktra termina al analizar al nodo original.
- **Casos randoms:** Para los casos randoms generamos para cada instancia de "n" nodos, una cantidad de ejes entre 0 y $n * (n-1)/2$. Cada eje también es premium o no decidido de forma aleatoria. Los ejes de origen y de destino son respectivamente el nodo 1 y el nodo n para cada instancia.

Como la complejidad depende de dos variables, decidimos experimentar fijando una variable en un valor 100 y variando la otra. Los experimentos realizados fueron los casos descriptos anteriormente, analizando cada instancia 10 veces y obteniendo su tiempo promedio. Los resultados obtenidos fueron los siguientes:

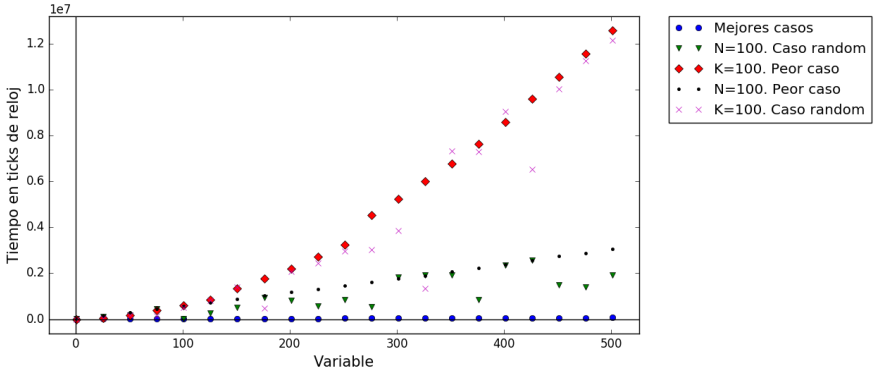


Figure 2: Todos los casos. El eje X indica la magnitud de la variable no fijada.

En estos resultados puede observarse la diferencia entre fijar una variable o la otra. Fijando la cantidad de nodos y variando la cantidad de rutas premium que se pueden utilizar, los tiempos obtenidos fueron inferiores a fijar la cantidad de rutas premium a utiliza y variando los nodos. Esto se debe a que variando los nodos se obtienen una cantidad mayor de ejes para analizar, haciendo del grafo más complejo, y variando la cantidad de rutas premium a utilizar se obtienen varios grafos de manera casi linear, los grafos se mantienen constantes y solo se van agregando copias del mismo con modificaciones para representar a las rutas premium.

Los mejores casos en el gráfico están para mostrar la diferencia de tiempos posible entre los casos. Representan a los tiempos obtenidos en ambos mejores casos. Los tiempos obtenidos fueron similares y debido al tamaño de la escala parecen tener tiempos nulos. Los tiempos de los mejores casos son:

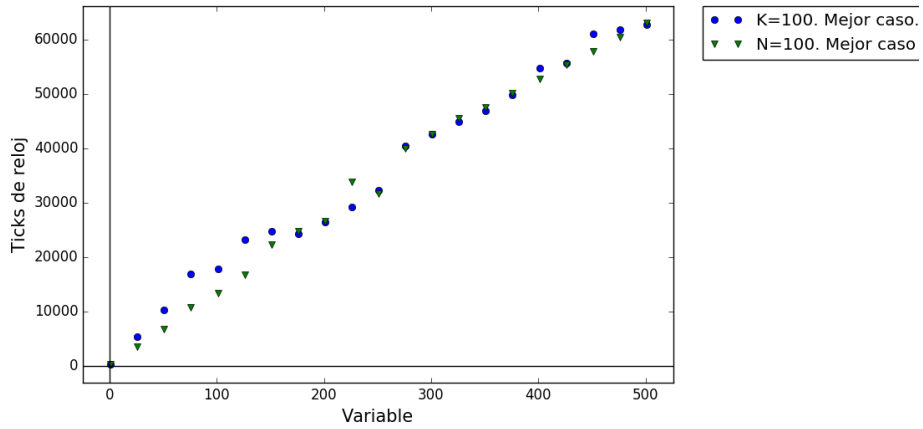


Figure 3: Mejores casos.

En los siguientes gráficos, separamos los casos por qué variable se fijo:

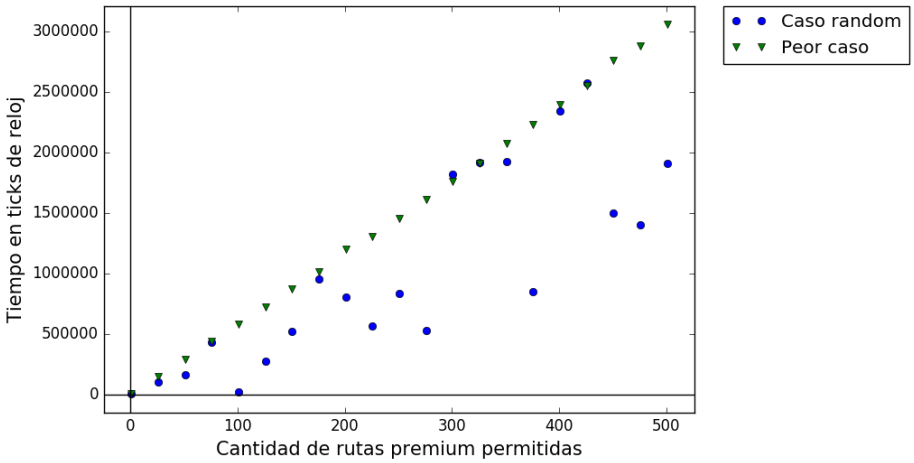


Figure 4: Casos con la cantidad de nodos fijas en 100.

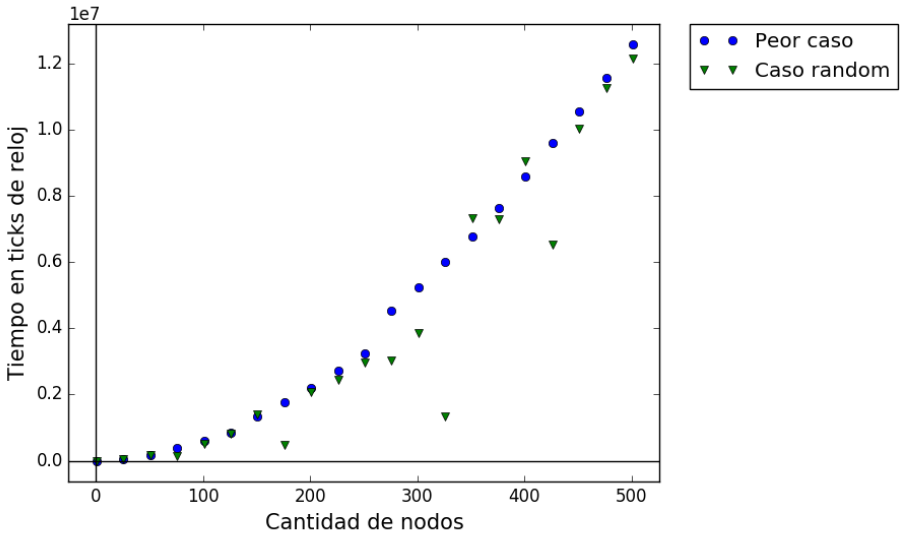


Figure 5: Casos con la cantidad de rutas premium a utilizar fijadas en 100.

Como puede observarse en ambos gráficos, los tiempos obtenidos aleatoriamente son a lo sumo tan malos como los peores casos, pero pueden ser considerablemente mejores.

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

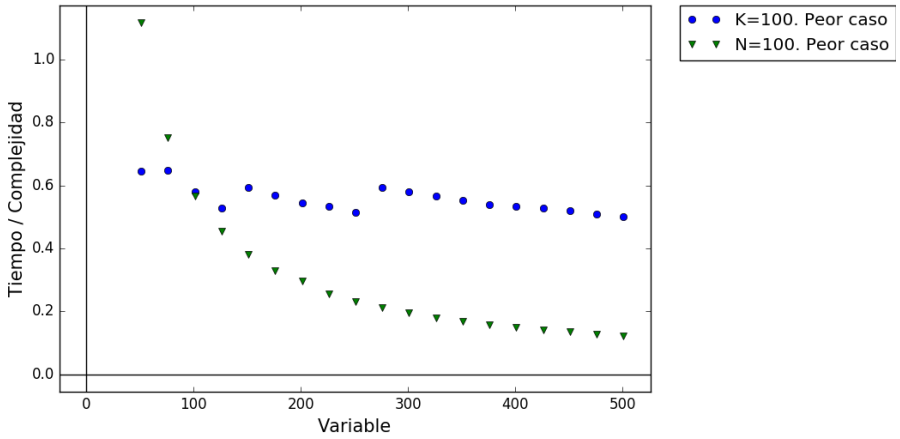


Figure 6: Peores casos divididos su complejidad temporal.

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.

3 Problema 2: Subsidiando el transporte

3.1 Introducción

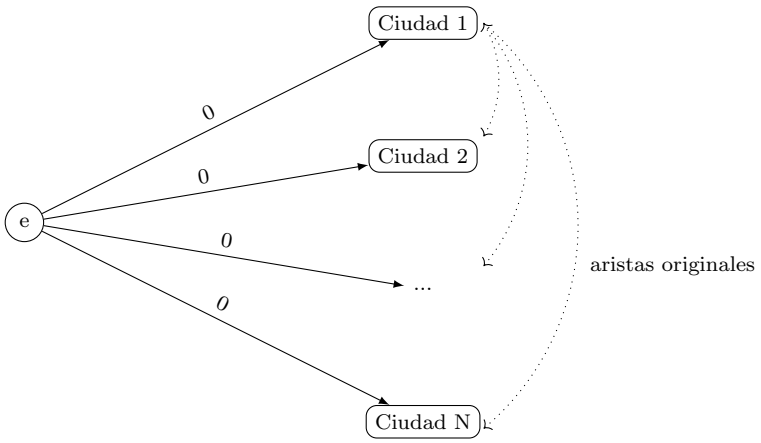
En éste problema se nos pasa por entrada las distintas rutas entre ciudades con el costo de transitarlas. Se nos pide encontrar un valor único para restar al costo de todos los peajes, siendo éste el máximo valor que se puede restar sin que una persona pueda salir de una ciudad y volver a la misma ganando 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 verificacion de los ciclos negativos es $\mathcal{O}(EV + E)$. Si bien esta solución se puede utilizar y haría uso de un Bellman-Ford sin ninguna modificación, su costo seria de $\mathcal{O}($ E+1 * (V+E) + V+E) = $\mathcal{O}$(((E+1)*V+(E+1)*E) dado que agregamos un nodo y E aristas.

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 $\mathcal{O}(EV + V)$ y sigue calculando si existe algun 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 3 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 4 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:     v ← ⌊ (vfin + vini)/2 ⌋
5:     repeat
6:         sinCiclosNeg ← BellmanFordModificado(n,m,aristas,v)
7:         if sinCiclosNeg then
8:             vini ← v                              ▷ vini siempre tiene un valor valido para C
9:             v ← ⌊ (vfin + vini)/2 ⌋
10:        else
11:            vfin ← v                              ▷ Como hay ciclos negativos, tomo un v más chico
12:            v ← ⌊ (vfin + vini)/2 ⌋
13:        end if
14:    until vfin-vini < 2
        return vini;
15: 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. 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 valido para C, por lo que devolvemos el valor requerido.

Verificar que v sea un valor valido 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.

3.5 Complejidad

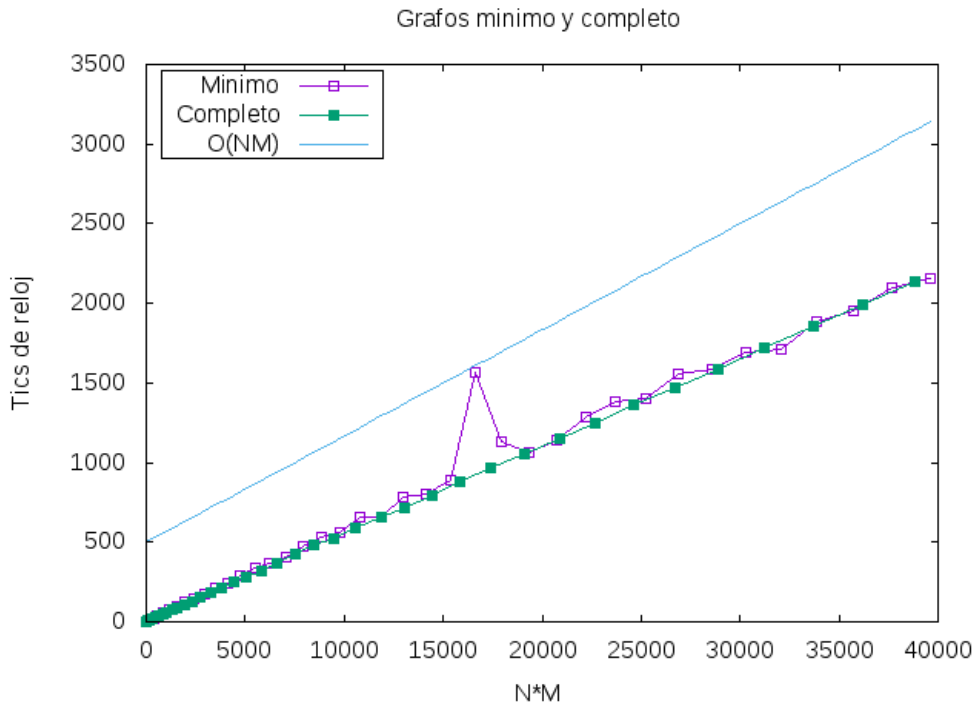
La complejidad del Bellman-Ford modificado es la misma que el algoritmo original, por lo que cuesta $\mathcal{O}(nm)$. Este es llamado $\lfloor log_2(cmax - cmin) \rfloor$ veces, que es $\mathcal{O}(log(c))$.

Concluimos entonces que el algoritmo cuesta $\mathcal{O}(nmlog(c))$

3.6 Experimentación

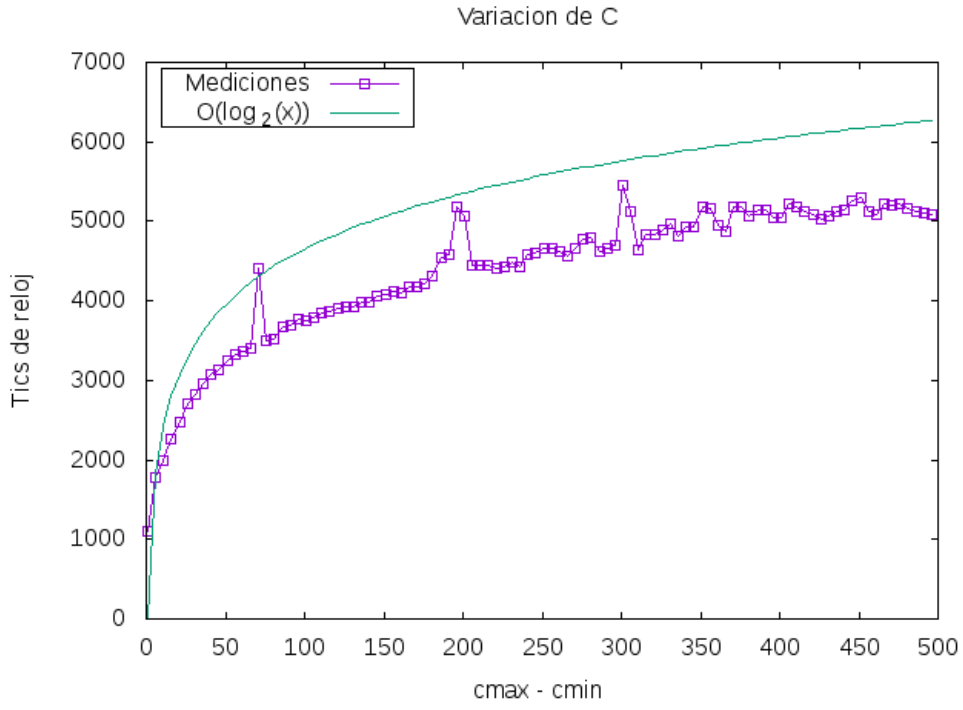
Para los experimentos armamos tests que varían uno de los parámetros, y toman el promedio de 50 repeticiones con grafos aleatorios distintos en cada caso. Decidimos medir N y M juntos en dos tipos de grafos: el grafo mínimo consta de una arista por nodo, donde cada arista tiene su origen en un nodo distinto al resto (lo único que nos asegura el enunciado que ocurre en el grafo de entrada). El otro caso es el grafo completo.

Siendo que el algoritmo de Bellman-Ford cuesta $\mathcal{O}(nm + m)$, el peor caso para el algoritmo debería ser el grafo completo. Sin embargo en los experimentos encontramos poca diferencia con el mínimo grafo que puede recibir el ejercicio. Para observar esto y graficar la cota temporal armamos 2 gráficos, uno en función de n*m y otro en función de c:



Podemos ver como los dos casos, a pesar de tener cantidades de aristas muy dispares, tardan tiempos similares. Por otro lado el caso simple parece mas susceptible a variaciones, aunque suponemos que tienen mas que ver con el uso de la cpu por parte de la pc que por el algoritmo.

Por otro lado medimos C tomando un grafo mínimo de n = 150 y calculando el tiempo del algoritmo variando la diferencia entre el mínimo peso del grafo y el máximo.



En este caso el gráfico fue el esperado, siendo que los tiempos quedaron acotados por $log_2(c)$

4 Problema 3: Reconfiguraciónon de rutas

4.1 Introducción

Se tiene una ciudad con rutas construidas y rutas sin construir,cada ruta tiene un consto 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 Primm 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 negativo que no pertenecían al árbol, es decir las que se demolieron.

Algorithm 5 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 6 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 7 USD Find

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

Algorithm 8 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 el problema se reduce a encontrar un árbol generador mínimo en el grafo.

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 algoritmos 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

Se realizaron experimentos sobre diferentes tipos de instancias clasificadas en:

- Mejor
- Peor
- Random

Donde calculamos el tiempo que tardó el programa en procesar una secuencia de entrada con pesos en las aristas de una forma determinada para cada instancia. Luego, se realizó un Curve Fitting para demostrar que nuestro algoritmo cumple con la complejidad estimada por la teoría.

4.6.1 Mejor Caso

El mejor caso es cuando los pesos de las aristas son asignados de forma decreciente a partir de un valor K fijo.

4.6.2 Peor Caso

Este caso se da cuando cuando pesos de las aristas son agregados de manera tal que se cruce una secuencia creciente con otra decreciente

4.6.3 Casos Random

Los casos random son instancias generadas con valores de entre 1 y 500.

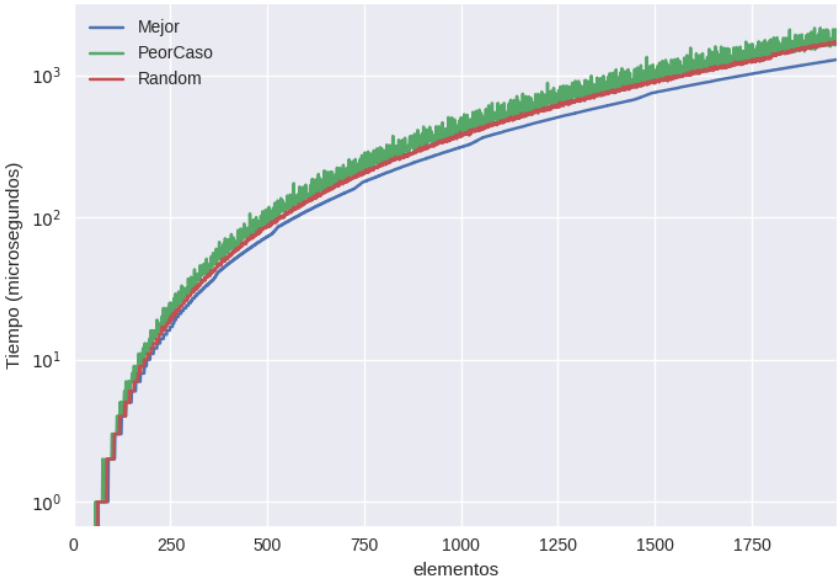


Figure 7: Tiempos del peor caso comparados con la complejidad demostrada

4.7 Fitting

Para demostrar que la complejidad es la pedida lo que hemos hecho fue, tomar cualquier curva de las 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, lo cual implica que que la complejidad es la correcta.

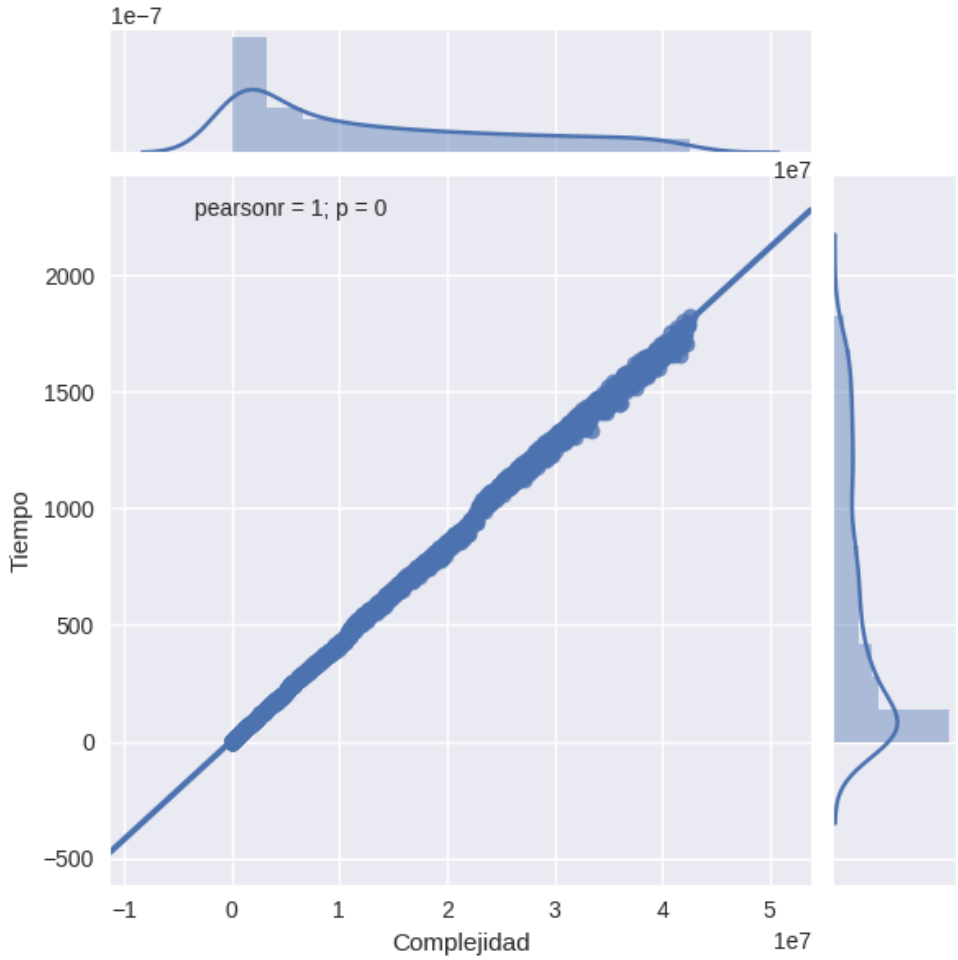


Figure 8: Pearson para una curva experimentada contra $n^2log(n)$