DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

18 de junio de 2017

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Grupo "Dijkstraidos"

Integrante	LU	Correo electrónico
Barylko, Roni Ariel	750/15	rbarylko@dc.uba.ar
Giudice, Carlos	694/15	cgiudice@dc.uba.ar
Szperling, Sebastián Ariel	763/15	sszperling@dc.uba.ar
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Delivery óptimo

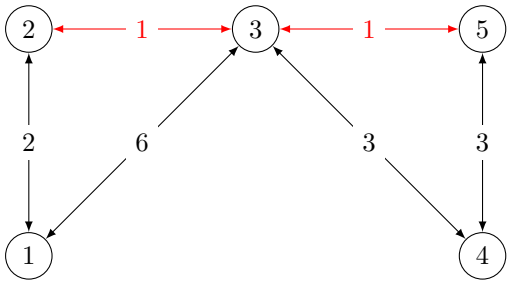
1.1. Descripción del problema

El problema plantea una provincia en la cual las ciudades están conectadas por dos tipos de rutas: las comunes y las premium. En ambos casos, las rutas son bidireccionales, conectan dos ciudades y tienen asociada una distancia no negativa.

A través de estas rutas, nuestra empresa busca transportar mercadería desde una ciudad origen a una ciudad destino, de manera que se recorra la menor distancia posible considerando que, por regulaciones provinciales, solo se puede pasar por k rutas premium en este recorrido (es decir, que la suma de distancias de las rutas utilizadas sea la menor posible, utilizando entre 0 y k rutas premium). La complejidad del algoritmo debe ser no peor que $\mathbf{O}(n^2k^2)$ donde n es la cantidad de ciudades y k la máxima cantidad de rutas premium utilizables.

Para ver mejor el problema, pongamos un ejemplo. Supongamos que tenemos las ciudades 1 a 5, y existen las siguientes rutas:

- Ruta común de 1 a 2, con una distancia de 2 km.
- Ruta común de 1 a 3, con una distancia de 6 km.
- Ruta premium de 2 a 3, con una distancia de 1 km.
- Ruta común de 3 a 4, con una distancia de 3 km.
- Ruta premium de 3 a 5, con una distancia de 1 km.
- Ruta común de 4 a 5, con una distancia de 3 km.



Entonces, tendríamos las siguientes respuestas para cada uno de los problemas planteados:

1. Camino mínimo de 1 a 5 con 2 rutas premium: **4**
2. Camino mínimo de 1 a 5 con 1 rutas premium: **7**
3. Camino mínimo de 1 a 5 con 0 rutas premium: **12**
4. Camino mínimo de 3 a 5 con 1 rutas premium: **1**
5. Camino mínimo de 3 a 5 con 0 rutas premium: **6**

1.2. Desarrollo

Dado este problema, podemos modelarlo utilizando grafos. Así, cada ciudad se representaría con un nodo, y cada una de las rutas que conecta dos ciudades, con una arista. Del mismo modo, el problema pasaría a ser alcanzar el camino mínimo de un nodo origen a un nodo destino, utilizando como máximo k aristas premium; y dado que las rutas son bidireccionales, podemos utilizar un grafo común no dirigido para esta representación.

A partir de esta representación, podemos poner el foco en las dificultades que trae consigo el problema. Sabemos que utilizando un algoritmo de camino mínimo, obtendríamos el camino más corto desde el nodo origen al nodo destino, pero nada sabríamos sobre cuantas rutas premium se están utilizando. De este modo, si bien tenemos una aproximación buena del problema, tenemos que buscar la manera de poner en consideración las rutas premium y su uso.

Un buen punto de inicio es considerar que, si bien tenemos un límite de rutas premium, podríamos encontrar un mejor camino que utilice una menor cantidad. Por ende, no alcanza con encontrar el camino mínimo que utilice k rutas premium, sino que debemos encontrar el mínimo de todos aquellos que utilicen hasta k premium. Es decir que, en lugar de plantearlo como un único problema, podríamos ver el ejercicio como la mejor solución de k subproblemas distintos, donde el i-ésimo subproblema nos da el camino mínimo utilizando exactamente i rutas premium.

Por otro lado, aún buscando resolver solo uno de los subproblemas planteados, estaríamos frente a la posibilidad de alcanzar el límite de rutas antes de recorrer todas (y por ende, de estar tomando una decisión errónea). Debemos, entonces, buscar la manera de llegar al último nodo habiendo elegido las mejores rutas premium posibles, siendo que no haya forma de tomar una ruta premium distinta y a través de ella se consiga un camino de menor distancia.

Para enfrentar este problema, podríamos generar un grafo con k níveles, donde cada nivel sea una copia exacta del grafo original, y el nodo J perteneciente al i-ésimo nivel represente que para llegar a ese nodo se utilizaron i rutas premium desde el nodo origen. De este modo, pasamos a tener un grafo de nk nodos, donde cualquiera de los caminos entre el nodo origen y los k nodos destinos pueden representar una respuesta válida.

Sin embargo, modelando el grafo de esta manera tendríamos problemas, puesto que nuestro grafo inicial era no dirigido y nuestra idea es que cada nivel represente la cantidad de rutas premium utilizadas hasta el momento. Por ende, no podría ocurrir que se pase de un nodo perteneciente del nivel $i + 1$ a un nodo del nivel i, ya que esto implicaría que, al utilizar una ruta premium más, disminuyó el nivel (es decir que disminuyó la cantidad de rutas premium utilizadas, siguiendo la idea de nuestro modelo, lo cual sería absurdo). Por lo tanto, debemos modelar el nuevo grafo de manera que:

1. Debe haber un nivel por cada posible respuesta
2. Cada vez que se utiliza una ruta premium, se sube de nivel
3. Cada vez que se utiliza una ruta que no es premium, se mantiene el mismo nivel
4. Se deben mantener las conexiones del grafo original (debe ser una representación fiel)

Dadas estas condiciones, tendremos un grafo con k niveles, como habíamos establecido anteriormente, pero debido a que hay rutas que solo se pueden recorrer en una dirección, tendremos que usar un **grafo dirigido**. En él, estableceremos dos tipos de relaciones, que aplicarán para todos los niveles del grafo.

La primer relación a definir será entre los nodos conectados por una arista común. No es dificil ver que, si en el grafo original había una arista común entre el nodo J y el nodo H de peso P, ahora habrá k representaciones de esa misma arista, para los nodos J_i y H_i con i entre 0 y k, donde el peso de cada una de ellas será P. Al ser nuestro nuevo grafo un digrafo, y como establecimos que cuando no se usa una ruta premium se mantiene el mismo nivel, bien podría ocurrir que nuestro camino vaya de J_i a H_i o viceversa, por lo que utilizaremos una arista para cada una de estas posibilidades. Es decir que, para cada nivel i, habrá una arista dirigida que conecte J_i con H_i y una que conecte H_i con J_i. De este modo, quedan definidas de manera correcta todas las aristas no premium pertenecientes al grafo inicial.

La segunda relación a definir será entre los nodos conectados por una arista premium. Al igual que en el otro caso, por cada arista premium habrá k representaciones de esta arista. Sin embargo, como al tomar una ruta premium debemos aumentar el nivel, hay que establecer una relación entre el i-ésimo y el (i+i)-ésimo nivel. Así, si en el grafo inicial había una arista premium entre los nodos J y K, ahora habrá una arista dirigida de J_i a $K_i + 1$ y una de K_i a $J_i + 1$. Notemos que, al tener una arista no dirigida en el grafo inicial, debe ser posible recorrer la ruta en ambas direcciones.

De esta manera, definimos nuestro grafo dirigido de k niveles de manera que represente correctamente el grafo inicial y, por consecuente, a nuestro problema. Para dejar más en claro como se produce la creación del grafo, planteamos el siguiente algoritmo, donde consideramos que los ejes funcionan del mismo modo para grafos dirigidos y no dirigidos, y el grafo es el que distingue si una arista funciona de manera bidireccional o no; y que la estructura de eje es:

- nat primerEje
- nat segundoEje
- nat peso

agregarEjeComun (**in** e: `eje`, **in** n: `nat`, **in** k: `nat`, **in/out** $grafoConNiveles$: `digrafo`)

```
for c ≤ k do
    nuevoPrimerNodo = e.primerNodo + c × n
    nuevoSegundoNodo = e.segundoNodo + c × n
    agregarEjeADigrafo(nuevoPrimerNodo, nuevoSegundoNodo, e.peso, grafoConNiveles)
    agregarEjeADigrafo(nuevoSegundoNodo, nuevoPrimerNodo, e.peso, grafoConNiveles)
end for
```

agregarEjePremium (**in** e: `eje`, **in** n: `nat`, **in** k: `nat`, **in/out** $grafoConNiveles$: `digrafo`)

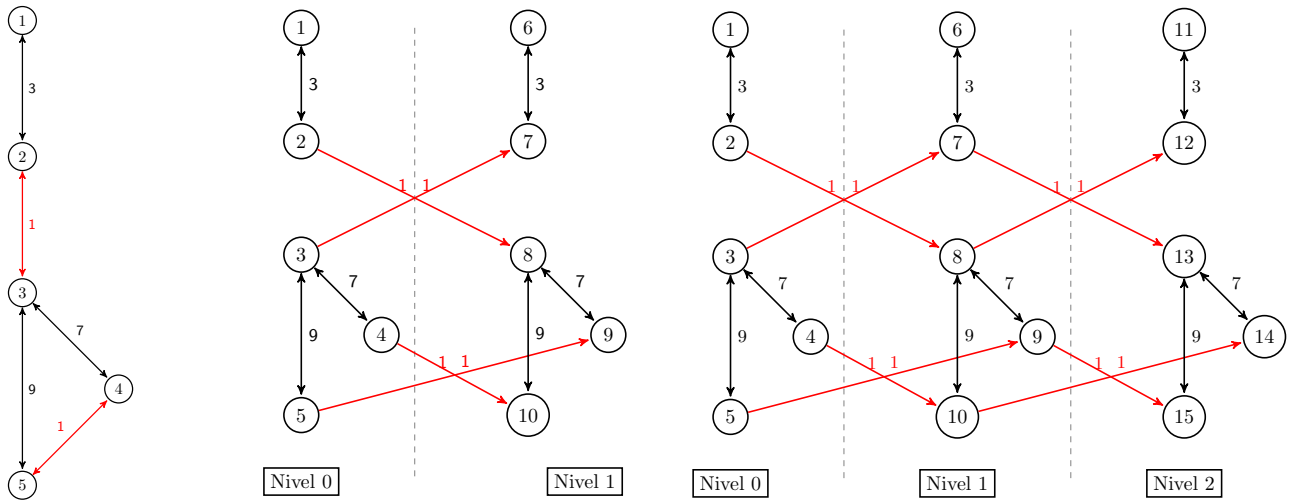
```
for c < k do
    nuevoPrimerNodo = e.primerNodo + c × n
    nuevoSegundoNodo = e.segundoNodo + c × n
    agregarEjeADigrafo(nuevoPrimerNodo, (nuevoSegundoNodo + n), e.peso, grafoConNiveles)
    agregarEjeADigrafo(nuevoSegundoNodo, (nuevoPrimerNodo + n), e.peso, grafoConNiveles)
end for
```

Con esta nueva representación, tendríamos un grafo similar al que se ve en imágen, donde el número que representa a cada nodo se define como:

$$J_i = J + c \times n$$

siendo J_i el número del nodo a agregar, J el valor del nodo original, c el nivel actual y n la cantidad de nodos en el grafo. Es así que, H y F representan al mismo nodo $\Leftrightarrow H \equiv F \pmod{n}$.

Grafo original **Con K = 1** **Con K = 2**



Dado que ya tenemos un grafo que nos permite identificar la cantidad de rutas premium utilizadas por la manera en la que está caracterizada cada nodo, basta con conocer nuestro nodo origen y buscar el camino más corto hasta las k posibles soluciones, para luego quedarnos con la mejor solución.

Por lo tanto, como sabemos que el nodo origen necesariamente estará situado en el primer nivel (es decir, que es único para nuestros propósitos), podemos utilizar un algoritmo que nos de el camino mínimo de un nodo a todos los demás y luego evaluar solo aquellos que nos den una respuesta válida a nuestro problema (un camino de origen a destino).

En consecuencia, podríamos usar tanto el algoritmo de Bellman-Ford como el de Dijkstra. Sin embargo, considerando que no tenemos aristas con pesos negativos, y que el uso de Dijkstra nos facilita el cumplimiento dela complejidad, nos quedaremos con este algoritmo para realizar la búsqueda.

Por lo tanto, con lo visto anteriormente, acabaríamos teniendo un algoritmo de este estilo:

deliveryOptimo (**in** *origen*: `nat`, **in** *destino*: `nat`, **in** n: `nat`, **in** k: `nat`, **in** *ejesComunes*: `lista[ejes]`, **in** *ejesPremium*: `lista[ejes]`) → res: nat

```
digrafo: grafoConNiveles ← crearDigrafoVacio
for e ∈ ejesComunes do
 | agregarEjeComun(e, n, k, grafoConNiveles)
end for
for e ∈ ejesPremium do
 | agregarEjePremium(e, n, k, grafoConNiveles)
end for
vector[nat]: distancias ← dijkstra(grafoConNiveles, origen)
for i ≤ k do
 | if distancias[destino + i × n] < res then
 |  | res ← distancias[destino + i × n]
 | end if
end for
```

1.3. Cota temporal

La cota temporal de este problema se puede reducir, como vemos en el algoritmo de deliveryOptimo que aparece en la sección **Desarrollo**, a conocer las complejidades de tres algoritmos:

- agregarEjeComun
- agregarEjePremium
- Dijkstra

Veamos cual es la complejidad de cada uno de ellos:

agregarEjeComun

Este algoritmo toma un eje, un valor n y un valor k, y agrega el mismo eje para cada uno de los niveles desde 0 a k. Por lo tanto, como agregar un eje se compone de tres asignaciones de costo $O(1)$, el costo del algoritmo se reduce al for que se encarga de realizar la misma acción para todos los niveles. Por lo tanto, su costo es $O(k)$, donde k representa el límite de rutas premium que se puede utilizar.

agregarEjePremium

Este algoritmo realiza las mismas acciones que $agregarEjeComun$, pero reasigna los ejes de manera distinta. Sin embargo, como aún realiza asignaciones de costo $O(1)$, podemos reducir la complejidad otra vez al ciclo que posee. Por lo tanto, como vimos con el algoritmo anterior, el costo es $O(k)$, donde k representa el límite de rutas premium que se puede utilizar.

Dijkstra

Implementamos el algoritmo de Dijkstra usando una cola de prioridad y listas de adyacencias. Internamente la cola de prioridad la representamos con Fibonacci Heap(de la biblioteca boost), este nos aporta inserción y disminución de la prioridad de un elemento en $O(1)$ en peor caso, y extracción del mínimo en $O(log(n))$ amortizado y $O(n)$ en peor caso (con n la cantidad de elementos que hay en el heap). La listas de adyacencias las implementamos con un array de listas de la STD, por lo que la complejidad del algoritmo de Dijkstra es de $O(m + nlog(n))$ con m la cantidad de aristas y n la cantidad de nodos.

Ahora, queda ver como nos afectan estas complejidades al algoritmo de deliveryOptimo. En este algoritmo, usamos dos ciclos: uno para los ejes comunes, y uno para los ejes premium. Como sabemos que la cantidad total de ejes es m, y que el costo de agregarEjePremium y de agregarEjeComun es $O(k)$, al aplicarle uno de estos dos algoritmos a todos los ejes tenemos una complejidad de $O(mk)$.

Luego, generamos el vector de distancias aplicandole Dijkstra al grafo con $k+1$ niveles que generamos. Sin embargo, como nuestro grafo tiene $k + 1$ repeticiones del grafo inicial, acaba teniendo k veces n nodos; es decir, nk nodos en total y esto mismo nos va a pasar con las aristas, donde acaba teniendo $2mk$ ejes en total (el 2 proviene de que al grafo original lo representamos con un dígrafo). Por lo tanto, al aplicarle Dijkstra, tendrá un costo de $O(mk + n * log(n))$.

Finalmente, recorremos todos los niveles para buscar el camino mínimo que va de origen a destino y utiliza entre 0 y k rutas premium. Esto implica un último ciclo, donde se recorren las k posibles soluciones, lo que nos da un costo de $O(k)$.

En resumen, tenemos tres costos significativos en nuestro algoritmo:

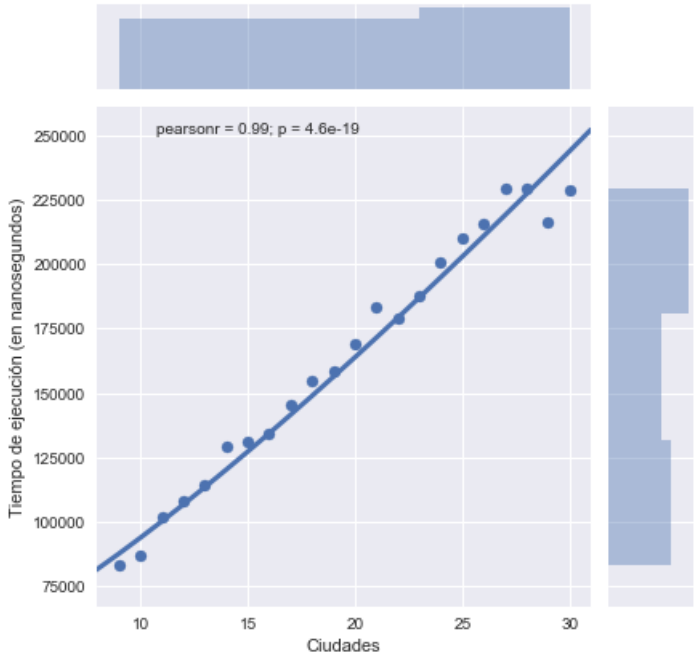
- $O(mk)$
- $O(mk + n * log(n))$
- $O(k)$

Por lo que el algoritmo nos termina quedando $O(mk+mk+n*log(n)+k)$ que a su vez es $O((2m+1)k+n*log(n))$ por lo que el algoritmo tiene una complejidad de $O(mk+n*log(n))$. Como sabemos la cantidad de aristas está acotada por la cantidad de nodos al cuadrado, osea k^2n^2 por lo que la complejidad se ajusta a lo pedido $O(k^2n^2)$.

1.4. Experimentacion

De cara a la experimentación, nuestra expectativa es que la performance del algoritmo se vea afectada por dos factores: la cantidad de ciudades (n) y la cantidad máxima de rutas premium a utilizar (k). Consideramos que la cantidad de rutas (m) no debería afectar los tiempos por la representación interna escogida al correr Dijkstra (matriz de adyacencia). Por un motivo similar, asumimos que la cantidad de rutas premium existentes tampoco debería influir en el timepo de ejecución, ya que es k quien determina la cantidad de niveles del supergrafo.

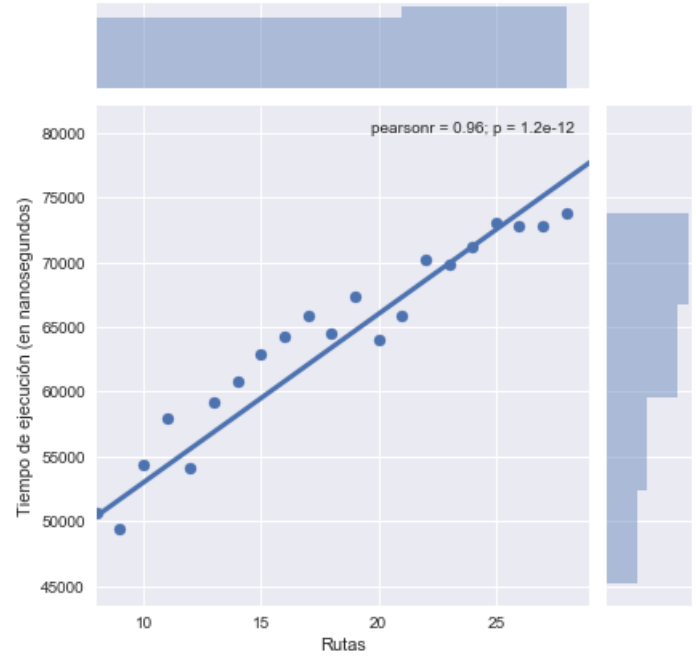
La metodología de prueba y medición de tiempo se encuentra detallada en los apéndices. Los siguientes gráficos corresponden a cada una de las variables aisladas, manteniendo todas las otras fijas en un mismo numero en tanto esto es posible.



Datos del gráfico

Rutas totales	m = 30
- Premium disponibles	p = 4
- Premium utilizadas	k = 2
Curva aproximada	$f(x) = 1900x * log(x) + 50000$

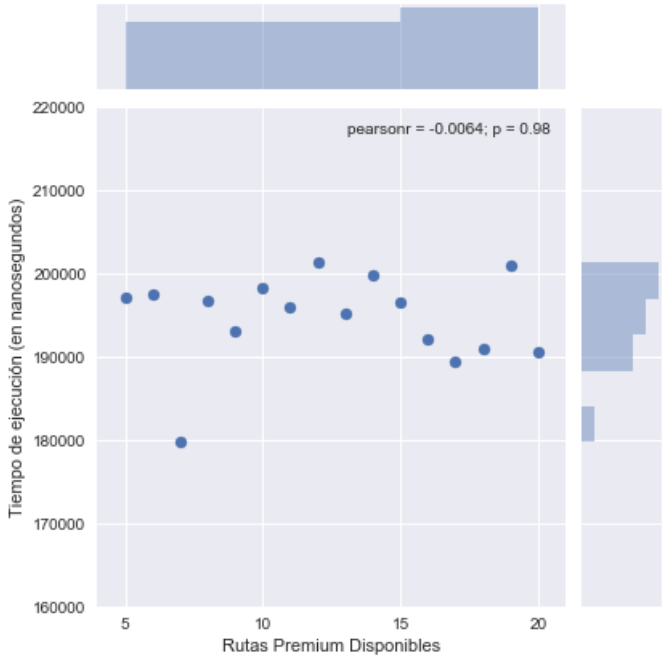
Como se puede apreciar en este gráfico, el tiempo de ejecución depende fuertemente la cantidad de ciudades. Esta parte del tiempo de ejecución corresponde al algoritmo de Dijkstra, cuya complejidad bajo nuestra implementación es $O(mk + n * log(n))$, ya que el resto del algoritmo no depende de n. En este análisis, m y k son constantes, lo que nos deja una curva que responde a la complejidad de $O(n * log(n))$.



Datos del gráfico

Ciudades totales	n = 8
Rutas Premium	
- disponibles	p = 4
- utilizadas	k = 2
Curva aproximada	$f(x) = 1300x + 40000$

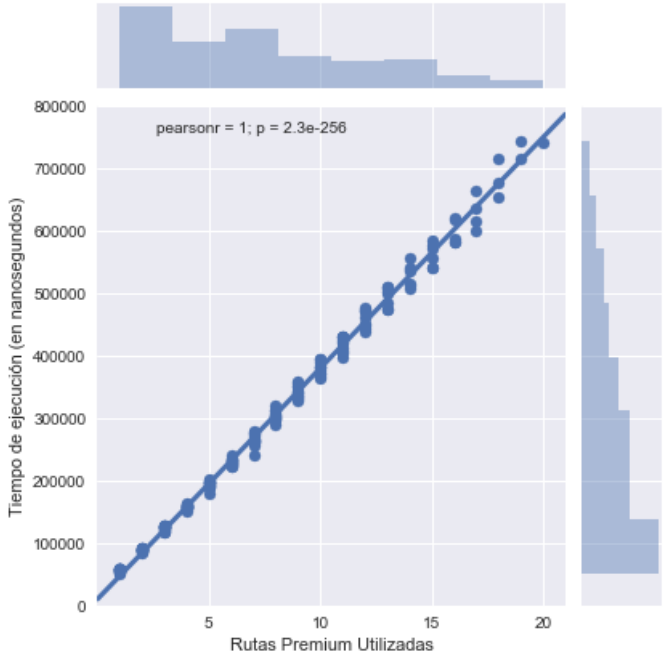
En este gráfico podemos ver el otro lado de la complejidad en Dijkstra, al igual que el impacto de la generación del supergrafo. Esto se condice con el costo $O(mk)$ previamente mencionado para ambos proceso.



Datos del gráfico

Ciudades totales	n = 10
Rutas	
- totales	m = 30
- premium utilizadas	k = 5

Como esperábamos, la cantidad de rutas premium totales no parece influir en la complejidad si no modificamos la cantidad total de rutas o los niveles del supergrafo. En particular, al analizar con el coeficiente de correlacion de Pearson, el valor es muy próximo a cero, lo cual indica que los valores son independientes.



Datos del gráfico

Ciudades totales	n = 10
Rutas totales	m = 30
Curva aproximada	$f(x) = 37000x + 10000$

Por otro lado, al aumentar la cantidad máxima de rutas premium utilizables, el impacto es nuevamente lineal. También se puede ver que la correlación lineal entre tiempo de ejecución y k es perfecta, con un coeficiente igual a 1 y un p-valor ínfimo indicando que la correlación debe existir. Ya que los 3 costos mencionados dependen de k, esto era esperado.

A su vez, en este gráfico no se limitó el total de rutas premium existentes, y se puede ver nuevamente que dicho valor no influye en la complejidad, dada la baja dispersión de los puntos.

2. Subsidiando el transporte

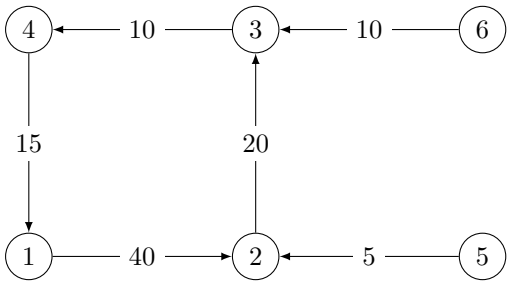
2.1. Descripción del problema

En este problema, observamos la provincia de Optilandia, cuyas ciudades estan conectadas por rutas de una sola dirección, donde no necesariamente se puede llegar de una ciudad a todas las demás. Sin embargo, sabemos que desde cualquier ciudad se puede llegar, al menos, a otra ciudad. Cada una de estas rutas tiene una cabina de peaje y, por ende, recorrer cada una de ellas tiene un costo. Sin embargo, por decisiones gubernamentales, cada uno de estos peajes se vio reducido por un costo fijo c (si antes la ruta A valía A1, y la ruta B valía B1, ahora valen A1 - c y B1 - c respectivamente), pudiendo generar que una ruta no solo no le cobre a sus usuarios, sino que acabe dándole dinero.

Si bien esto no es un problema para el gobierno, siempre y cuando se evite que un usuario pueda irse desde una ciudad, hacer un recorrido y volver a la misma habiendo ganado plata. Por lo tanto, como el gobierno busca maximizar el subsidio otorgado, debemos buscar el valor c que permita otorgar el mayor subsidio por peaje sin que exista la posibilidad de que un usuario le saque plata al Estado. La complejidad del algoritmo debe ser no peor que $\mathbf{O}(nm.log(c))$, donde n es la cantidad de ciudades, m es la cantidad de rutas y c es el costo del máximo peaje.

Veamos un ejemplo del problema. Supongamos que tuvieramos 6 ciudades, con las siguientes rutas iniciales y con los siguientes costos de peaje1 expresados en pesos:

- Ruta de 1 a 2. Costo de peaje: 40 pesos.
- Ruta de 2 a 3. Costo de peaje: 20 pesos.
- Ruta de 3 a 4. Costo de peaje: 10 pesos.
- Ruta de 4 a 1. Costo de peaje: 15 pesos.
- Ruta de 5 a 2. Costo de peaje: 5 pesos.
- Ruta de 6 a 3. Costo de peaje: 10 pesos.



Como vemos, la única manera de salir de una ciudad y volver a la misma es que arranquemos en las ciudades 1, 2, 3 ó 4, y recorramos las cuatro rutas que las conectan. Por ende, debemos asegurar que al recorrerlas no se le saca plata al estado; es decir, que al aplicarle el subsidio fijo, se descuenta menos de (40+20+10+15) = 85 pesos. Considerando que las cuatro rutas tienen el mismo descuento, el mayor subsidio que se podría realizar es la cuarta parte de 85, que si lo redondeamos es 21 pesos. Por lo tanto, el mayor descuento que se podría realizar a las rutas de esta provincia es de 21 pesos.

2.2. Desarrollo

Dado este problema, y considerando que hay que tener en cuenta la mano en la cual corren las rutas, la mejor manera de modelarlo sería utilizando digrafos. Así, cada ciudad se representaría con un nodo, y cada una de las rutas que conecta dos ciudades, con una arista dirigida. Consideraremos que un recorrido abusivo es representado por un ciclo negativo en el digrafo.

Como nuestro problema principal es averiguar cual es el mayor subsidio que se le puede otorgar a todas las rutas sin que se generen ciclos de rutas negativos, no es dificil que rápidamente encontremos una visión certera de lo que debemos hacer en el problema. Como primer aproximamiento, podemos asegurar que nuestro objetivo será reducir el costo de las rutas de manera que vayamos obteniendo mejores valores hasta que, superado el valor máximo, encontremos ciclos negativos en nuestro grafo.

Por ende, sabemos de entrada que necesitaremos un algoritmo capaz de reconocer ciclos negativos; y considerando los límites de complejidad brindados, concluímos que Bellman-Ford cumple nuestros propositos. Este algoritmo nos permitirá utilizar aristas negativas, y averiguar en cada uno de los nodos si en sus componentes hay o no ciclos negativos.

Dado que en el problema original queremos detectar la existencia de ciclos negativos para distintas versiones del mismo grafo, usamos una función que permite crear una nueva versión del grafo a partir del original. Diremos que una p-versión nueva del grafo contiene la misma cantidad de nodos y representa al mismo conjunto de adyacencias, y su diferencia radica en que para todo eje de u a v con peso w perteneciente al grafo original, hay un eje de u a v con peso (w-p) perteneciente a la p-versión. Así, utilizaremos un algoritmo que, para cada versión, nos diga si efectivamente al aplicarle Bellman-Ford se encuentran o no ciclos negativos (para esto, utilizamos una versión del algoritmo que nos devuelve *true* si encuentra ciclos negativos, y $false$ si no lo hace)

Del mismo modo, dado que Bellman-Ford parte de un nodo s y actualiza las distancias entre s y el resto de los nodos, solo puede detectar ciclos negativos si estos pertenecen a la misma componente conexa que s. Si el grafo G de

entrada no es conexo, no existe nodo de partida que sirva para reconocer a todas las componentes de G. Por ende, proponemos la creación de un nodo con $d_{out} = n \wedge d_{in} = 0$. En otras palabras, para todo nodo v perteneciente a G, existe un eje que va de s a v. Sin embargo, para que el mismo no afecte el resultado final, deberemos agregarlo luego de realizar la p-versión del grafo original (es decir que, sea cual sea la p-versión, las aristas que conecten a s con cualquier otro nodo tendrán peso nulo)

A partir de esto, el algoritmo de Bellman-Ford puede utilizar a s como nodo de partida y tendremos garantizado que reconoceremos todos los ciclos negativos del grafo. De este modo, en un grafo con n nodos y m aristas, pasaríamos a tener $n+1$ y $m+n$ aristas.

```
ajusteParaBellmanFord (in p: int, in ejesGrafo: lista[ejes], in tamano: int) → res: bool
  lista[ejes] ejesPVersion ← ajustarEjes(ejesGrafo, p)
  lista[ejes] ejesPVersion ← agregarEjeGeneral(ejesPVersion, tamaño)
  res ← bellmanFord(ejesPVersion, tamaño+1)
```

Es importante aclarar que, al pasarle como parámetro *(tamaño+1)* a Bellman-Ford, estamos enviando el nodo de origen desde el cual se correrá el algoritmo (el cual es, justamente, el último nodo que colocamos nosotros)

Diremos que c es el peso de la arista mas pesada del grafo original. Veamos que la versión p = 0 es equivalente al grafo original, y por ende no puede tener ciclos negativos. Por otro lado, veamos que si el grafo original tiene ciclos, la versión p = c + 1 tendrá ciclos negativos, porque todas sus aristas serán negativas. Por ende, sabemos que nuestra solución esta acotada inferiormente por 0 y superiormente por c.

Por otro lado, teniendo en cuenta un Q cualquiera, sabemos que si la versión Q carece de ciclos negativos entonces toda versión con un valor de subsidio menor a Q también carecerá de ellos. La intuición aquí reside en notar que aumentar los pesos de un grafo sin ciclos negativos producirá el aumento del peso de todos sus ciclos, y por lo tanto, que todos ellos continuen sin ser negativos. Del mismo modo, podemos notar lo inverso: si la versión Q posee ciclos negativos, toda versión con un valor de subsidio mayor a Q los contendrá.

Entonces, sabiendo que un valor Q bien nos habla de todos los mayores o menores a él; y considerando que nuestra solución se encuentra entre 0 y c, podemos realizar una búsqueda binaria tal que para cada versión Q, si la misma posee ciclos negativos, nuestra solución se acotará entre 0 y Q. En cambio, si no los posee, su solución se encontrará entre Q y c.

Por lo tanto, con lo ya expuesto, podemos utilizar el algoritmo con ajuste de Bellman-Ford que expusimos anteriormente para averiguar cual es el máximo subsidio realizable, aplicando una suerte de búsqueda binaria sobre los valores.

```
subsidioDeRutas (in n: nat, in ejesGrafo: lista[ejes]) → res: nat
  nat: cotaInferior ← 0
  nat: cotaSuperior ← pesoMax(ejesGrafo)
  while cotaInferior < cotaSuperior do
  |  nat: nuevaCota ← (cotaInferior + cotaSuperior) / 2
  |  bool: tieneCiclos ← ajusteBellmanFord(ejesGrafo, nuevaCota)
  |  if tieneCiclos then
  |  |  cotaSuperior ← nuevaCota
  |  else
  |  |  cotaInferior ← nuevaCota
  |  end if
  end while
  res ← cotaInferior
```

2.3. Cota temporal

Luego del análisis realizado, y viendo el algoritmo subsidioDeRutas dado para el problema, el conjunto de operaciones significativas para deducir la complejidad se reduce al *while* en el cual, con la ayuda del Bellman-Ford con ajuste, acabamos obteniendo el mayor subsidio posible.

Analicemos, entonces, esta porción del algoritmo. Buscamos un valor entre 0 y c, como habíamos dicho inicialmente, y cada vez que se corre el ciclo reducimos esta búsqueda a la mitad de los valores (0 y Q, o bien Q y c). Por ende, tenemos un *while* que corre en $O(log(c))$, dándonos como resultado final $O(log(c).BF))$ donde BF es la complejidad del Bellman-Ford con ajuste.

Pasemos a analizar este otro algoritmo. El Bellman-Ford con ajuste realiza 3 operaciones: reajusta el valor de todos los ejes, agrega un nodo con n ejes de peso nulo y finalmente corre Bellman-Ford desde este nodo agregado. Es facil ver que las primeras dos operaciones se reducen a recorrer todos los ejes y todos los nodos, respectivamente, y por

ende corren en $O(m)$ y $O(n)$; mientras que el algoritmo de Bellman-Ford, si bien tiene complejidad $O(n.m)$ conocida, se puede ver afectada al agregar un nodo de grado n a nuestro grafo, por lo que se requiere un análisis más exhaustivo.

Veamos lo que ocurre con nodos y aristas respectivamente. Al tener n nodos y m aristas, y agregar un nuevo nodo con n aristas nuevas, pasamos a tener $n+1$ nodos y $m+n$ aristas. Por ende, si repasamos las complejidades obtenidas con estos valores, el Bellman-Ford con ajuste tendrá una cota temporal de $O(m) + O(n) + O((n+1)(m+n)) \subseteq O(n(m+n)) \subseteq O(n^2 + n.m)$.

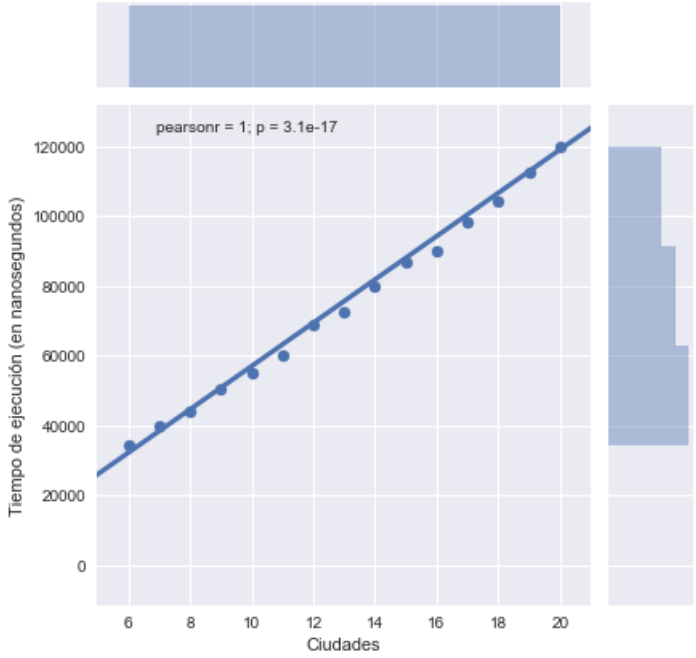
Sin embargo, tenemos como premisa que todo nodo perteneciente al grafo debe tener grado de salida mayor o igual a 1 ($d_{out} \geq 1$). Por lo tanto, por cada nodo del grafo original tenemos al menos una arista, y podemos acotar $m \geq n$. De este modo, podemos asegurar las siguientes relaciones: $O(n) \subseteq O(m) \rightarrow O(n^2) \subseteq O(n.m)$.

Por lo tanto, si $O(n^2) \subseteq O(n.m)$, podemos reducir la complejidad del Bellman-Ford con ajuste de $O(n^2 + n.m)$ a $O(n.m + n.m)$ que es exactamente $O(n.m)$. De este modo, demostramos que agregar el nodo con n aristas en el contexto de este problema no afecta la complejidad del algoritmo de Bellman-Ford.

Entonces, como habíamos dicho anteriormente, la complejidad de nuestro problema se reduce a $O(log(c).BF))$ donde BF es la complejidad del Bellman-Ford con ajuste. Y como vimos, la complejidad de este último algoritmo es de $O(n.m)$, por lo que nuestra cota temporal es finalmente $O(log(c).n.m)$, que cumple las condiciones de cota temporal exigidas por el ejercicio.

2.4. Experimentacion

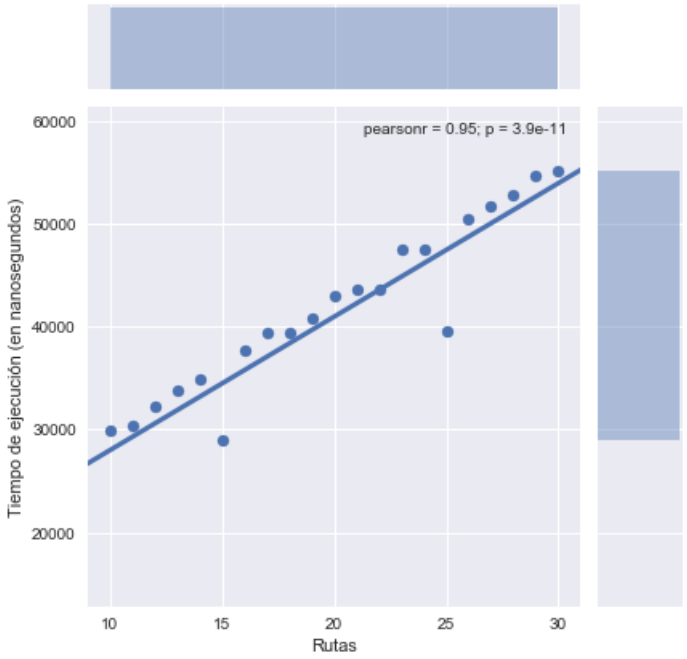
La experimentación de este ejercicio representó un desafío, ya que en la generación de grafos resultó difícil garantizar condiciones que diesen resultados homogéneos.



Datos del gráfico

Rutas	m = 30
Peso máximo	c = 30
Curva aproximada	$f(x) = 6200x - 5000$

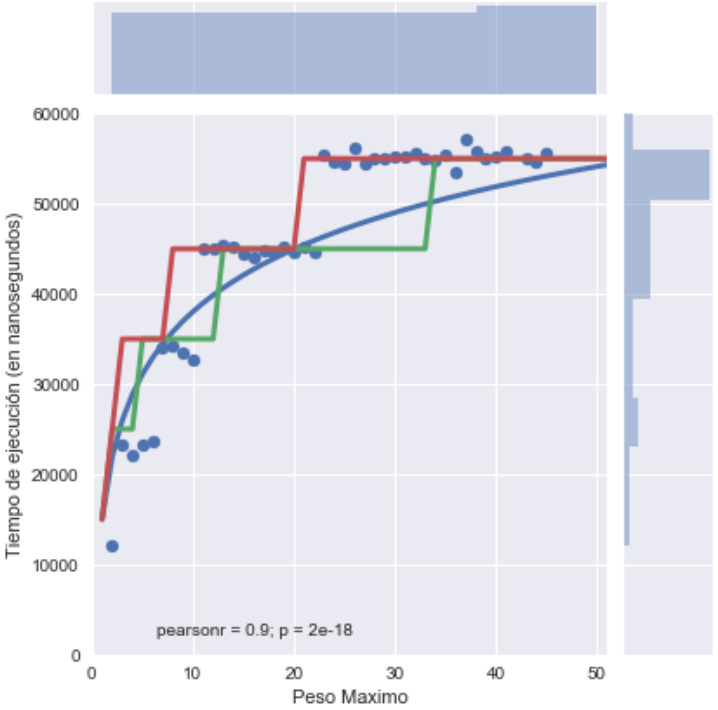
En este gráfico se puede observar que el algoritmo performa de manera perfectamente lineal con respecto a las ciudades. Nuestro analisis dio un coeficiente de correlación igual a 1, con muy bajo margen de error. Esto es esperable, ya que más allá de la distribución interna del grafo, el algoritmo de Bellman-Ford itera n + 1 veces para relajar las distancias, y los ejes adicionales que agregamos desde el nuevo nodo universal dependen de la cantidad de ciudades, no de las rutas entre ellas.



Datos del gráfico

Ciudades	n = 10
Peso máximo	c = 30
Curva aproximada	$f(x) = 1300x + 15000$

A su vez, podemos que la cantidad de rutas afecta el algoritmo de manera similar. En este caso, nuestras mediciones mostraron más variación. Esto puede deberse a que, dependiendo de la distribución de las aristas en el grafo, puede que el algoritmo omita algunos pasos de relajación de distancias. Sin embargo, sí debe verificarlo en cada iteración, por lo cual la gran mayoría de las mediciones se mantivo consistente.



Datos del gráfico

Ciudades	n = 10
Rutas	m = 30
●	$f(x) = 10000 * log(x) + 15000$
●	$f(x) = 10000 * [log(x)] + 15000$
●	$f(x) = 10000 * \lceil log(x) \rceil + 15000$

A la hora de analizar el impacto del peso máximo, tuvimos que considerar el hecho que el logaritmo tiene valores decimales, pero nuestra implementación (que realiza una búsqueda binaria) solo puede operar con enteros. Por esto, además del gráfico de la función logarítimica, incluimos dos ajustes de los valores del logaritmo a enteros: por redondeo (expresado $[log(x)]$) y por parte entera alta o "techo" ($\lceil log(x) \rceil$).

Observando estas curvas con redondeo, podemos ver que el máximo peaje tiene impacto logarítmico sobre el tiempo de ejecución, ajustandose entre ambos redondeos mencionados, lo que condice con nuestras expectativas y con la cota de complejidad pedida.

3. Reconfiguración de rutas

3.1. Descripción del problema

Este problema plantea otra provincia de Optilandia, cuyas ciudades están conectadas por rutas pero con ciertos problemas: algunas ciudades no están conectadas y otras poseen varias maneras para viajar entre ellas. Para solucionarlo, el gobierno hará obras en las rutas de modo que haya una y sólo una forma de llegar desde cualquier ciudad a otra, construyendo nuevas rutas o destruyendo rutas existentes.

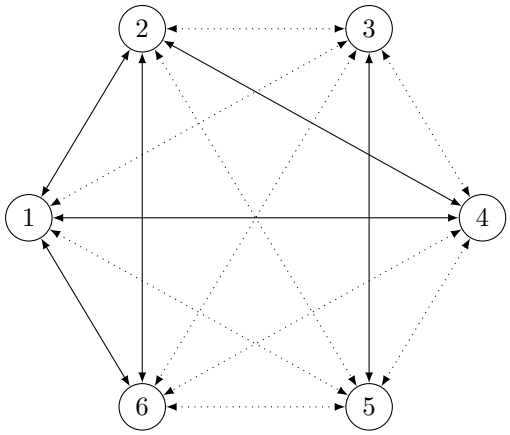
Nos piden un algoritmo que, dadas las rutas y los costos de construcción y destrucción, nos diga que rutas hay que destruir y construir para satisfacer el problema de forma que gastemos lo menor posible. Como requisito, la complejidad de este algoritmo no puede ser peor que $\mathbf{O}(n^2 log(n))$, donde n es la cantidad de ciudades de la provincia.

Veamos un ejemplo del problema. Supongamos que tuvieramos 6 ciudades, con las siguientes rutas iniciales y con los siguientes costos de destrucción expresados en pesos:

- Ruta de 1 a 2. Costo de destrucción: 100 pesos.
- Ruta de 1 a 4. Costo de destrucción: 230 pesos.
- Ruta de 1 a 6. Costo de destrucción: 65 pesos.
- Ruta de 2 a 4. Costo de destrucción: 190 pesos.
- Ruta de 2 a 6. Costo de destrucción: 300 pesos.
- Ruta de 3 a 5. Costo de destrucción: 150 pesos.

Por otro lado, tendríamos las siguientes rutas a disposición para construir, con el siguiente costo:

- Ruta de 1 a 3. Costo de construcción: 150 pesos.
- Ruta de 1 a 5. Costo de construcción: 130 pesos.
- Ruta de 2 a 3. Costo de construcción: 190 pesos.
- Ruta de 2 a 5. Costo de construcción: 50 pesos.
- Ruta de 2 a 6. Costo de construcción: 80 pesos.
- Ruta de 3 a 4. Costo de construcción: 300 pesos.
- Ruta de 3 a 6. Costo de construcción: 170 pesos.
- Ruta de 4 a 5. Costo de construcción: 200 pesos.
- Ruta de 4 a 6. Costo de construcción: 150 pesos.
- Ruta de 5 a 6. Costo de construcción: 140 pesos.



Por lo tanto, la solución más eficiente para dejar un único camino entre todas las ciudades sería:

1. Destruir la ruta de 1 a 6, a un costo de 65 pesos
2. Destruir la ruta de 1 a 2, a un costo de 100 pesos
3. Construir la ruta de 2 a 5, a un costo de 50 pesos

Lo cual nos daría un costo total de 215 pesos, siendo esta la forma más barata de dejar todas las ciudades conectadas.

3.2. Desarrollo

Como es usual, podemos representar a las ciudades como nodos de un grafo y a las rutas como los ejes. Dado que no se especifica si las rutas son de una o dos manos vamos a asumir que las rutas son bidireccionales. Como existen rutas que se pueden destruir y rutas que se pueden crear, vamos a dividir estos ejes en dos conjuntos: el conjunto C, que tendrá los ejes con los costos de construcción de rutas; y el conjunto D, que va a tener los costos de destrucción. En ambos casos, el costo de destrucción y construcción se representará como el peso del eje correspondiente a esa ruta. Notemos que $C \cup D$ contiene las aristas de un grafo completo.

Llamaremos R a nuestro grafo solución. El objetivo del algoritmo es que todas las ciudades se conecten solo de una manera, lo que en grafos sería equivalente a que existiese exactamente un camino simple entre todo par de nodos (es decir, que el grafo solución fuera un árbol). Por lo tanto, **nuestro objetivo es que R sea el conjunto de aristas de un árbol generador del grafo completo de ejes $C \cup D$.**

Una vez encontrada la solución, vamos a tener rutas que quedaron intactas y rutas que se construyeron. Al subconjunto de aristas de C que contiene las rutas que deben construirse para lograr la solución óptima lo llamaremos RC, y al subconjunto de D que contiene las rutas que hay que mantener lo llamaremos RD. Podemos notar que $D - RD$ es el conjunto de rutas que hay que destruir, y que $RD \cup RC$ nos da como resultado la representación de la provincia una vez terminado el plan, o sea $RD \cup RC = R$.

Proponemos el siguiente algoritmo:

1. Obtener los datos de entrada para crear D y C
2. Con D construir RD
3. Conseguir el RC óptimo dado RD
4. Calcular los costos de destrucción y construcción

A medida que nos informan las rutas, vamos agregandolas en dos listas de ejes, dependiendo si es una existente o si es de construcción. Una vez creadas las listas, corremos Kruskal (en este caso priorizando los ejes caros) sobre los ejes de destrucción. Es importante aclarar que, para poder utilizar el algoritmo sobre nuestro problema, implementamos Kruskal con Disjoint Set, de forma que siempre nos devuelva un bosque. Cuando finaliza el Kruskal, recorremos los ejes que quedaron y sumamos sus costos, para luego restárselos al costo total de D (ya que eso nos da el costo total de los ejes que destruimos). Después de obtener RD, corremos Kruskal de vuelta pero esta vez para buscar el árbol generador mínimo. Para esto, le tenemos que pasar el Disjoint Set que nos había quedado del anterior Kruskal (por referencia), y los ejes de las rutas que se pueden construir. Una vez más, esto nos devuelve los ejes que vamos a utilizar. Calculamos su costo, lo sumamos con el anterior y ya tenemos la respuesta final.

El esquema de demostración constará de las siguientes afirmaciones, necesarias para elegir a RD y a RC:

1. RD tiene que ser un bosque.
2. RC tiene aristas que conectan a todas las componentes de RD, formando su unión un árbol.
3. RD sera un subgrafo de D tal que para cada componente conexa $D_i \in D$, existe una componente conexa RD_i árbol generador de D_i.
4. Para encontrar al RD óptimo, es necesario que cada RD_i sea árbol generador máximo de D_i.

A continuación, las demostraciones:

1. Lo primero que podemos notar es que, si RD no es un bosque, tiene ciclos. Y por ende, si tiene ciclos, no hay manera de que agregando aristas se obtenga un árbol. Pero esto es absurdo, porque vimos que R tiene que ser un árbol. Por ende, RD tiene que ser un bosque.
2. Como R tiene que ser árbol y RD bosque, RC tiene que tener ejes que unan las componentes conexas de RD, sin formar ciclos. Además, queremos que el costo de la suma de las aristas que se agreguen sea mínimo. El algoritmo de Kruskal nos va a resolver esto, ya que en el invariante mantiene un bosque y agrega golosamente aristas que conectan componentes hasta alcanzar un árbol. Nos vamos a aprovechar de esto y vamos aplicar Kruskal partiendo del bosque conformado por las aristas de RD. Como es bosque, no rompe el invariante y en cada paso conecta a una de las componentes de RD con una arista perteneciente a C, agregando dicha arista a RC.
3. Entonces, sabemos buscar un RC óptimo dado un RD. Ahora, tenemos que buscar un RD tal que los costos resultantes sean mínimos. Para esto se nos ocurrió que RD tiene que contener las aristas de un "Bosque Generador Máximo de D", lo cual definimos como un subgrafo de D tal que para cada componente conexa $D_i \in D$, existe una componente conexa $BGM_i \in$ al BGM de D tal que BGM_i es árbol generador máximo de D_i.

   Veamos por qué el BGM(pueden ser varios) es el bosque óptimo que tenemos para asignar a RD. Tenemos a D con componentes conexas $D_1, D_2, ..., D_k$, de la cual tomamos D_i con i entre 0 y k. Al sacar un eje cualquiera de D_i, si este eje estaba en un ciclo la componente seguirá teniendo los mismos nodos, pero si no, estaríamos dividiendo D_i en dos componentes conexas, llamemoslas D_{i1} y D_{i2}, y veamos que esta división de componentes encarece la construcción de R dado este RD.

Llamemos a RD' a un bosque subgrafo de D tal que las componentes conexas tienen los mismos nodos que las componentes conexas de D y llamemos RD'' a otro tal que le quitamos un eje e, de la componente conexa D_i dividiéndola en D_{i1} y D_{i2}. A lo que queremos llegar es que, si aplicamos Kruskal para conseguir el RC óptimo, con RD'' el RC resultante tendrá los mismos ejes y uno más, que si le hubiésemos pasado RD'. Es decir, tendría exactamente los mismos ejes (por ser los de menor costo) más uno que reemplace el eje e. Por ende, al costo de agregar los ejes de Kruskal, se le sumaría el de destruir el eje e y agregar un último eje que permita la creación del árbol.Entonces, como exponemos, el costo total de construcción y destrucción sería más caro; por ende, nunca será conveniente quitar ejes que no pertenezcan a ciclos.

Cuando aplicamos el Kruskal para conseguir el RC, estamos uniendo componentes conexas. Como RD'' tiene una componente más, el RC que parte de RD'' tendrá k ejes y el que parte de RD' tendrá $k-1$, los cuales están todos incluidos en los de RD''. El algoritmo de Kruskal matiene las componentes conexas y evita que se agreguen ejes que unen nodos dentro de una misma componente para que no se formen ciclos, o sea que si unimos dos componentes tendremos una nueva que contiene a los dos.

Cuando aplicamos el algoritmo, dado RD'', en algún momento se encontrará con el problema de generar una componente conexa que contenga a D_{i1} y a D_{i2}. Para esto, el algoritmo elegirá un eje que nos combine las componentes, al cual llamaremos eje d. Al utilizar Kruskal sabemos que, tanto antes como después de agregar el eje d, los demás ejes agregados serán los mismos que en el Kruskal de RD', puesto que son los mejores posibles en ambos casos. La única diferencia reside en que en RD' existe el eje e, el que quitamos en RD'' que une D_{i1} y D_{i2}. Esto signfica que en RD' tenemos estas dos componentes conectadas sin el costo adicional de borrar e y agregar d. Por lo tanto, si llamamos A a la suma de costos de las aristas agregadas por Kruskal que comparten RD' y RD'', y B a la suma de costos de las aristas eliminadas del grafo de entrada que comparten RD' y RD'', tendríamos los siguientes valores:

$$\textbf{costo}(RD') = A + B$$
$$\textbf{costo}(RD'') = A + B + e + d$$

Y como todos los valores son positivos, porque tanto construir como destruir rutas es costoso (nunca se gana dinero) podemos afirmar que $\text{costo}(RD'') \geq \text{costo}(RD')$.

Una deducción que se obtiene de este razonamiento es que sacar mas de un eje empeora todavia más la solución, porque cada vez que le sacamos un eje a un bosque, la cantidad de componentes conexas aumenta en uno. Como dijimos, partir una componente conexa en dos empeora la solución. Podemos usar estas premisas para un razonamiento inductivo que justifica la necesidad de elegir un RD tal que para toda componente conexa $D_i \in D$ exista una componente conexa $RD_i \in RD$ que es árbol generador de D_i. En otras palabras, si destruimos el único camino existente entre dos rutas, deberemos necesariamente construir uno nuevo que vuelva a unirlas, y por lo tanto podemos concluir no solo que destruir una ruta útil es perjudicial, sino que destruir una mayor cantidad es más perjudicial aún.x

4. Ahora nos queda ver que el bosque generador máximo es el óptimo. Definimos el costo de BG, bosque generador de D, como la suma de los pesos de los ejes pertenecientes a $D - BG$ y buscamos que RD sea el bosque generador de costo mínimo. Si RD fuera distinto del BGM de D, significaría que su costo es menor que el costo del BGM, lo cual es absurdo porque este es el bosque generador obtenido a partir de la destrucción de las rutas más baratas. Entonces RD tiene que ser el BGM de D. Por ende, si usamos Kruskal de manera modificada, nos dará este mismo resultado.

3.3. Cota temporal

Para tener una cota de la complejidad temporal de nuestro algoritmo, debemos enfatizar en la complejidad de la implementación de Kruskal, ya que es el corazón de la resolución del problema. El algoritmo básicamente lo que hace es:

1. Creamos un conjunto de ejes A vacío que va terminar siendo nuestra solución
2. Crear una estructura B que contenga las componentes conexas de un bosque, inicializada con los nodos del grafo como nodos separados.
3. Creamos un conjunto C con las aristas del grafo.
4. Mientras C no sea vacío.

a) Sacamos la arista de menor o mayor peso (según los parámetros del algoritmo) de C

b) Si la arista conecta dos componentes diferentes, se la añade a A y se combinan estas componentes de B.

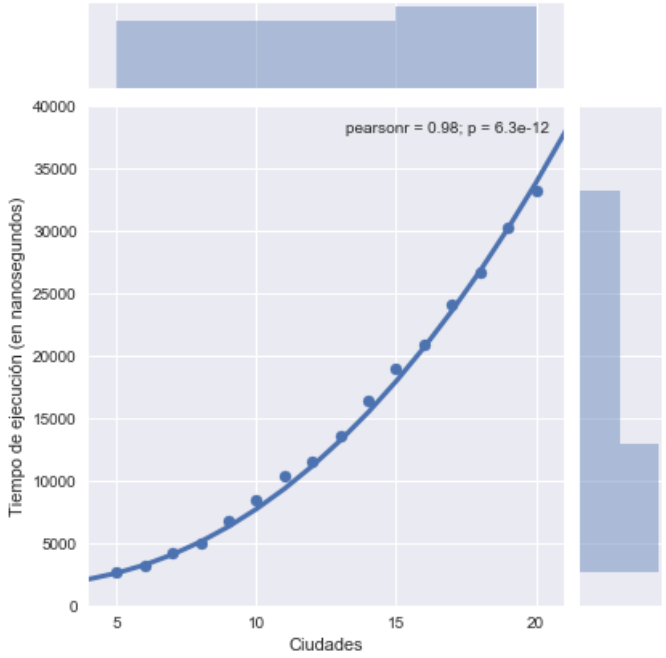
c) Caso contrario, descartamos esta arista

Primero ordenamos los ejes. Como usamos el sort de la Standard Library, en el peor de los casos este paso nos costará $O(mlog(m))$ o lo que es lo mismo $O(mlog(n))$. Después, tenemos que iterar las aristas. Esta parte está fuertemente atada a la estructura de datos que se usa internamente para el bosque, y como nosotros usamos un Disjoint Set (Conjuntos Disjuntos) y lo implementamos con un par de heurísticas **Union by Rank** y **Path Compression**, con esto podemos probar que una secuencia de k operaciones de la estructura corren en tiempo $O(k\alpha(n))$ donde α es la inversa de la función de Ackermann, por lo que la iteración de las aristas nos da una complejidad en peor caso de $O(m\alpha(n))$. De cualquier manera, esto se ve mitigado por el ordenamiento de los ejes, así que el algoritmo de Kruskal, en nuestra implementación, tiene una complejidad temporal de $O(mlog(n))$ en peor caso.

En la resolución del problema corremos Kruskal dos veces: una para el grafo original y otra para su complemento, y como en el peor de los casos tenemos n^2 aristas, estas dos corridas tiene una complejidad de $O(n^2log(n))$. Por último, tenemos que recolectar el peso de las aristas resultantes, que son $n-1$ porque es un árbol y esto lo podemos solucionar en tiempo lineal, o sea que la complejidad $O(n^2log(n))$ se ajusta a lo pedido.

3.4. Experimentacion

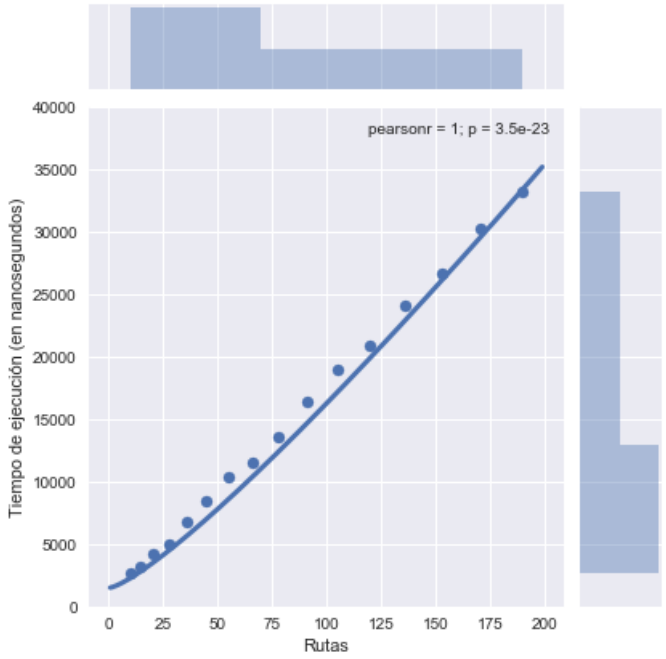
Al experimentar con este problema, nos esperabamos encontrar que el problema solo estuviese definido por la cantidad de ciudades, ya que la cantidad de rutas dependía de la misma (si unimos las rutas existentes con las potenciales, obtenemos un grafo completo). Supusimos que la proporción de rutas existentes y a construir no causaría ninguna diferencia de complejidad, ya que el algoritmo requiere que se recorran todas y la suma siempre da igual para un n dado.



Datos del gráfico

Rutas existentes	$\|D\| = 10$
Curva aproximada	$f(x) = 27x^2log(x) + 1500$

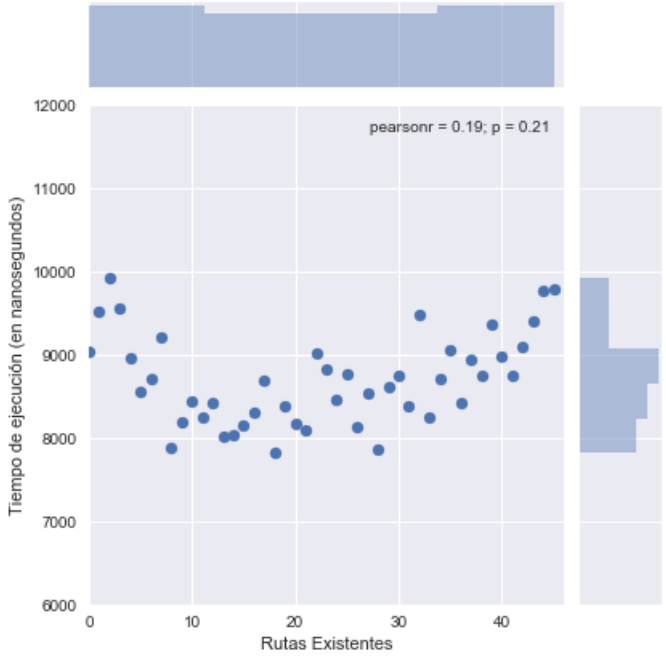
Efectivamente, la cantidad de ciudades tiene un impacto de aproximadamente $O(n^2log(n))$. Podemos analizar este mismo aspecto desde el lado de las rutas, usando las mismas mediciones y transformando el eje X:



Datos del gráfico

Rutas existentes	$\|D\| = 10$
Curva aproximada	$f(x) = 32x * log(x)$ $+1500$

La complejidad, vista de esta forma, sería $O(mlog(n))$. La curva graficada corresponde a $O(mlog(m))$, que por propiedades del logaritmo ya mencionadas es equivalente a nivel asintótico.



Datos del gráfico

Ciudades	n = 10

Por otro lado, podemos confirmar que el algorítmo no se ve afectado por la proporción entre las rutas ya existentes y aquellas a construir.

4. Apéndices

4.1. Apéndice I: generación y análisis de datos

Para poder analizar las complejidades de los algoritmos propuestos, se utilizaron las siguientes herramientas para generar mediciones y graficar datos:

- Un generador de grafos aleatorios con `std::random` (parte de la biblioteca estándar de C++11).

  Dependiendo los distintos requerimientos de los diversos problemas, tuvimos que crear distintos generadores de grafos. En particular:

  1. Un generador de grafos completos, utilizado para el problema 3 y como base para otros generadores
  2. Un generador de árboles, usando grafos completos como base y un algoritmo basado en Kruskal para obtener el arbol generador
  3. Un generador de grafos conexos, utilizado en el problema 1, partiendo de un grafo y un arbol generador y agregando ejes adicionales
  4. Un generador de grafos no necesariamente conexos, utilizado para el problema 2.

  Los generadores utilizan generadores de distribuciones uniformes provistas por la `std` de C++.

  Cada uno de los grafos generados se utilizó multiples veces para testear varios aspectos de un mismo problema.

- Mediciones de tiempo con `std::chrono` (parte de la biblioteca estándar de C++11).

  Cada algoritmo fue probado 100 veces consecutivas con cada entrada, conservando solo el valor de tiempo menor para reducir el ruido por procesos ajenos al problema.

  La unidad de medición preferida fue microsegundos (`std::chrono::microseconds`, $seg \times 10^{-6}$).

- Graficado con `matplotlib.pyplot`, `pandas` y `seaborn` (con Python y Jupyter Notebook)

  Se utilizaron los DataFrames de Pandas para el manejo de datos (guardados en `.csv`), y las funciones de regresión de Seaborn para el graficado, en conjunto con matplotlib.

Los gráficos incluyen el coeficiente de correlación (o r") de Pearson, al igual que un p-valor para la hipotesis nula que los valores pueden haber sido generados sin correlación real. Esto quiere decir, un R cercano a 1 con un p-valor mínimo indica que hay una fuerte correlación positiva entre los valores graficados; un p-valor elevado, por otro lado, invalida una posible correlación positiva o negativa.

4.2. Apéndice II: herramientas de compilación y testing

Durante el desarrollo se utilizaron las siguientes herramientas:

- CMake

  Se decidió utilizar CMake para la compilación por su simplicidad y compatibilidad con otras herramientras. Junto con el código se provee el archivo `CMakeLists.txt` para compilar el mismo.

- Google Test

  Para generar tests unitarios con datos reutilizables se usó Google Test. Dichos archivos eran importados por otro `CMakeLists.txt` y no están incluídos en la presente entrega del trabajo práctico.

- Namespace `Utils`

  Dentro de `Utils.h` se definió una función de logging que fue utilizada al programar para detectar errores y ver otros detalles del proceso. La función `log` sigue estando incluída en los algoritmos, pero su funcionalidad se encuentra apagada por un `#define` y no debería generar ningún costo adicional (ya que usa printf por detrás y no genera el output salvo que sea necesario).

5. Informe de cambios

Cambios generales

- Se agregaron gráficos representando los ejemplos provistos para cada problema
- Se realizaron correcciones ortográficas y de redacción, para facilitar la lectura y el entendimiento
- Los gráficos de complejidad fueron rehechos para explicitar más información, en particular:
  - Coeficiente de Correlación de Pearson
  - Funciones de las curva graficadas
  - Constantes utilizadas al analizar una sola variables
- Se volvieron a tomar algunas mediciones que tenían ruido excesivo

Ejercicio I: Delivery óptimo

Se realizó un cambio en el algoritmo de Dijkstra, para mejorar su complejidad. En lugar de utilizar una matriz de adyacencia, pasamos a usar una cola de prioridad implementada con Fibonacci Heap y listas de adyacencia, de forma que la complejidad de este algoritmo sea la más óptima posible.

Dadas estas modificaciones, los análisis de complejidad fueron realizados nuevamente y tuvimos que tomar nuevas mediciones.

Ejercicio II: Subsidiando el transporte

Este algoritmo tuvo que ser refactorizado, ya que contaba con errores y su demostración era difícil e insuficiente. El mismo fue modificado de la siguiente forma:

- Al principio se agrega un nodo universal al grafo; el mismo es utilizado como origen siempre al correr Bellman-Ford
- Ya no se verifica si el grafo es conexo o cuáles son sus componentes conexas; el nodo universal es agregado siempre
- En consecuencia de lo anterior, ya no se utiliza DisjointSet para tomar representantes de las componentes conexas existentes

Esto nos permite respetar la cota de complejidad, simplificar el algoritmo y su demostración de correctitud, y considerar casos particulares que anteriormente fallaban.

Debido a estos cambios, las mediciones se tomaron nuevamente y todas las demostraciones y los análisis fueron reescritos.

Ejercicio III: Reconfiguración de rutas

- Se reorganizó la demostración, buscando dar una idea más intuitiva y aclarar las demostraciones que podían resultar confusas
- Se rehizo la cota temporal, extendiendo la explicación y mejorando su dialéctica para facilitar el entendimiento
- En la experimentación, adicionalmente, se analizó la complejidad en base a la cantidad de aristas del grafo; esto no utiliza mediciones distintas, los valores del eje X fueron transformados teniendo en cuenta que $m = \frac{n*(n-1)}{2}$, y se utiliza una curva acorde