# 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.

un dibujo ayuda.

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.

es mejor evitar esta cuenta y mantenerlo como par.

| agregarEjeComun (**in** $e$ : eje, **in** $n$ : nat, **in** $k$ : nat, **in/out** $grafoConNiveles$ : digrafo) |
|---|
| **for** $c \leq k$ **do**<br>  nuevoPrimerNodo = e.primerNodo + c × n<br>  nuevoSegundoNodo = e.segundoNodo + c × n<br>  agregarEjeADigrafo(nuevoPrimerNodo, nuevoSegundoNodo, e.peso, grafoConNiveles)<br>  agregarEjeADigrafo(nuevoSegundoNodo, nuevoPrimerNodo, e.peso, grafoConNiveles)<br>**end for** |

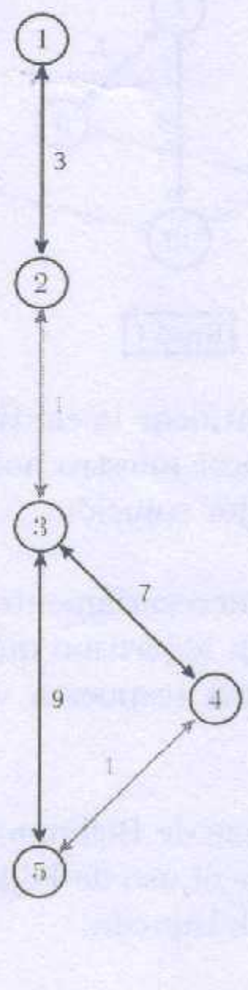
| agregarEjePremium (**in** $e$ : eje, **in** $n$ : nat, **in** $k$ : nat, **in/out** $grafoConNiveles$ : digrafo) |
|---|
| **for** $c < k$ **do**<br>  nuevoPrimerNodo = e.primerNodo + c × n<br>  nuevoSegundoNodo = e.segundoNodo + c × n<br>  agregarEjeADigrafo(nuevoPrimerNodo, (nuevoSegundoNodo + n), e.peso, grafoConNiveles)<br>  agregarEjeADigrafo(nuevoSegundoNodo, (nuevoPrimerNodo + n), e.peso, grafoConNiveles)<br>**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**

Informen la representación del grafo usada. ¿¿ Por qué' eligieron esa ?! (matrizAdy)

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:

- $O(mk)$
- $O((nk)^2)$
- $O(k)$

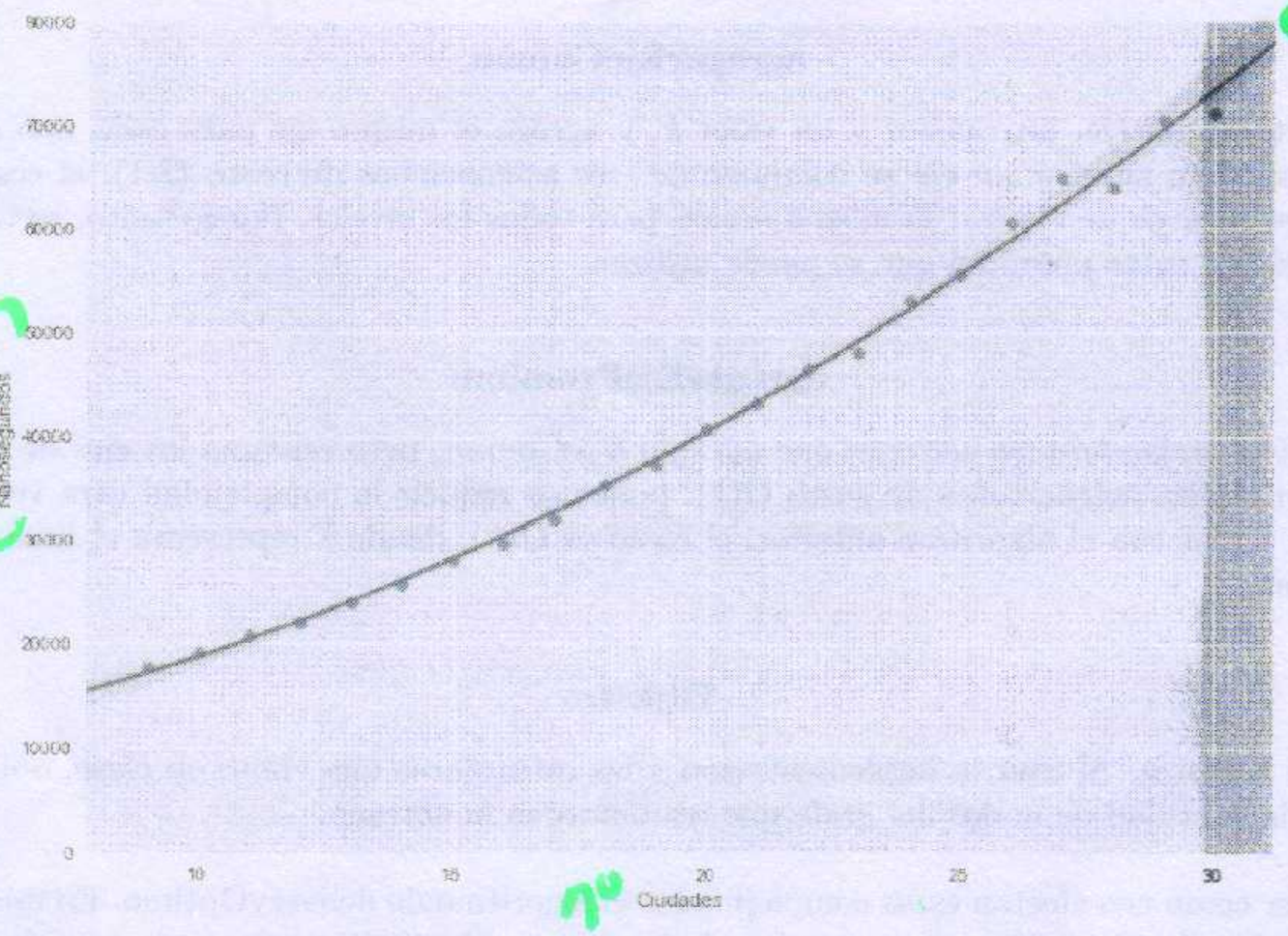
Es trivial notar que $O(k) \subseteq O(mk)$, por lo que $O(k) + O(mk) \subseteq O(k+mk) \subseteq O(m(k+1) \subseteq O(mk)$. Por otra parte, por propiedades de arbol, sabemos que m $\leq n^2$ para cualquier grafo; por lo tanto, podemos acotar $m$ por $n^2$. Entonces, $O(m) \subseteq O(n^2) \rightarrow O(mk) \subseteq O(kn^2) \subseteq O((nk)^2)$. Por lo tanto, $O(mk + (nk)^2) \subseteq O((nk)^2)$.

Por lo visto en el parrafo anterior, tenemos: $O(mk + k + (nk)^2) \subseteq O((nk)^2)$. Por lo tanto, nuestro algoritmo tiene complejidad $O((nk)^2)$ que es igual a $O(n^2k^2)$.

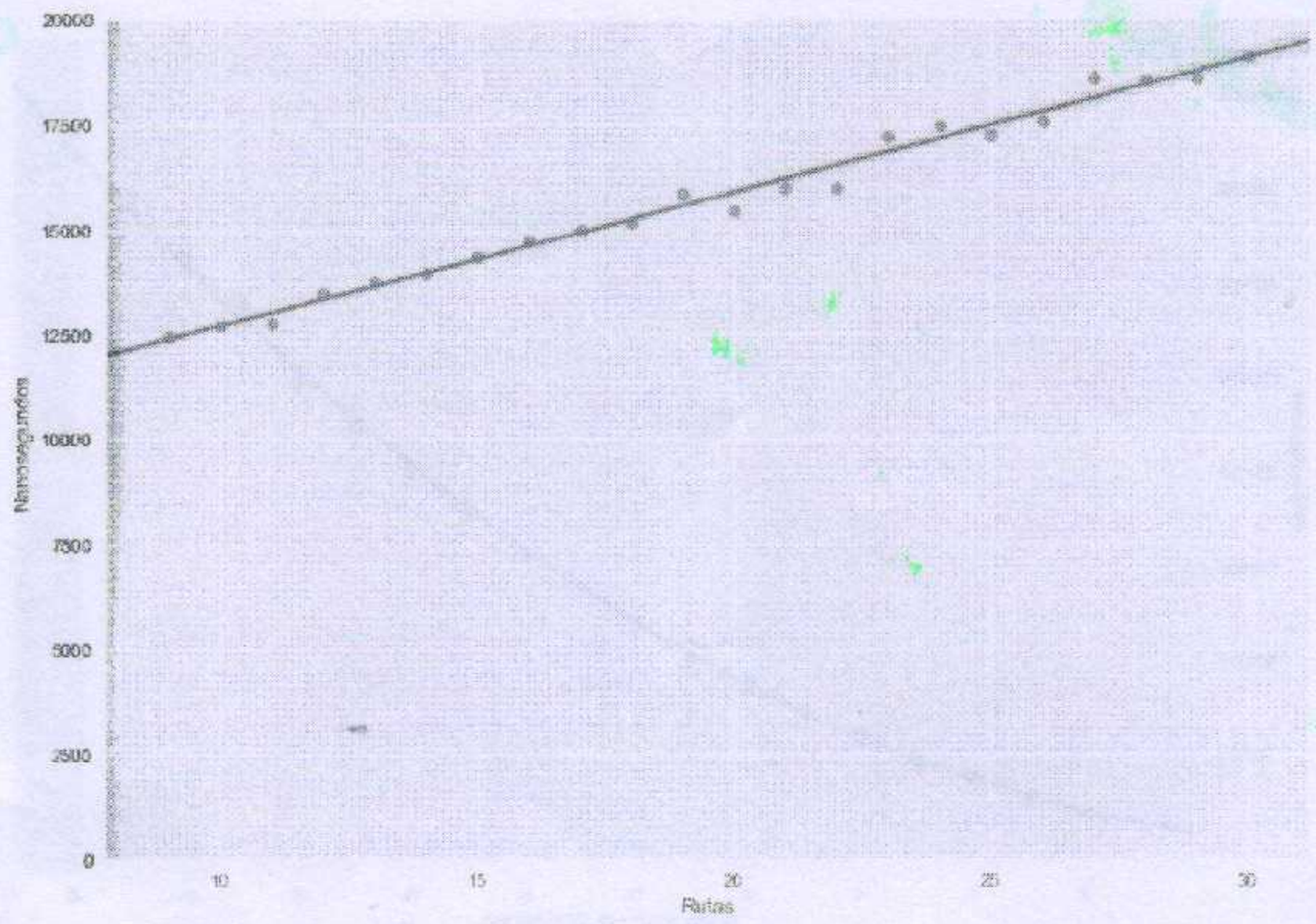
## 1.4. Experimentación

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.

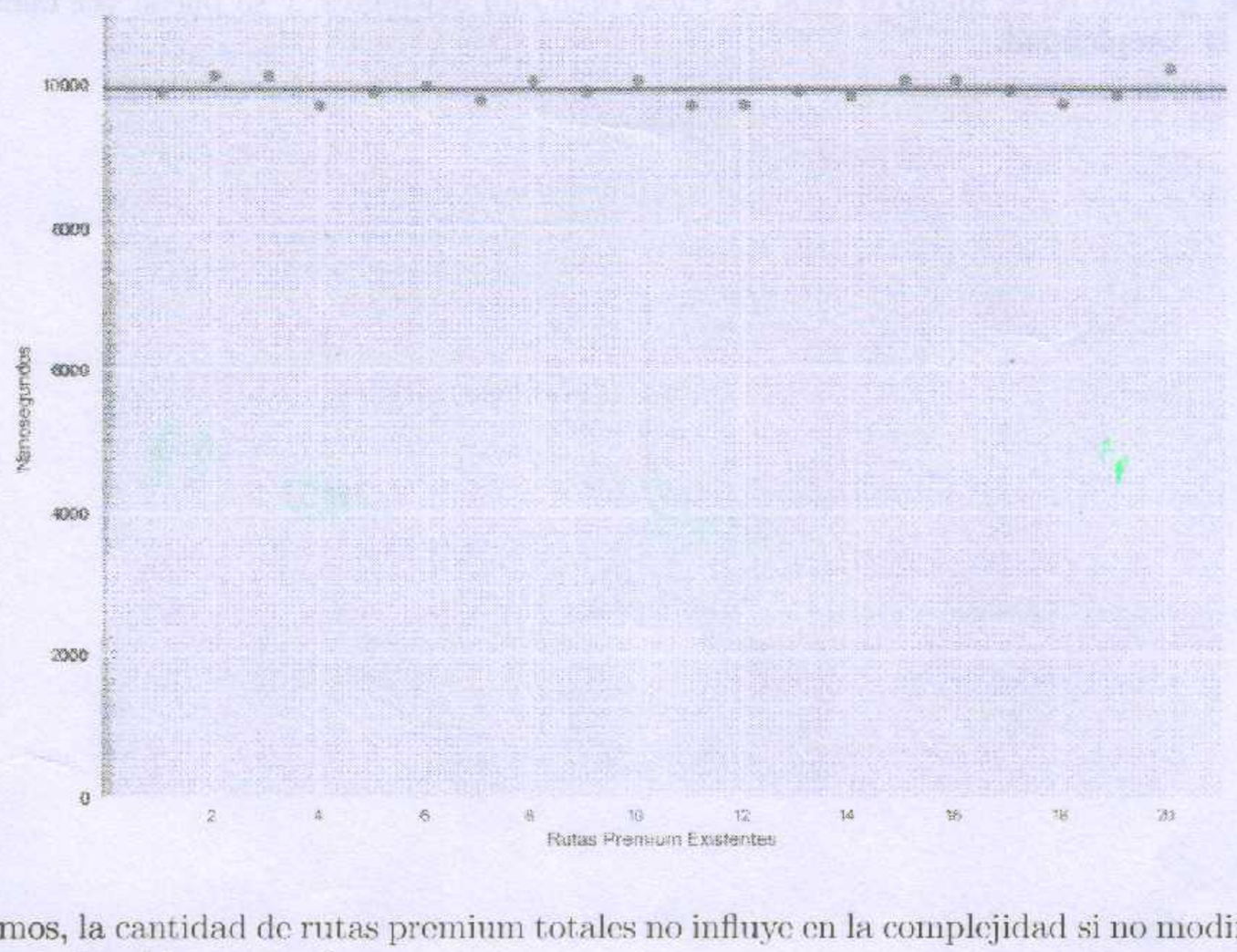
La metodología de prueba y medición de tiempo se encuentra detallada en los apéndices. Los siguientes gráficos corresponden a cada una de las variables aisladas, manteniendo todas las otras fijas en un mismo numero en tanto esto es posible.



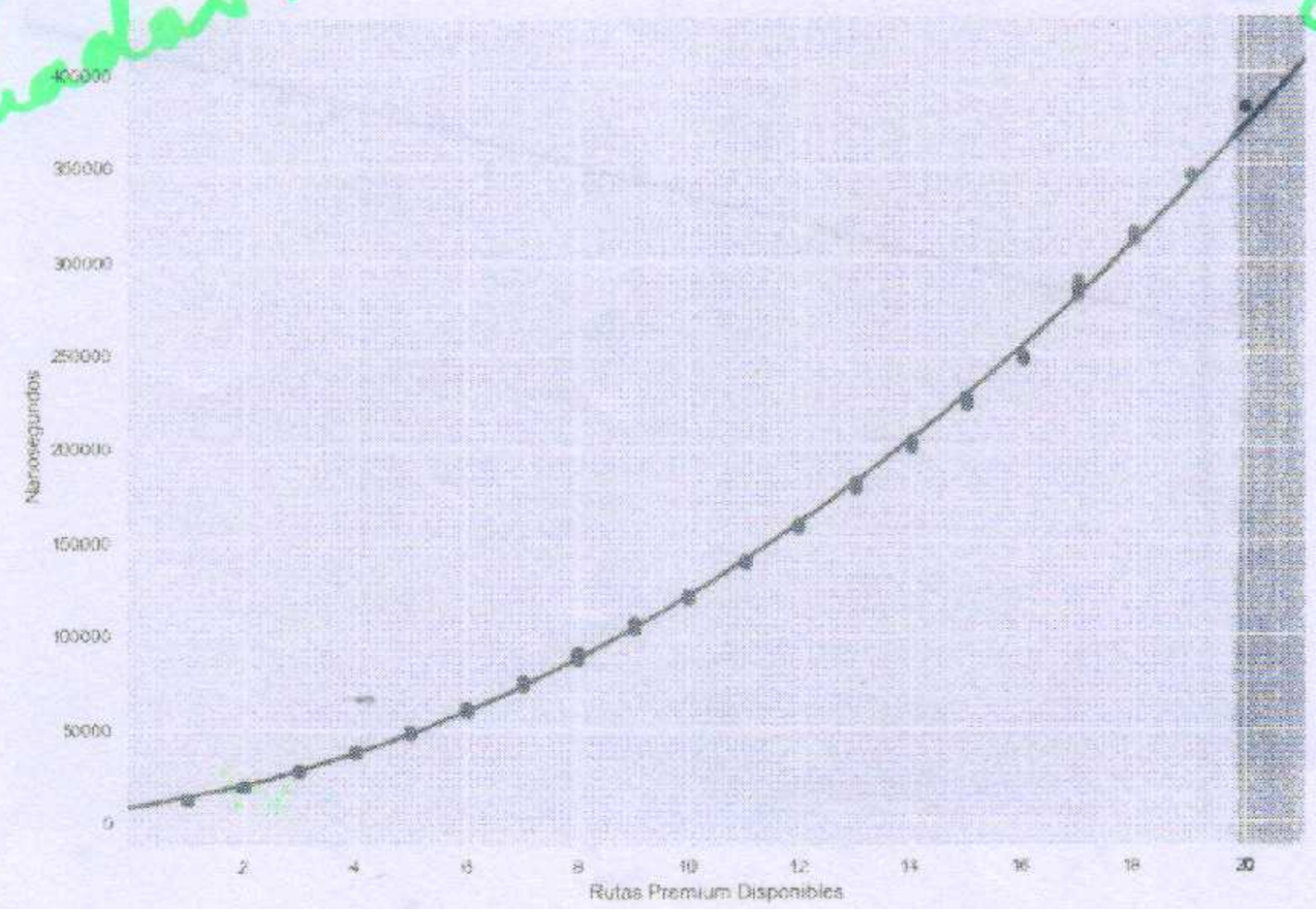

Como se puede apreciar en este gráfico, el tiempo de ejecución depende fuertemente la cantidad de ciudades. Esto no es una sorpresa, ya que como hemos mencionado nuestra cota de complejidad es $O(n^2k^2)$.




**En este gráfico**, podemos notar que si bien la cantidad de rutas no define la complejidad del algoritmo (hay otros aspectos que definen la complejidad), en nuestra implementación tienen un impacto lineal. Esto se condice con el costo $O(mk)$ previamente mencionado.




**Como esperábamos**, la cantidad de rutas premium totales no influye en la complejidad si no modificamos la cantidad total de rutas o los niveles del supergrafo.




Por otro lado, al aumentar la cantidad máxima de rutas premium utilizables, el impacto es nuevamente cuadrático. También se puede ver que la correlación entre tiempo de ejecución y k es más fuerte, ya que los 3 costos mencionados dependen de k.

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.

aplicarle Bellman Ford se encuentran o no ciclos negativos (para esto, suponemos que Bellman-Ford nos devuelve *true* si encuentra ciclos negativos, y *false* si no lo hace)

```
ajusteParaBellmanFord (in p: int, in ejesGrafo: lista[ejes], in origen: int) → res: bool
    lista[ejes] ejesPVersion ← ajustarEjes(ejesGrafo, p)
    res ← bellmanFord(ejesPVersion, origen)
```

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 necesariamente lo tendrán. Por ende, sabemos que nuestra solución está 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 el peso de cada arista de un grafo sin ciclos negativos producirá el aumento del peso del ciclo, y por lo tanto, que se mantenga por arriba de 0. Y 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.

Dado que Bellman-Ford solo puede detectar ciclos negativos en digrafos si son fuertemente conexos, y sabiendo que el grafo de entrada no necesariamente cumple tal hipótesis, será necesario realizar un preprocesamiento al grafo de entrada. Para esto, diremos que un nodo v es huérfano si y solo si $d_{in}(v) = 0$.

Dado que ningún nodo puede llegar a un nodo huérfano, y que en un grafo fuertemente conexo debe existir un camino de ida y de vuelta entre todo par de nodos, si un digrafo contiene nodos huérfanos no es fuertemente conexo. Por el mismo motivo, dado que los nodos huérfanos tampoco pueden pertenecer a un ciclo dirigido, no son relevantes en nuestra bísqueda de ciclos. Entonces, podemos contemplar el grafo donde no hay nodos huérfanos, simplemente aislándolos del resto del digrafo, y eliminando todos sus ejes de salida. Sin embargo, al eliminar los ejes de un nodo huérfano, estaríamos reduciendo el grado de entrada de sus hijos, pudiendo producir nuevos nodos huérfanos, que también deberían ser aislados.

En síntesis, lo que deberíamos hacer es tomar el grafo actual, y quitar todos los nodos huérfanos del mismo, reiteradas veces hasta que finalmente no haya ningún nodo con $d_{in}(v) = 0$. Por lo tanto, para generar un grafo acorde a nuestros propósitos, utilizaremos un algoritmo similar a este:

```
borrarNodosHuerfanos (in n: nat, in lista[ejes]: ejesGrafo)
    lista[nat]: adyacentes ← listaDeAdyacencia(ejesGrafo)
    lista[nat]: gradoEntrada ← gradoDeEntrada(ejesGrafo)
    pila[nat]: nodosHuerfanos ← vacia()
    while j < n do
        if nodosHuerfanos.esVacia() then
            if gradoEntrada[j] = 0 then
                agregarNodosHuerfanos(nodosHuerfanos, adyacentes, gradoEntrada, j, ejesGrafo)
                gradoEntrada[j] ← NULL
            else
                j ← j + 1
            end if
        else
            nat: k ← nodosHuerfanos.dameTope()
            agregarNodosHuerfanos(nodosHuerfanos, adyacentes, gradoEntrada, k, ejesGrafo)
        end if
    end while
```

Cuando logramos aislar a todos los nodos huerfanos del digrafo principal, ya estamos en condiciones de asegurar que para todo nodo v, $d_{in}(v) > 0 \wedge d_{out}(v) > 0$. Sin embargo, a causa de la siguiente propiedad, no es hipótesis

suficiente para que el digrafo sea fuertemente conexo:

Orientar un grafo es darle una dirección a cada eje. Un grafo conexo G es orientable de forma tal que se convierta en un digrafo fuertemente conexo si y sólo si cada eje de G pertenece a un circuito simple de G.

Veamos que los ejes que no pertenecen a un ciclo en el grafo subyacente no pueden pertenecer a un ciclo en el digrafo. Entonces, si el digrafo tiene tales ejes, no es fuertemente conexo. Como esos ejes no pueden pertenecer a un ciclo dirigido, no son importantes en nuestro análisis, y por lo tanto eliminarlos no nos quita soluciones. Por lo tanto, deberemos buscar la manera de quitar estos ejes, para así obtener, finalmente un grafo compuesto por componentes fuertemente conexas.

Para esto, tomaremos nuestro grafo actual (al cual ya le hemos eliminado los nodos huérfanos correspondientes) y consideraremos un bosque generador particular (como bien sabemos, es posible que haya ciclos en el grafo, por lo que es importante marcar que no cambiará el resultado cual es el bosque generador que tomemos). A partir de aquí, tenemos dos tipos de ejes: los que pertenecen al bosque generador, a los cuales llamaremos **inciertos**, y los cuales no pertenecen.

Como tenemos un bosque generador, si agregamos cualquier arista que conecte dos ejes formaremos un ciclo. Por lo tanto, podemos asegurar que todos los ejes que no pertenezcan al bosque generador son parte de un ciclo de nuestro grafo, y por ende conforman una solución parcial. Sin embargo, aún debemos determinar cuales de las aristas inciertas son las que forman ciclos. Para esto, empezaremos por considerar el grafo que conecta a todas las aristas que ya sabemos útiles (las seleccionadas) y considerar a cada una de ellas como parte de una componente conexa. Es decir que si una arista seleccionada incide al nodo 2 y 3, y otra al nodo 3 y 4, necesariamente serán parte de la misma componente conexa.

habría que demostrarlo

el complemento del bosque?

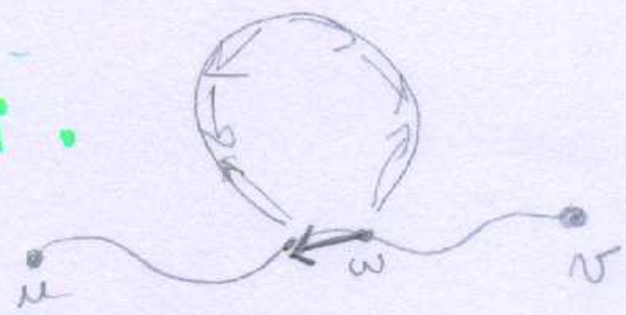
Luego, viendo cada una de las aristas inciertas, deberemos recorrer estas componentes; y al encontrar una arista que incide sobre dos nodos de la misma componente, aseguraremos que esta arista es parte de un ciclo (puesto que ya había un camino entre estos dos nodos, por ser parte de la misma componente, y ahora hay un camino nuevo), y por ende, de las aristas que influyen sobre nuestro problema.

Es importante resaltar que, luego de agregar una arista a las seleccionadas, deberemos repasar las aristas inciertas ya recorridas, pues contaremos con un nuevo conjunto de componentes conexas y, por lo tanto, de ciclos. En términos de pseudocódigo, tendríamos algo así:

```
subsidioDeRutas (in/out inputEdges: lista[ejes])
    bool: changesMade ← true
    lista[ejes]: edgesToUse ← kruskalParaBosque(inputEdges)
    lista[ejes]: inputEdges ← inputEdges - edgesToUse
    while changesMade do
        changesMade ← false
        set: ds ← ∅
        for (u,v) ∈ inputEdges do
            ds.join(u, v)
        end for
        for (u,v) ∈ edgesToUse do
            if ds.connected(u,v) then
                inputEdges.add((u,v))
                edgesToUse.erase((u,v))
                changesMade ← true
            else
                ds.join(u, v)
            end if
        end for
    end while
```

Por lo tanto, luego de borrar los nodos huérfanos y asegurarnos que todas las aristas de la última versión del grafo pertenecen a un ciclo, contamos con las siguientes hipotesis: para todo nodo v del grafo, $d_{in}(v) > 0 \wedge d_{out}(v) > 0 \wedge$ en el grafo subyacente, v pertenece a al menos un ciclo. Esto alcanza para decir que toda componente conexa es fuertemente conexa. Tomemos dos nodos u, v del grafo. Como ambos pertenecen a la misma componente conexa existe un camino entre ellos. Por hipótesis todos los ejes del camino pertenecen a un ciclo. Si estos ejes estuvieran orientados de manera

La demostración no es clara, e intuimos que no funciona. un dibujo ayudaría.




que no se puede llegar de u a v o de v a u, significaría que hay al menos un nodo w en el camino por el que no se puede pasar, en otras palarbas o bien $d_in(w) = 0$ o bien $d_out(w) = 0$. Esto contradice a las hipótesis, absurdo.

Por lo tanto, con lo ya expuesto, podemos utilizar el algoritmo con ajuste de Bellman-Ford que expusimos anteriormente. Sin embargo, como solo podemos asegurarnos que cada una de las componentes conexas de nuestro grafo es fuertemente conexa, deberemos correr este algoritmo para un representante de cada componente. Por otra parte, como sabemos que cada componente, efectivamente, es fuertemente conexa, bastará con que un representante de la componente nos asegure si hay o no ciclos negativos. Por lo tanto, nuestro algoritmo sería algo así:

```
subsidioDeRutas (in n : nat, in ejesGrafo : lista[ejes]) → res: nat
  borrarNodosHuerfanos(n, ejesGrafo)
  borrarEjesQueNoFormanCiclo(ejesGrafo)
  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
  |  res ← cotaInferior
  end while
```

## 2.3. Cota temporal

### borrarNodosHuerfanos

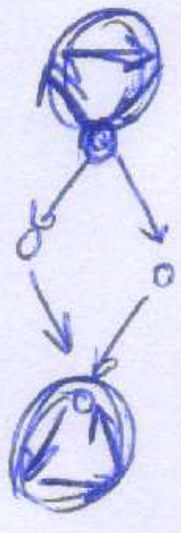
Este algoritmo genera, inicialmente, una lista de adyacencia para cada nodo, un vector de n lugares donde el i-ésimo espacio nos dice el grado de entrada del i-ésimo nodo y una pila vacía. Como nos dan la lista de ejes, al recorrerla podemos generar la lista de adyacencia de cada uno de los ejes y, a su vez, ir contando para cada nodo cuantos nodos tienen ejes apuntando hacia él; y puesto que la entrada a ambos vectores es $O(1)$, y agregar un elemento a una lista también, el costo relevante es el de recorres todas las aristas: $O(m)$, donde m son las aristas del grafo.

Por otra parte, tenemos el ciclo que se encarga de recorrer los $n$ nodos y averiguar si efectivamente es un nodo huerfano o no. Como vemos en el algoritmo expuesto arriba, hay un ciclo que se recorre $n$ veces, lo cual nos da una complejidad de $O(n)$. Sin embargo, no es cierto que el valor $j$ aumenta cada vez que corre el ciclo, sino que depende tambien del hecho de que la pila de nodos huerfanos esté vacía.

Como vimos en el desarrollo, esta pila se encarga de recorrer la lista de adyacencia del nodo huerfano que estamos evaluando, con un costo de $O(n)$, donde n son todos los nodos del grafo (el mayor tamaño posible de la lista de adyacencia). Este proceso se puede producir, como mucho, para los $n$ nodos, en caso de que a través de este algoritmo fueramos descubriendo que cada uno de ellos es huérfano. Por lo tanto, la complejidad de esta porción del algoritmo sería $O(n^2)$.

Pero entonces, al repasar el ciclo, nos encontramos que en el peor caso puede llegar a recorrer dos veces la lista. Suponiendo que, efectivamente, terminaramos apilando los $n$ nodos, tendríamos una complejidad de $O(n^2)$; pero por otro lado, de manera ajena a lo que ocurriera en la pila, el valor de $j$ iría aumentando y, en el peor caso, corriendo n veces la función agregarNodosHuerfanos, dándonos una complejidad de $O(n^2)$. Por ende, aún considerando los dos peores casos en simultáneo (es decir, que los $n$ nodos fueran encontrados recorriendo el vector, y a su vez, agregados a la pila) tendríamos $O(n^2 + n^2) \subseteq O(n^2)$

### borrarEjesQueNoFormanCiclo



Este algoritmo utiliza, para empezar, una versión de Kruskal adaptada para bosques, que mantiene el costo original de Kruskal y, por lo tanto, nos cuesta $O(m.log(n))$, donde m es la cantidad de ejes y n la cantidad de aristas, y lo cual resultará despreciable para la complejidad que tiene el algoritmo.

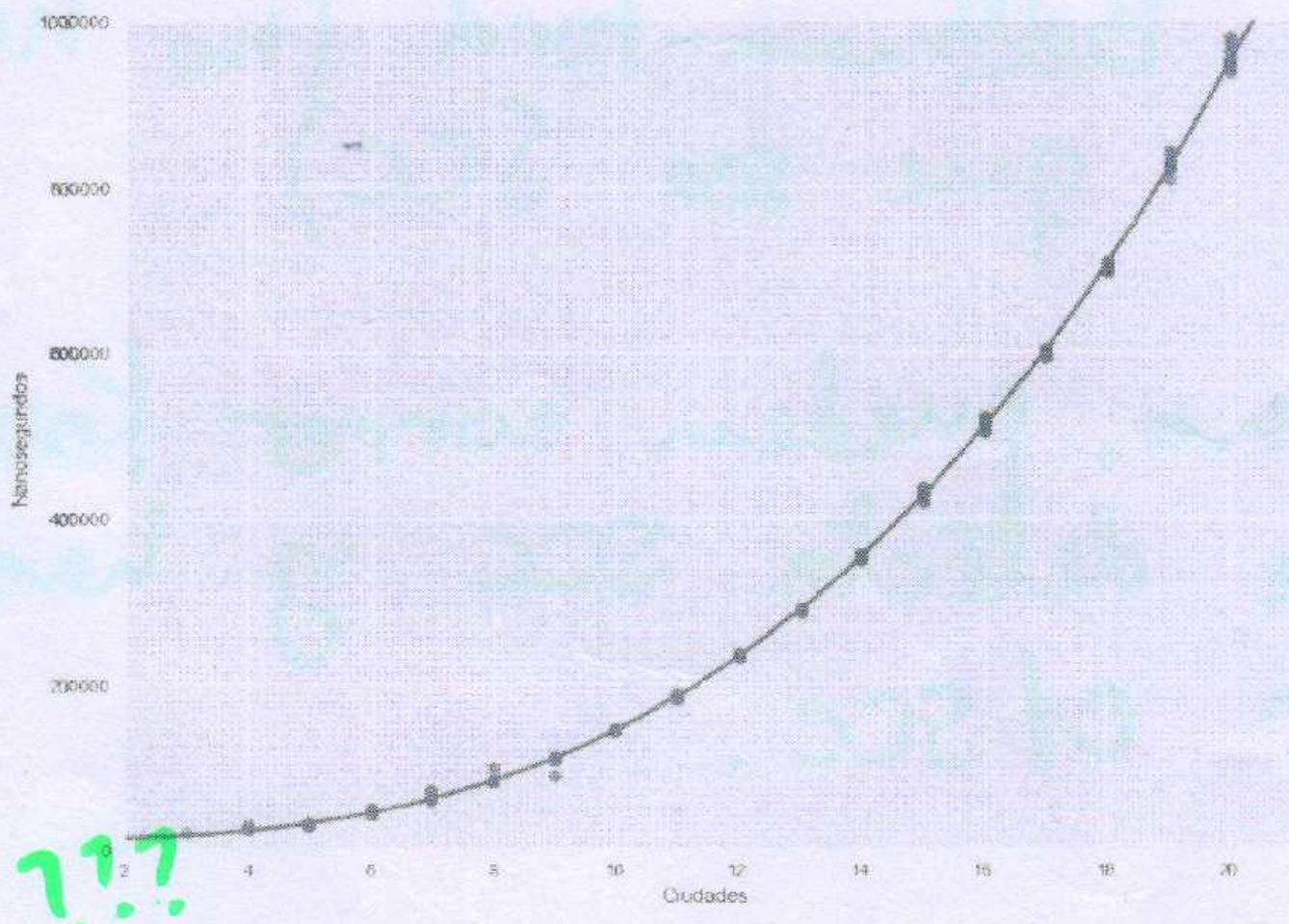
Luego, realiza la iteración de los ejes. Como cada iteración en la que se añaden ejes a los seleccionados conecta al menos a dos componentes de nuestra lista de incidencia, y hay a lo sumo $n$ componentes distintas, el ciclo itera a lo

sumo $n$ veces a todos los ejes, realizando para cada eje operaciones de find y unión, lo que nos da una complejidad de $O(n.m)$. Por otra parte, gracias a las optimizaciones de la estructura disjoint set, podemos despreciar el valor de sus operaciones, considerandolas $O(1)$.

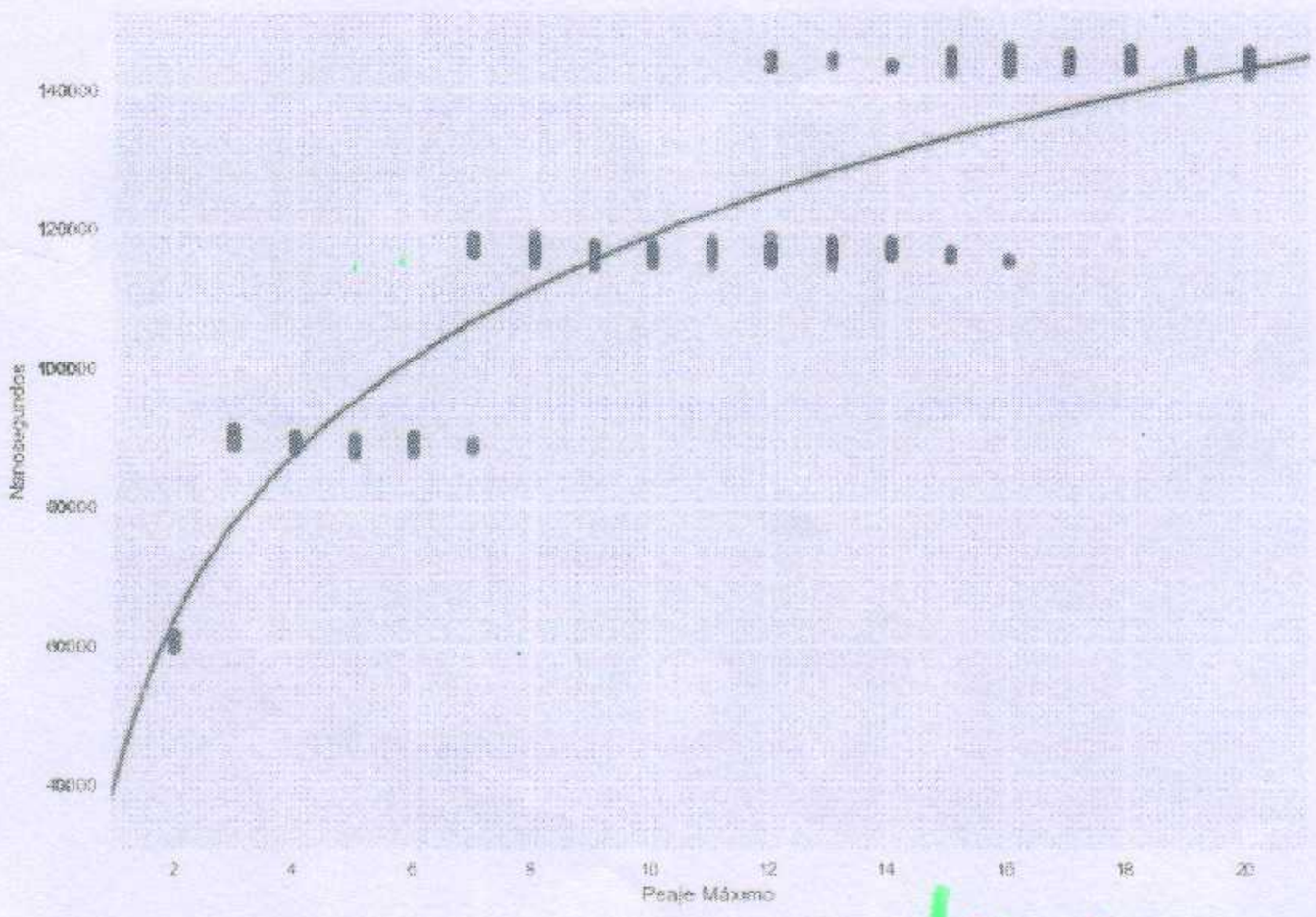
Finalmente, nuestro algoritmo definitivo itera de manera binaria, lo cual nos da un ciclo que se recorre $O(log(c))$ veces para la complejidad $O(n.m)$, dándonos, en definitiva, una cota temporal de $O(n.m.log(c))$

## 2.4. Experimentación

La experimentación de este ejercicio representó un desafío, ya que en la generación de grafos resultó difícil garantizar condiciones que diesen resultados homogéneos debido a nuestra implementación. Esto requeriría crear grafos fuertemente conexos, de modo que ningún eje estuviese fuera de un ciclo. Sin embargo, sí pudimos observar fácilmente los casos de grafos completos.




En este gráfico se puede observar que, en grafos completos, el algorítmo tiene complejidad cúbica con respecto a la cantidad de ciudades. Esto se ajusta a la complejidad pedida, ya que $m = \frac{n \times (n-1)}{2} \approx n^2$, es decir, $O(nm) \subseteq O(n^3)$.



nunca experimentan sobre la parte de hacer SC el grafo.

Por otro lado, podemos ver que el máximo peaje tiene impacto logarítmico sobre el tiempo de ejecución, lo que condice con nuestras expectativas y con la cota de complejidad pedida.

- Si agregan un nodo x tq $(x, v) \in E$ $\forall v \in V$, pueden correr Bellman-Ford (no hace falta que sea SCC).

- Si no, pueden correr Kosaraju que detecta SCC y hacer Bellman en cl SCC.

## 3.2. Desarrollo

Como es usual, podemos representar a las ciudades como nodos de un grafo y a las rutas como los ejes, como no se especifica si las rutas son de una o dos manos vamos a asumir que las rutas son bidireccionales. Pero en este caso tenemos rutas existentes que se pueden destruir y rutas que se pueden crear, vamos a dividir estos ejes en dos conjuntos $C$ y $D$, el primero va a tener los ejes con los costos de construcción de rutas y el segundo va a tener los costos de destrucción, los costos de las rutas se representan como el peso del eje que representa esa ruta. Notemos que $C \cup D$ contiene las aristas de un grafo completo.

Una vez encontrada la solución, vamos a tener rutas que quedaron intactas y rutas que se construyeron, entonces vamos a llamar $RC$ un subconjunto de aristas de $C$ que contiene las rutas que deben construirse para lograr la solución óptima y $RD$ un subconjunto de $D$ que contiene las rutas que hay que mantener, 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, llamaremos solución al conjunto de ejes $R$.

El objetivo del algoritmo es que todas las ciudades se conecten solo de una manera, lo que en grafos sería que existe exactamente un camino simple entre todo par de nodos y cómo sabemos, si pasa esto, el grafo es un árbol. Entonces el queremos llegar a que $R$ sea el conjunto de aristas de un árbol generador del grafo de entrada.

Lo primero que podemos notar, es que, si $RD$ no es un bosque, tiene ciclos. Si tiene ciclos, no hay manera de que agregando aristas se obtenga un árbol. Como ya dijimos que $R$ tiene que ser un árbol, $RD$ tiene que ser un bosque.

Como $R$ tiene que ser árbol, y $RD$ bosque, $RC$ tiene que tener ejes que unen 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 la invariante mantiene un bosque y agrega golosamente aristas que conectan componentes hasta llegar a 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 la invariante y en cada paso conecta a una de las componentes de $RD$ con una arista perteneciente a $C$ y agregando dicha arista a $RC$.

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 $D$", con bosque generador máximo(BGM) de $D$ nos referimos a un subgrafo de $D$ tal que para cada componente conexa $D_i$ de $D$, existe una componente conexa $BGM_i$ perteneciente al BGM de $D$ tal que $BGM_i$ es árbol generador máximo de $D_i$.

Veamos por qué el BGM es el bosque óptimo que tenemos para asignar a $RD$. Tenemos a $D$ con componentes conexas $D_1, D_2, \ldots, D_k$, al sacar un eje cualquiera, digamos que este eje pertenecía a $D_i$, si este eje estaba en un ciclo la componente seguirá teniendo los mismos nodos, pero sino, entonces dividiremos a $D_i$ 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, si le pasamos $RD''$ el $RC'$ resultante tendrá los mismos ejes y uno más, que si le hubiésemos pasado $RD'$ y como estamos diciendo, el costo total de construcción sería más caro y el de destrucción también porque estamos quitando un eje y eso nos cuesta.

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 $k-1$ y que estos $k-1$ están todos incluidos en los de $RD''$. El algoritmo de Kruskal va manteniendo las componentes conexas y evita que se agreguen ejes que unen nodos dentro de una misma componente para que no se formen ciclos, osea que si unimos dos componentes tenemos una nueva componente que contiene a los dos, cuando aplicamos el algoritmo dado $RD''$ en algún momento a tener una componente conexa que contenga a $D_i1$ y otra que contenga a $D_i2$, llamemoslas $A$ y $B$ respectivamente, y el algoritmo elegirá un eje que nos combine las componentes, la intuición está en que los ejes que escogió antes y lo que escoge después son los mismos que en el Kruskal de $RD'$. Notemos que hasta que el algoritmo escoja un eje que une $A$ y $B$, todos estos ejes son también válidos para cuando lo aplicamos en $RD'$, de hecho va a escoger estos ya que son los mejores posibles en ambos casos, el problema es que en $RD'$ existe el eje $e$, el que quitamos en $RD''$ que une $D_i1$ y $D_i2$ osea que vamos ya a tener combinados $A$ y $B$ en una componente sin este costo adicional que tiene al agregar el eje en $RD''$ para unir $A$ y $B$, después el estado de componentes conexas que igual y el algoritmo avanzara igual en ambos casos.

Esto no funciona si algún costo es negativo.

Con esto vimos que si dividimos una componente del bosque nos empeora el costo, también podemos decir que empeora no sólo cuando dividimos una sola componente si no cuando hacemos múltiples divisiones, que es lo mismo que ir haciendo una por una, por lo que por cada división va a empeorar. Gracias a esto podemos apreciar que no tenemos que dividir las componentes conexas, o lo que es lo mismo tener un bosque generador, ahora nos queda ver que el bosque generador máximo es el óptimo. Definimos al 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 el costo de $RD$ es menor que el costo del BGM de $D$ lo cual es absurdo porque el BGM de $D$ es el bosque generador obtenido a partir de la destrucción de las rutas más baratas. Entonces $RD$ tiene que ser el BGM de $D$. Y para conseguirlo vamos a usar Kruskal tambien y vamos a modificarlo para que pare cuando no pueda agregar más ejes, y este nos dará como resultado el BGM.

Resumiendo, los pasos a seguir son:

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

A medida que nos informan las rutas, vamos agregando las rutas 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 prioriza los ejes caros) de sobre los ejes de destrucción, implementamos Kruskal con Disjoint Set, y para reusar el Disjoint Set en la parte de construcción, este recibe por referencia la estructura. Cuando finaliza el Kruskal recorremos los ejes que nos devuelve y nos fijamos los costos, para después restarle al costo total del grafo $D$ (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 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 toda la respuesta.

## 3.3. Cota temporal

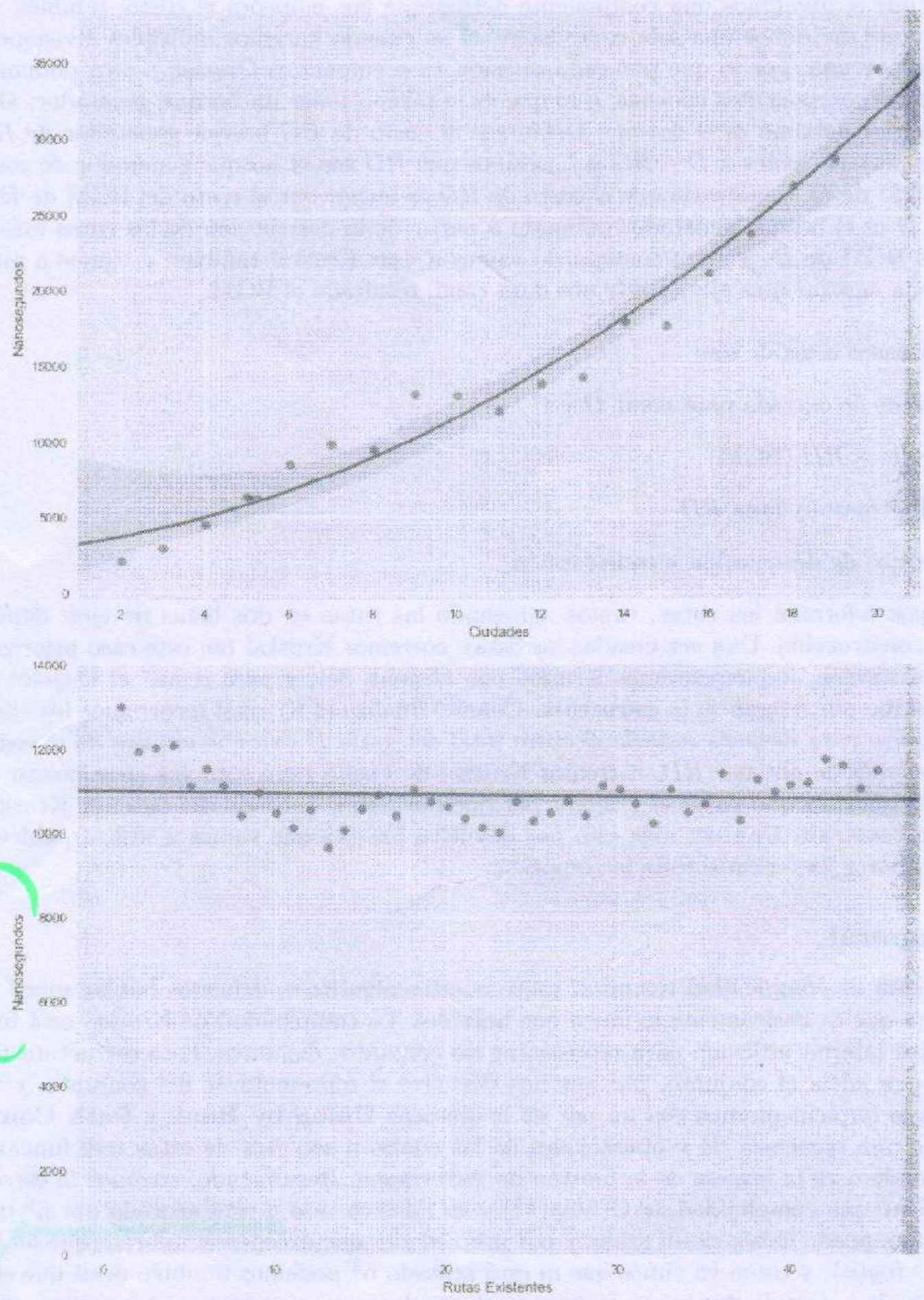
Para tener una cota de complejidad temporal sobre nuestro algoritmo, debemos hablar sobre nuestra implementación de Kruskal, ya que es básicamente lo unico que hacemos. La complejidad de Kruskal está fuertemente atada a la estructura de datos interna utilizada para representar los conjuntos disjuntos. Esta estructura tiene tres funciones en la interfaz, *Init* que inicia el conjunto, *ind* que nos devuelve el representante del conjunto, y *Union* que une los conjuntos. Nosotros lo implementamos con un par de heurísticas **Union by Rank** y **Path Compression**, tal que se puede probar que una secuencia de $k$ operaciones de las cuales n son *Init* de estas tres funciones se ejecutan en tiempo $O(k\alpha(n))$ donde $\alpha$ es la inversa de la función de Ackermann. Resumiendo, creamos la estructura, hacemos el sort de la std que tiene una complejidad de $O(nlog(n))$ y en nuestro caso n está acotado por $n^2$ que son la cantidad de aristas máximas que puede haber en un grafo, y por propiedades que exceden el informe(que fueron vistas en clase) nos queda $O(n + m * log(n))$ y como ya vimos que m está acotado $n^2$ podemos también decir que es $O(n^2log(n))$ .Por último, este algoritmo lo corremos dos veces, y para recolectar los costos recorremos las aristas, así que en peor caso es $O(n^2)$, quedando una complejidad temporal del algoritmo de $O(n^2log(n))$ que se ajusta a lo pedido.

confuso

## 3.4. Experimentacion

Al experimentar con este problema, nos esperábamos 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.

¿Qué casos graficam? ¿Metodología? (detallas más el apéndice)




Efectivamente, la experimentación confirmó que el algorítmo no se ve afectado por la proporción entre las rutas ya existentes y aquellas a construir. Por otro lado, la cantidad de ciudades tiene un impacto aproximadamente cuadrático. Esto es de esperarse, ya que la complejidad efectiva es $O(n^2 log(n))$, pero el aspecto cuadrático es más visible que el logarítmo.

# 4. Apéndices

## 4.1. Apéndice I: generación 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 varias veces 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.

## 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).