DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# TP2 : AGM - Camino Mínimo

**Yéndose por las ramas**

Algoritmos y Estructura de Datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| Destuet, Carolina | 753/12 | carolinadestuet@gmail.com |
| Milicich, Mariana | 534/14 | milicichmariana@gmail.com |
| Murga, Christian Mariano | 982/12 | christianmmurga@gmail.com |
| Perez, Lucía Belén | 865/13 | luu.-18@hotmail.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) 4576-3359
http://www.fcen.uba.ar

**Resumen**

En el presente trabajo, vamos a resolver dos problemas asociados a grafos. El primer problema se basará en árboles generadores mínimos (*minimum spanning tree*) sobre grafos conexos con tres implementaciones distintas(*Prim*, *Kruskal*, *Kruskal con Path Compression*). En el segundo problema analizaremos un caso del problema de camino mínimo (*Shortest path problem*) con pesos en los nodos. Mostraremos una forma de reducir este nuevo problema a uno de camino mínimo simple (sin pesos en los nodos) y analizaremos 5 algoritmos diferentes para encontrar las soluciones óptimas junto con sus complejidades teóricas y experimentales en múltiples casos diferentes.

**Palabras clave:** Grafos conexos, árboles generadores mínimos, camino mínimo, *Shortest path problem*, *Prim*, *Kruskal*, *Dijkstra*, *BellmanFord*, *Floyd*, *Dantzig*.

# Índice

# 1. Ejercicio 1: Hiperconectados

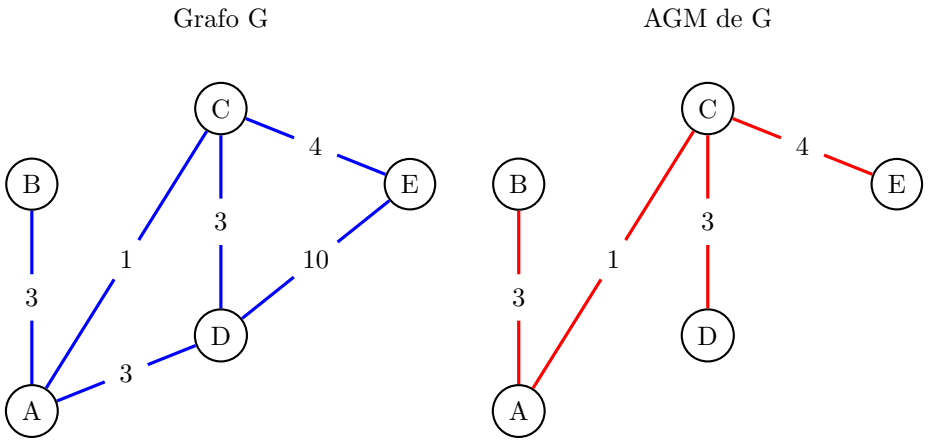
## 1.1. Descripción del problema

Tenemos $n$ ciudades y $m$ posibles conexiones de a pares entre ellas, al mismo tiempo, para cada conexión hay un costo asociado $c$. Dadas estas condiciones, es de nuestro interés armar una red entre todas las ciudades de modo que dadas dos ciudades cualesquiera, haya un camino que las conecte (cuando decimos camino nos referimos a la definición en el contexto de grafos) y, a su vez, de manera tal que esta red $r$ tenga un costo mínimo, es decir, si hay otra red $r'$ que conecte a todas las ciudades, entonces el costo de $r'$ es mayor o igual al de $r$. Tener en cuenta que el costo de una red es exactamente la suma de todas las conexiones de a pares de ciudades que haya en la red. Notemos que puede existir más de una red que cumpla con las condiciones impuestas por el problema, de aquí se desprende la siguiente pregunta que se puede hacer acerca de una conexión entre dos ciudades: ¿dicha conexión está en **alguna** red, en **todas** o en **ninguna**?

El problema descripto se puede traducir a un problema de grafos en los siguientes términos: consideremos $G$ al grafo con pesos cuyos nodos son las $n$ ciudades, cuyas aristas son las $m$ conexiones, y en donde el peso de cada arco es el costo $c$ de la conexión asociada aquel. Luego, el problema original de armar una red se traduce exactamente al de hallar un árbol generador mínimo de $G$ y, dada una conexión $v$ entre dos ciudades, saber en cuántas redes se encuentra esta conexión, se reduce a calcular cuántos árboles generadores mínimos de $G$ contienen a la arista $e$ asociadada a $v$.

Damos un ejemplo a continuación para $n = 5$ con las ciudades $A, B, C, D, E$, cantidad de conexiones $m = 6$ y conexiones $(A, B, c = 3)$, $(A, C, c = 1)$, $(A, D, c = 3)$, $(C, D, c = 3)$, $(C, E, c = 4)$, $(D, E, c = 10)$, donde $G$ es el grafo inducido por el problema y $AGM$ es un posible árbol generador mínimo de $G$, o sea, una red de conexiones entre las ciudades que satisface las condiciones dadas en el enunciado del problema.

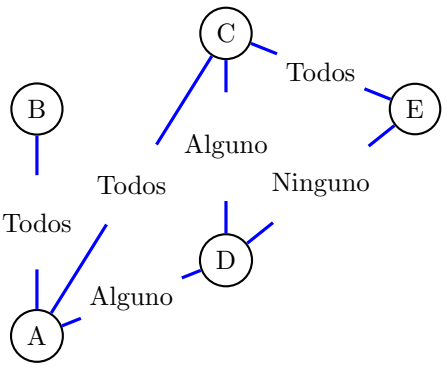
Observemos que la arista $(D, E)$ no está contenida en ningún AGM de $G$ puesto que es posible conectar a los nodos $D$ y $E$ de forma menos costosa a través del nodo $C$. Por otro lado, la arista $(A, B)$ debe estar en todos los árboles generadores mínimos de $G$ dado que $B$ está únicamente conectado con $A$ por lo que no hay otra forma de llegar a él desde otro nodo si se saca esta arista. Por último, como ejemplo de arista que no está en algún $AGM$ es la arista $(A, D)$, si saco esta arista, puedo conectar los nodos $A$ y $B$ mediante el nodo $C$ siempre que esté la arista $(A, C)$, como los dos arcos tienen el mismo peso, son reemplazables uno por el otro.



## 1.2. Desarrollo

La solución del problema al grafo G de la sección previa es la siguiente:

Grafo G con aristas clasificadas




La arista (A,B) pertenece a *todos* los AGM de G pues por definición de AGM es necesario que el conjunto de nodos del AGM coincida con el de G; por su parte, la arista (C,E) pertenece a *todos* los AGM pues la otra conexión del nodo E con el resto de los nodos de G no conforma un AGM, ya que el peso total de un árbol utilizando dicha arista genera un peso mayor que usar la (C,E); es por este motivo que (D,E) no pertenece a *ningún* AGM. Análogamente, (A,C) por poseer un peso tan bajo es una arista que se encuentra en *todo* AGM de G, ya que de no utilizarse, el árbol obtenido superaría el peso total de los AGM de G. Luego, las aristas (A,D) y (C,D) pertenecen a *algún* AGM de G ya que intercambiar una por la otra generan árboles de igual peso.

A continuación desarrollaremos la solución que proponemos para clasificar a todas las aristas de un grafo según su presencia o no en los AGM posibles del grafo.

Dado un grafo conexo no orientado $G$ y una arista $x$ con un peso $p_x$ obtenemos un AGM de G, llamémoslo $A_G$.

En primer lugar, para identificar si x pertenece a ningún AGM o a alguno, decrementamos en una unidad a $p_x$, es decir, peso(x) pasa a ser $p_x - 1$, obteniendo entonces un grafo $G'$. Tras esta modificación, generaremos un AGM para $G'$, llamado $A_{G'}$ y compararemos los pesos de ambos AGM, teniendo las siguientes opciones:

- Si el peso de $A_{G'}$ es igual al de $A_G$, entonces la arista $x$ no pertenece a *ningún* AGM de $G$, ya que de estar incluida en un AGM, el peso de $A_{G'}$ cambiaría en una unidad con respecto al peso de $A_G$.
- Si el peso de $A_{G'}$ es menor que el de $A_G$, podemos decir con certeza que x pertenece a por lo menos un AGM, entrando en la categoría de aristas que pertenecen a *algún* AGM de $G$, ya que lo único que hemos cambiado fue el peso de $x$, y este cambio en el peso total no puede estar dado por utilizar una arista de peso menor por la forma en que los algoritmos que utilizamos generan los AGM, es decir, dicha arista no podría incluirse en $A_{G'}$ sin estar incluida antes en $A_G$.

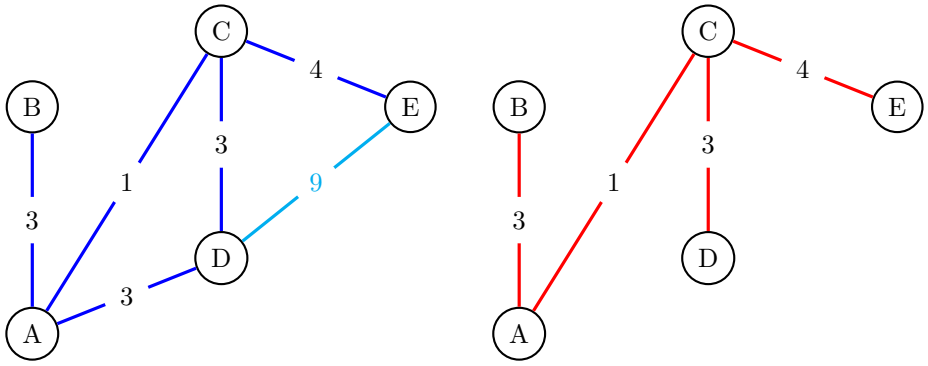
La otra modificación que haremos sobre $p_x$ es incrementarlo en una unidad, pasando a ser peso(x) $= p_x + 1$, obteniendo entonces un grafo $G''$. Tras esta modificación, generaremos un AGM para $G''$, llamado $A_{G''}$ y compararemos los pesos de ambos AGM, teniendo las siguientes opciones:

- Si el peso de $A_{G''}$ es mayor que el de $A_G$, decimos que $x$ pertenece a *todos* los AGM de $G$. El motivo por el que podemos afirmar esto es porque si hubiera una arista de peso menor que pudiese reemplazar a $x$ en $A_{G''}$, indica que o tendría que haber estado en $A_G$ en lugar de $x$ o bien que los pesos coinciden por lo que $x$ y dicha arista estarían dentro del conjunto de aristas que pertenecen a *algún* AGM de $G$.
- Si el peso de $A_{G''}$ es igual al de $A_G$, entonces la arista $x$ no pertenece a todos los AGM de $G$. Con este resultado no podemos decir que pertenece a ninguno o a alguno ya que no nos proporciona la información suficiente: o bien en $A_G$ no se utilizó $x$, o hay una arista que puede ser utilizada en su lugar ya que sus pesos (sin modificar) coinciden.

Utilizando el ejemplo del grafo G del apartado previo y el resultado que expusimos al principio de este, tomamos como ejemplo tres aristas y les aplicamos la solución:

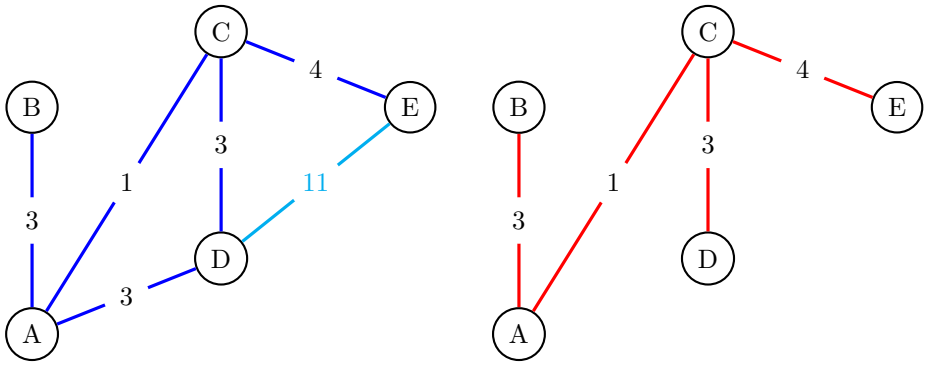
- (D,E): peso = 10.
  - Decrementar su peso: p' = 9. Calculando el AGM del grafo modificado, el peso de este árbol coincide con el del AGM original, ya que, por ejemplo, (C,E) poseen un peso menor y también pertenecerá al AGM del grafo con modificaciones. Entonces, (D,E) clasifica dentro de las aristas que no están en ningún AGM de G.

Grafo G' AGM de G'




  - Incrementar su peso: p' = 11. Generando un AGM con esta modificación se obtiene un peso igual al del AGM del grafo original, por lo que se determina que no pertenece a todos.

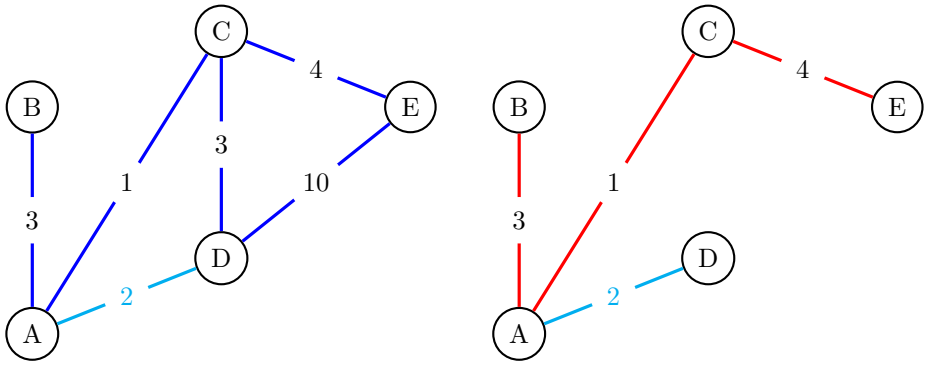
Grafo G" AGM de G"




(D,E) fue correctamente clasificada. De hecho, no se produjeron contradicciones en ambas modificaciones: al decrementar su peso, sabemos que no pertenece a ningún AGM de G, y de incrementarlo, que no pertenece a todos los AGM.

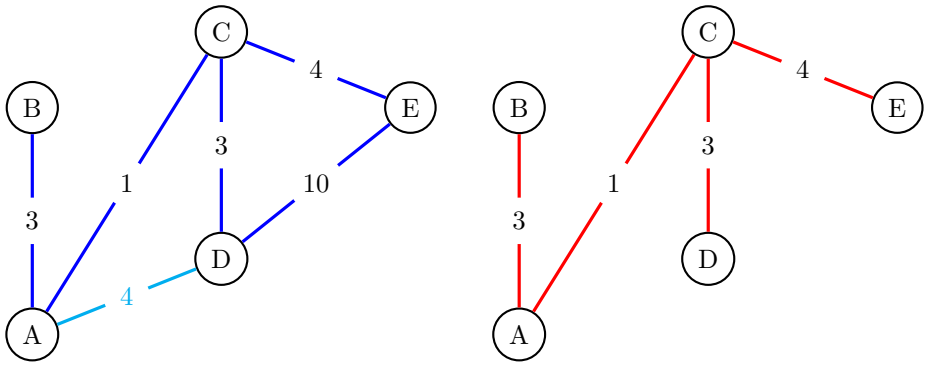
- (A,D): peso = 3.
  - Decrementar su peso: p' = 2. Generando un AGM con esta modificación se obtiene un peso menor que el del AGM de G sin modificar, ya que se utilizará esta arista para construirlo y se determina (A,D) que pertenece a algún AGM del G original.

Grafo G' AGM de G'




- Incrementar su peso: p' = 4. Diremos que (A,D) no pertenece a todos los AGM, porque la arista que de utilizará en su lugar para generar el AGM del grafo modificadó será (C,D) cuyo peso es 3.

Grafo G" AGM de G"

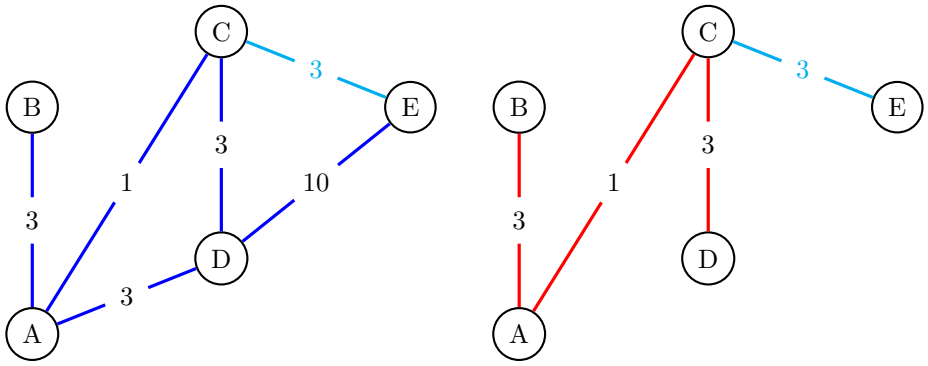



(A,D) fue correctamente clasificada. Nuevamente, podemos decir que no se produjeron contradicciones en ambos casos: como en el primer paso se dijo que (A,D) pertenece a algún AGM y en el segundo que no pertenece a todos, se concluye que pertenece a algún AGM de G.

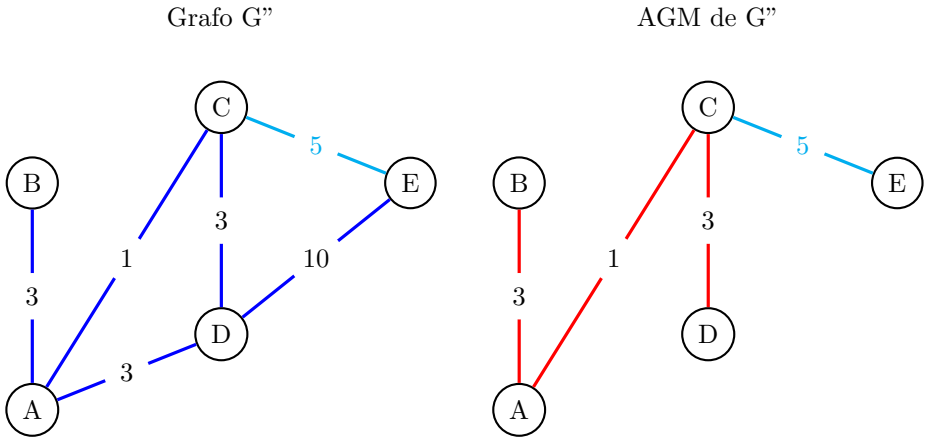
- (C,E): peso = 4.

  - Decrementar su peso: p' = 3. Generando el AGM para el grafo modificado, se obtiene un peso menor que el del AGM de G sin modificar, puesto que (C,E) pertenece al conjunto de aristas del AGM. Con esta observación, podemos decir que (C,E) pertenece a algún AGM.

Grafo G' AGM de G'

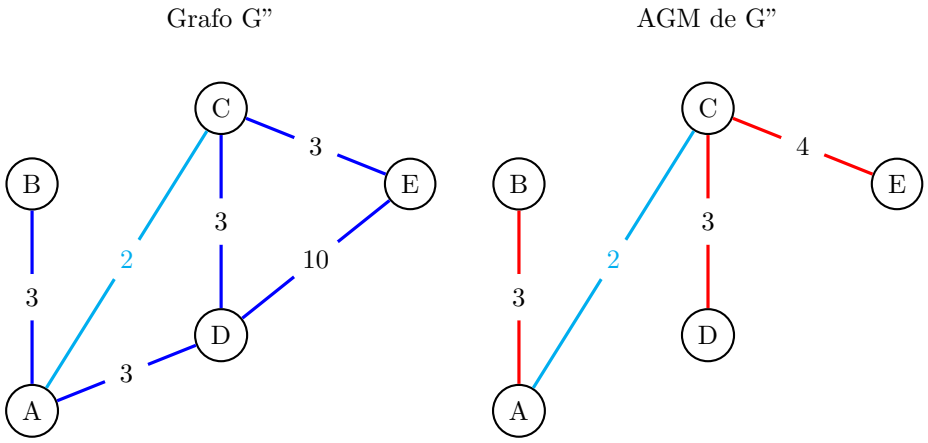



  - Incrementar su peso: p' = 5. El AGM resultante de esta modificación presenta un peso mayor que el de $A_G$, y en base a este resultado se deduce que la arista (C,E) pertenece a todos los AGM de G.




(C,E) fue correctamente clasificada siguiendo la solución que planteamos.

- (A,C): peso = 1. Como ya explicamos anteriormente, una arista de peso 1 seguro está en algún árbol generador mínimo, falta determinar si (A,C) está en todos.
  - Incrementar su peso: p’ = 2. El peso del nuevo AGM es mayor en comparación con el del AGM original, ya que se utilizó nuevamente esta arista para generar este otro árbol. Podemos decir que (A,C) pertenece a todos los AGM.




(A,C) fue correctamente clasificada. Al incrementar el peso se llega a que pertenece a todos los AGM de G.

Concluimos en que nuestra forma de generar los caminos mínimos es correcta al utilizar los algoritmos mencionados, y nuestra forma de decidir sobre la clasificación de las aristas de los grafos conexos que utilizaremos también lo es justamente por utilizar la lógica de construcción para los AGM que tienen los algoritmos y aplicarla a nuestro objetivo.

## 1.3. Pseudocódigo

Como ya mencionamos, vamos a resolver este problema utilizando por separado tres algoritmos: Prim, Kruskal y Kruskal con Path Compression. A continuación describimos brevemente cada uno junto con su invariante y complejidad:

- **Prim** es un algoritmo de tipo goloso (en cada paso construimos la mejor solución que podemos) y crea un árbol generador mínimo agregando de un nodo a la vez. Se puede dividir en tres sencillos pasos:

  1) Para empezar seleccionamos un nodo arbitrariamente del grafo.

  2)Elegimos la arista menos pesada del conjunto de todas las posibles aristas formadas entre vértices que ya incluimos en el árbol y vértices que todavía no fueron incluidos y la incluímos en el árbol junto con el nodo de la arista que no estaba en el árbol.

  3) Repetimos 2) hasta que todos los vértices fueron incluidos al árbol.

  **Invariante** En cada paso del algoritmo iterativo, el conjunto T de nodos y aristas agregadas forman un subgrafo de algún árbol generador mínimo del grafo y si estamos en la iteración iésima del ciclo externo de Prim, dicho subgrafo tendrá exactamente $i$ nodos.

  **Complejidad** $O(n^2)$

- **Kruskal** El algoritmo de Kruskal parte de componentes conexas y las va uniendo de a una en cada paso del algoritmo. Inicialmente, ordenamos las aristas del grafo de menor peso a mayor peso, y cada vértice forma su propia componente conexa. El algoritmo reiteradamente considera la arista menos pesada de las que no fueron añadidas aún y verifica que el par de nodos de dicha arista no estén en la misma componente. De ser así, la arista es descartada dado que añadirla crearía un ciclo en el árbol que se quiere construir. Si, en cambio, los nodos yacen en distintas componentes conexas, añadimos la arista al conjunto de aristas solución y unimos las componentes. Seguiremos considerando aristas hasta tener $n-1$ aristas en la solución final.

  **Invariante** Cualesquiera sean las aristas que tenemos añadidas en el conjunto solución en cada paso iterativo del algoritmo, dichas aristas forman parte de algún árbol generador mínimo.

  **Complejidad** $O(m * n)$

- **Kruskal con Path Compression** La idea de este algoritmo es exactamente igual que la de Kruskal y se mantiene el mismo invariante, la diferencia es que se usa una representación distinta de disjoint-set a la de Kruskal para las componentes conexas que se van uniendo a lo largo del algoritmo. Dicha representación implica una mejora en términos de complejidad.

  **Complejidad** $O(m * log(n))$

**Algoritmo 1:** Ejercicio 1

```
   input  : aristas, n, m
   output: clasificación de aristas en todos, alguno, ninguno
 1 ejes ← aristas                                                    // O(m)
 2 res ← array(m)                                                    // O(m)
 3 agm ← crearAGM(aristas,n,m)
 4 for i ← 0 to m do
 5 |   peso(ejes[i]) ←peso(arista[i]) - 1                           // O(1)
 6 |   agm1 ← crearAGM(ejes,n,m)
 7 |   if peso(agm1) = peso(agm) then
 8 |   |   res[i] ← "ninguno"                                       // O(1)
 9 |   end
10 |   if peso(agm1) < peso(agm) then
11 |   |   res[i] ← "alguno"                                       // O(1)
12 |   end
13 |   peso(ejes[i]) ←peso(arista[i]) + 1                          // O(1)
14 |   agm2 ← crearAGM(ejes,n,m)
15 |   if peso(agm2) > peso(agm) then
16 |   |   res[i] ← "todos"                                        // O(1)
17 |   end
18 |   peso(ejes[i]) ←peso(arista[i])                              // O(1)
19 end
20 return res
```

La función **peso** toma un grafo que es árbol y suma los pesos de sus ejes. Como un árbol tiene n - 1 ejes (siendo n la cantidad de vértices), la complejidad de esta función es $O(n)$.

Por otro lado, como consideramos tres algoritmos distintos para generar árboles generadores mínimos (Prim, Kruskal y Kruskal con Path Compression) entonces la función **crearAGM** va a depender de qué algoritmo elija para generar el árbol.

## 1.4. Complejidad

Como se observa en el pseudocódigo y dado que tenemos un ciclo global en nuestro algoritmo, éste nos queda con una complejidad temporal de:

$$O(m + costo(crearAGM) + m * 2 * (costo(crearAGM) + n))$$

Las complejidades que va a tomar nuestro algoritmo dependiendo de qué función utilice para crear AGM son:

- **Prim:** $O(m * n^2)$
- **Kruskal:** $O(m^2 * n)$
- **Kruskal con Path Compression:** $O(m^2 * log(n))$

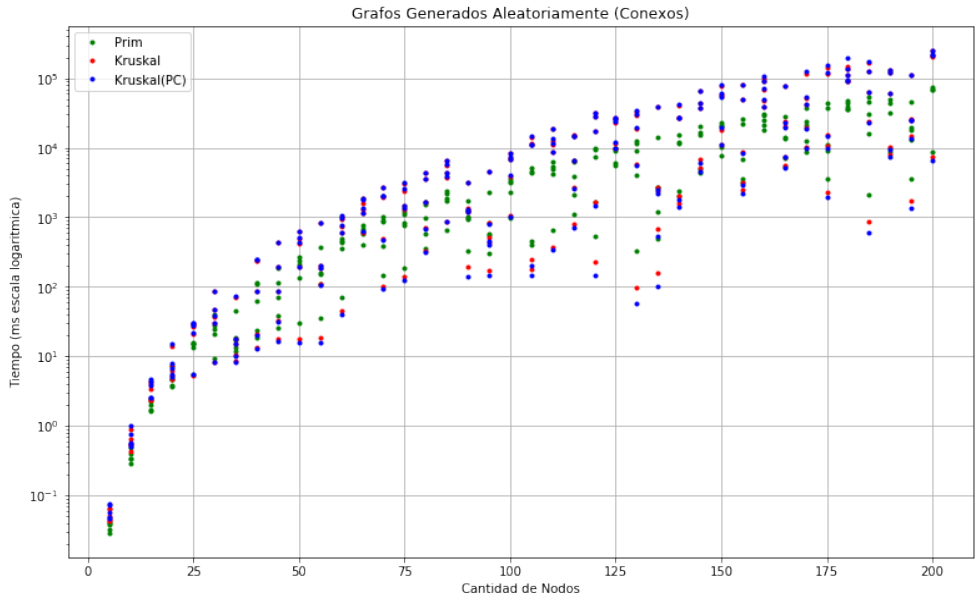
## 1.5. Experimentación

En base a una cantidad importante de test de casos generados aleatoriamente, graficamos los tiempos de ejecución de los diferentes algoritmos de camino mínimo. Mediante un scrypt de python generamos nuestras instancias y las corremos en nuestro código de c++ guardando los resultados en un archivo .csv. Luego graficamos estos resultados usando Jupyter Notebook y Pandas. Todos los experimentos se ejecutaron en las computadoras de los laboratorios del Departamento de Computación.

A continuación se muestran tres experimentos que se desarrollaron para este problema con el objetivo de hacer un análisis comparativo entre los algoritmos propuestos para la solución del ejercicio y ver cómo se comporta cada uno en términos de complejidad temporal de acuerdo a las características del grafo subyacente y cómo es su dependencia en términos de los parámetros de entrada (cantidad de nodos del grafo y cantidad de aristas).
En los tres experimentos se generan instancias de grafos con cantidad de nodos $n \in \{5, ..,100\}$ y peso de aristas $p \in \{1, ..,100\}$ distribuidas aleatoriamente.
Elegimos 100 como máximo peso posible de las aristas por que nos importa que haya varias aristas de pesos iguales en un grafo para generar casos de aristas que se encuentran solo en **alguno** de los AGM de ese grafo.

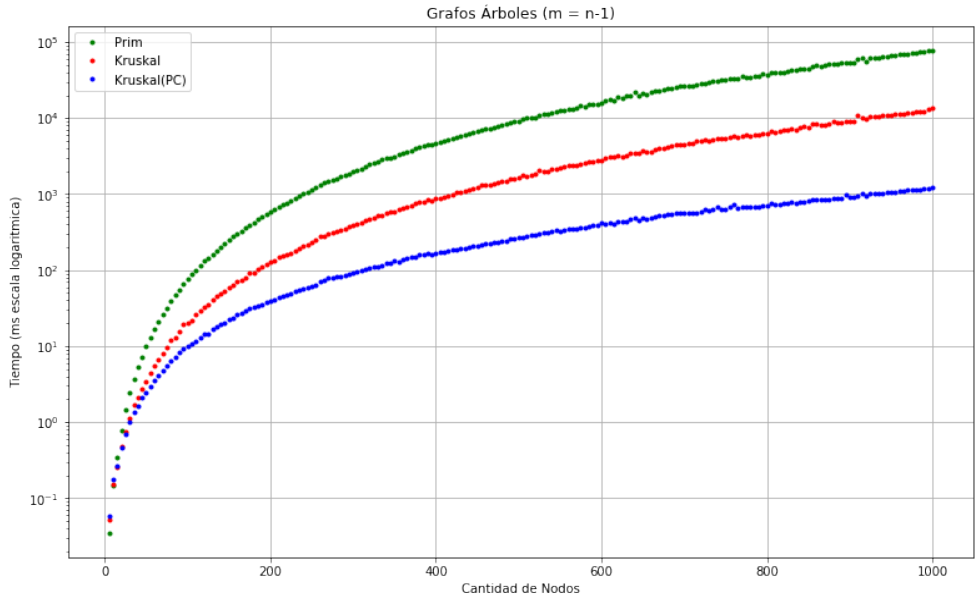
### 1.5.1. Experimento con grafos conexos

El primer experimento y más general de todos que hicimos fue generar instancias de grafos conexos variando desde la mínima cantidad de aristas necesarias para garantizar conexión hasta grafos completos con máxima cantidad de aristas posibles. Dado que, como se puede apreciar en el gráfico inmediatamente posterior, no se ve un patrón muy claro de comportamiento para cada uno de los tres algoritmos, decidimos a partir de estos resultados variar más controladamente los dos parámetros de entrada posibles (cantidad de nodos y de aristas) y explotando la relación que hay entre ellos dos para determinadas configuraciones de grafos.

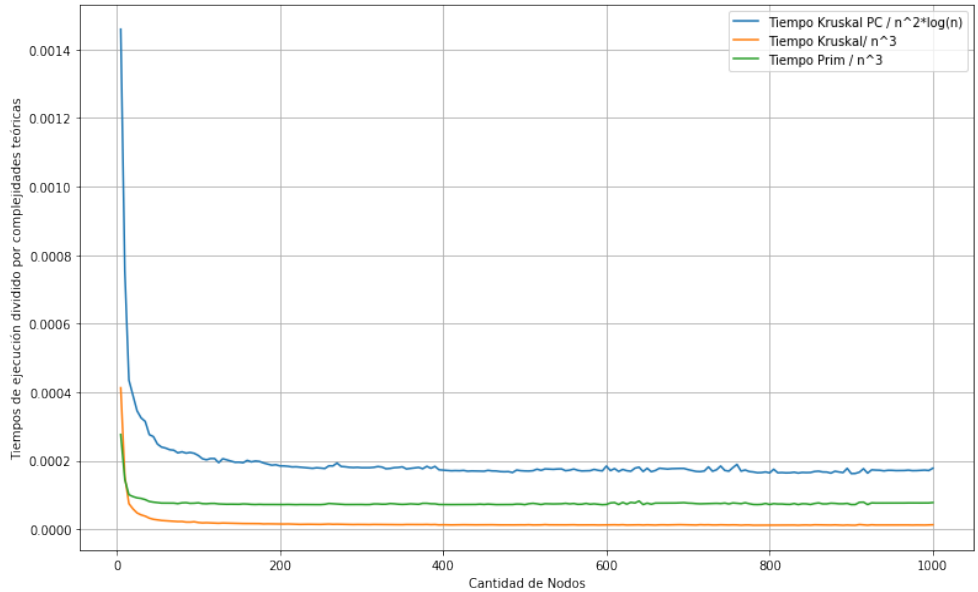



### 1.5.2. Experimento con Árboles

En este experimento decidimos establecer una relación proporcional entre cantidad de aristas y de nodos creando grafos esparsos ($m = n - 1$). Dado que las complejidades temporales teóricas de Prim, Kruskal y Kruskal con PC **para este problema** son $O(m*n^2)$, $O(m^2*n)$ y $O(m^2*log(n))$ respectivamente, esperábamos que los tres tuvieran tiempos de ejecución similares, en especial Kruskal y Prim. Los resultados obtenidos, como se ven en el gráfico expuesto a continuación, son mejores en cuanto a tiempos de ejecución para los dos Kruskal, ganando el de Path Compression peores para Prim. Sin embargo, podemos ver que las tres funciones de complejidad temporal real son extremadamente similares.

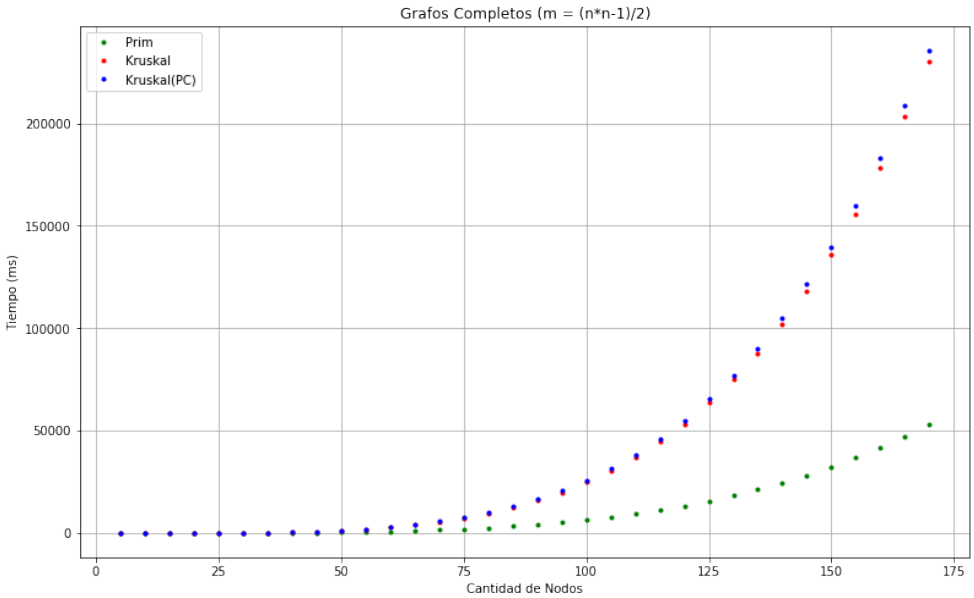



Como dijimos anteriormente, los tiempos de ejecución de los tres algoritmos parecen tener una fuerte relación proporcional entre ellos. Para ver si para cada uno se respetaba su complejidad temporal teórica (que en este caso, como $m = n - 1$, serán $O(n^3)$, $O(n^3)$ y $O(n^2 * log(n))$ para Prim, Kruskal y Kruskal con PC respectivamente), dividimos cada función de tiempo de ejecución de los algoritmos por la función que corresponde a su complejidad teórica. Esperábamos que fuera así y que dicha relación diera una constante, lo cual fielmente se refleja en el gráfico que vemos a continuación.

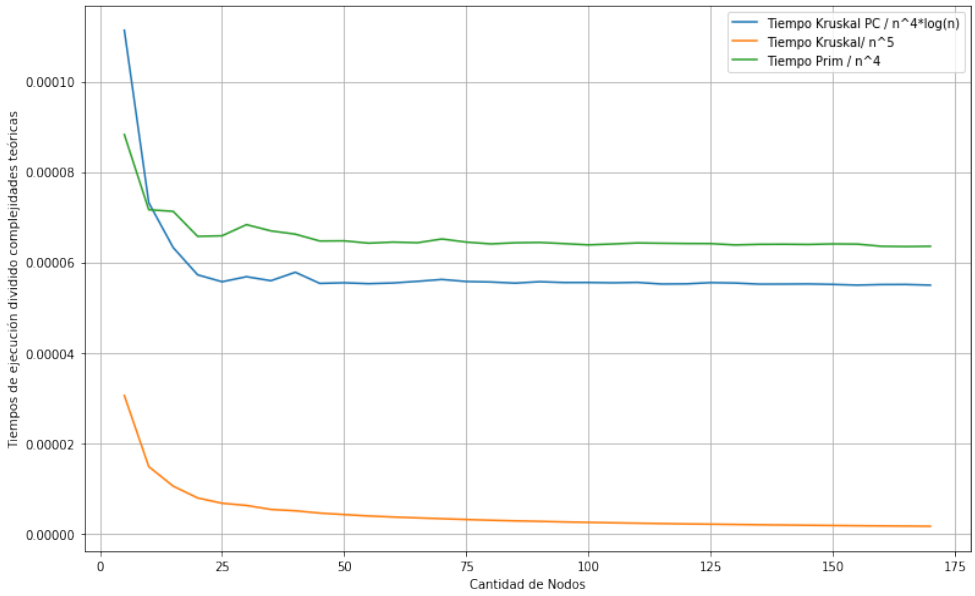



### 1.5.3. Experimento con grafos completos

En este último experimento asociado al ejercicio 1 decidimos generar instancias de grafos completos, donde sabemos que $m = n * (n-1)/2$. Para dichos casos esperábamos obtener tiempos de ejecución mejores para Prim en contraste con los Kruskal dado que en aquel algoritmo depende de forma cuadrática de la cantidad de nodos mientras que en los Kruskal hay un aporte mayor del parámetro $m$ en la complejidad temporal. Efectivamente, como vemos en el gráfico obtenido a partir de los resultados del experimento, los tiempos de ejecución de Prim son mejores que los de ambos Kruskal y entre los dos Kruskal, no podemos notar una mejora del Path Compression versus Kruskal.




Por último, para medir cómo es la relación entre la complejidad temporal teórica y la obtenida en tiempos de ejecución real, se dividió la función del tiempo de ejecución obtenido para cada algoritmo por la función de complejidad teórica. En los tres se puede ver que la relación entre ambos es proporcional dada que la función obtenida de la división se aproxima a una constante.




## 1.6. Conclusiones

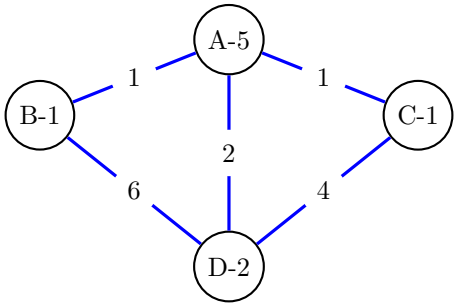
A partir de la experimentación hecha se pudo llegar a la conclusión de que las ventajas en cuanto a complejidad de tiempo de estos tres algoritmos depende directamente de la cantidad de nodos y de aristas del grafo sobre el que se quiere aplicar el algoritmo. Dado que en cuanto a implementación, Kruskal y Prim no son muy diferentes en términos de simplicidad de código, nos atenemos a sus complejidades temporales para concluir que dado que Prim tiene complejidad $O(n^2)$, Kruskal $O(m*n)$ y Kruskal con Path Compression $O(m*log(n)))$, para grafos conexos con pocas aristas o con una cantidad proporcional a la cantidad de nodos, Kruskal puede arrojar tiempos iguales o incluso mejores que Prim. Ahora bien, para grafos muy densos, puesto que Prim depende exclusivamente de la cantidad de nodos, que siempre será menor al número de aristas, Prim arrojará resultados mucho más óptimos que los algoritmos de Kruskal.

# 2. Ejercicio 2: Hiperauditados

## 2.1. Descripción del Problema

El problema plantea una red de $n$ ciudades donde hay $m$ conexiones de a pares de ellas a través de rutas bidireccionales. Para dos ciudades cualesquiera de esta red se puede llegar de una a la otra posiblemente atravesando otras ciudades de por medio. Dada una ciudad $n_i$ $(i = 1, ..., n)$, hay un costo $c_i$ de nafta por litro asociada a ella; a su vez, dadas dos ciudades $A$ y $B$ para las cuales existe una ruta $m_i$ entre ellas, $l_i$ es la cantidad de litros de nafta que se necesitan para recorrer toda la ruta. Si se dispone de un vehículo con una capacidad de tanque de 60 litros, es de nuestro interés averiguar cuál es el camino que el vehículo debe tomar para desplazarse entre dos ciudades dadas de forma de gastar lo mínimo posible en costo y asumiendo que el vehículo tiene el tanque en cero al estar en la ciudad de partida.

Para entender mejor el problema, veamos un ejemplo:




En este ejemplo tenemos 4 ciudades, A,B,C,D, y 5 rutas entre ellas, el costo del litro de nafta en cada ciudad está expresado en dólares. Supongamos que queremos movilizarnos de la ciudad A a la ciudad D gastando lo mínimo posible en nafta. Notar que hay tres posibilidades para desplazarnos de A hacia D:

- Tomamos la ruta directa de A a D, dado que se gastan dos litros de nafta para atravesarla y que partimos con un tanque vacío, el costo de dicho camino es de 10 dólares.
- Podemos pasar por la ciudad B de por medio haciendo el recorrido A a B, B a D. Como el litro de nafta es más caro en A que en B, nos conviene cargar lo mínimo indispensable para llegar de A a B y, luego, cargar en B lo que haga falta para llegar a D. Tenemos entonces un costo total de $5 * 1 + 1 * 6 = 11$ dólares si seleccionamos dicho camino.
- Primero nos desplazamos de A a C, y luego de C a D. Como la nafta es más cara en A que en C, cargamos lo mínimo que haga falta para poder llegar a C desde A para luego cargar en C los litros de nafta que se necesiten para atravesar la ruta que une C con D. Por lo tanto, gastamos un total de $5 * 1 + 1 * 4 = 9$ dólares si elegimos el camino descripto para movernos de A a D.

Como se puede ver a partir de analizar todas las posibles formas de llegar de A a D, no necesariamente el camino más directo, que es la ruta que une a las dos ciudades, es el que conviene tomar si el objetivo principal es minimizar el costo de la nafta. En este caso particular, al haber una ciudad intermedia, C, en donde la nafta por litro es mucho más barata que A, conviene "desviarse" y pasar por esta ciudad para cargar nafta antes de llegar a la ciudad destino. Si se vuelven a desplegar las tres posibilidades pero ahora para ir de D a A, se puede llegar a ver que en dicho caso sí conviene tomar la ruta directa de D a A. Hicimos tal observación para hacer notar que no es simétrico el problema, dicha antisimetría surge del hecho de que el costo de nafta depende no solamente del peso de las aristas entre ciudades si no también del costo de nafta por litro en cada ciudad.

## 2.2. Desarrollo

Como podemos ver en el problema original, $G$ tiene ciertos valores en los vértices y se quiere minimizar cada camino entre pares de ciudades dependiendo de dichos valores. Sería ideal transformar nuestro grafo original a un nuevo grafo $G'$ en donde la información de los nodos se traduzca a pesos de aristas dado que, de ese modo, podemos aplicar algoritmos de camino mínimo a $G'$ y obtener así, nuestra solución al problema aplicado al grafo $G$.

Con esto en mente vamos a definir $G'$ grafo dirigido de la siguiente manera:

- por cada nodo-ciudad del grafo original, vamos a tener $(c+1)$ nodos en $G'$, cada uno de ellos representando un nivel de litros de nafta posible en el rango [0,c] que soporta el tanque. En nuestra ilustración de $G'$, estos nodos asociados a cada ciudad son los que figuran en una misma columna.

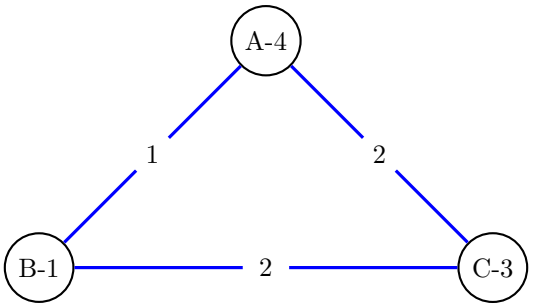
- en $G'$ hay dos tipos de aristas: por un lado (1) aristas que representan cargar nafta en una misma ciudad, y por otro, (2) las que representan desplazarse de una ciudad a otra en el problema original. Tener en cuenta que el peso de las aristas ahora va a ser la plata que cuesta ir de un nodo a otro.

  (1) Las aristas que simbolizan cargar nafta, y que en $G'$ unen dos nodos que significan la misma ciudad, tendrán como peso el costo de la nafta en dicha ciudad por la cantidad de litros cargados. En el gráfico de $G'$, dichas aristas son las verticales en color azul.

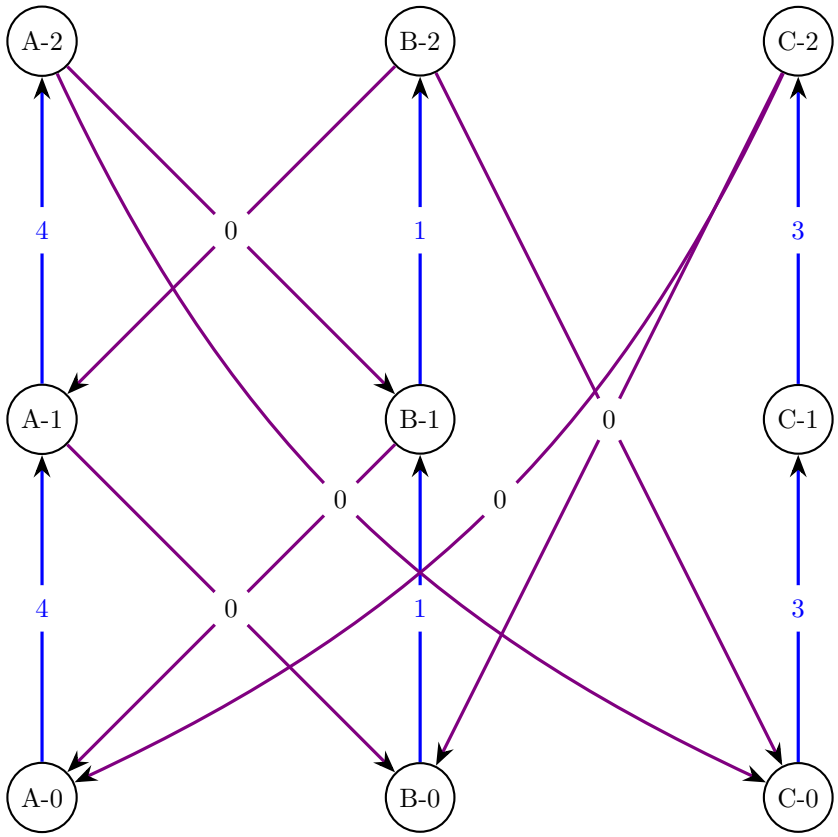
  (2) Una arista e en G que representa consumo de nafta entre ciudades a y b, se va a traducir a $c - peso(e)$ aristas en G', todas con peso 0. Solo va a existir una arista desde (a, n) hacia (b,n') si n-n' = $peso(e)$. En el gráfico de $G'$, tales aristas son las diagonales en color violeta.

  Observar que en caso en que el peso de la arista entre a y b en $G$ sea mayor a la capacidad de litros del tanque, esta arista no será considerada en $G'$.

Por cuestiones de simplicidad y sin pérdida de generalidad, vamos a considerar un ejemplo en donde la capacidad del tanque de nafta es de 2 litros. A continuación se expone el siguiente grafo $G$:




Por lo explicado anteriormente, $G'$ nos queda de la siguiente manera:

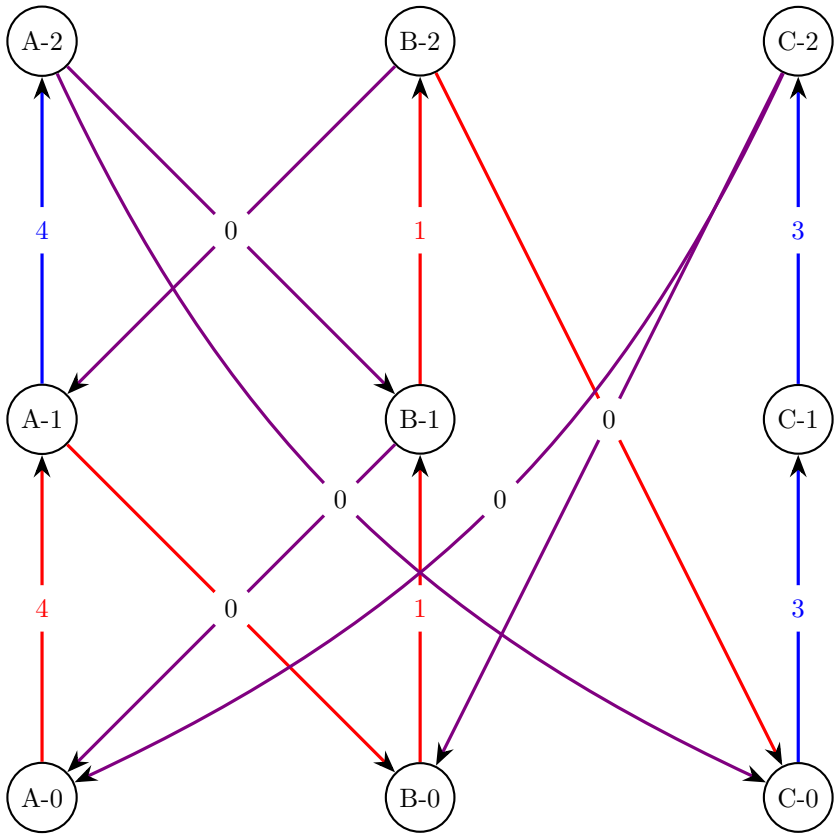



Veamos cómo se traduce calcular el costo mínimo entre las ciudades A y C en la transformación de nuestro grafo G al grafo G':

En el problema original se asume que el vehículo está inicialmente en la ciudad de partida con el tanque de nafta vacío, entonces en $G'$, el nodo desde donde se quiere calcular la distancia mínima es $(A-0)$. En $G'$ tenemos tres nodos asociados a la ciudad $C$, $(C-0)$, $(C-1)$ y $(C-2)$, ahora bien, si llegué a $C$ con el tanque no vacío, significa que en alguna ciudad cargue litros extra de lo que necesitaba para atravesar el recorrido que me queda hasta llegar a $C$, como el objetivo es gastar lo mínimo indispensable, el nodo al que quiero llegar en $G'$ es $(C-0)$, es decir, llego con el tanque vacío a $C$.

En $G$ hay dos posibilidades para llegar de $A$ a $C$, ir directamente de $A$ a $C$ con un costo de 8, o bien ir de $A$ a $B$ y luego de $B$ a $C$. Pasar por $B$ me suma 1 litro más de nafta de consumo al recorrido anterior, pero en proporción, como la nafta en $B$ sale un cuarto de lo que sale en $A$, conviene cargar lo mínimo indispensable en $A$ para llegar a $B$ (1 litro) y luego cargar en $B$ lo que falta para llegar a $C$ (2 litros), el costo total en este nuevo recorrido es 6 en contraste con los 8 de ir directo de $A$ a $C$. En $G'$ el recorrido menos costoso descripto anteriormente se traduce a lo siguiente:

Partimos del nodo $(A-0)$, cargamos en $A$ un litro, que en $G'$ es desplazarse hacia el nodo $(A-1)$ con un costo igual a 4. Luego, vamos hacia $B$ consumiendo lo mínimo de nafta que necesitamos para llegar, o sea que en $G'$ llegamos a $(B-0)$ con un costo 0 (se llega a la ciudad B con el tanque vacío). Para desplazarnos de $B$ a $C$ dijimos que necesitamos cargar 2 litros de nafta en $B$, lo cual en $G'$ se traduce a moverse hacia el nodo $(B-1)$ y luego al $(B-2)$ con un costo total de 2. Finalmente, nos movemos del nodo $(B-2)$ al $(C-0)$ con costo 0. El costo total de este camino es 6, el cual es el valor mínimo de costo para desplazarse desde $A$ hacia $C$ en $G$.




Veamos ahora que nuestro algoritmo resuelve el problema. Sean A y B dos ciudades del problema, es decir A y B dos nodos de nuestro grafo de entrada $G$, quiero ver que el costo mínimo de ir desde A hasta B en $G$ (llamémoslo $C_G$) es el mismo que el costo del camino mínimo desde (A-0) hasta (B-0) en $G'$ (llamémoslo $C_{G'}$).

Queremos ver que $C_G = C_{G'}$.

Supongamos que esto no pasa, entonces va a pasar que $C_G < C_{G'}$ o $C_G > C_{G'}$.

Supongamos que $C_G < C_{G'}$. Sean $P$ y $P'$ caminos de costo mínimo en $G$ y $G'$ respectivamente que unen A con B, es decir, $P$ tiene costo $C_G$ y $P'$ tiene costo $C_{G'}$. Ahora bien, si $P$ es de la pinta:

$$(c_1, n_1, gasto_1), ..., (c_k, n_k, gasto_k)$$

donde $c_i$ es un nodo de $G$ (es decir una ciudad), $n_i$ es el precio de la nafta en $c_i$ y $gasto_i$ es la cantidad de nafta que cargó en el nodo $c_i$.

Notar que $c_1 =$ A, $c_k =$ B y $gasto_k = 0$ pues el camino $P$ parte de A y llega hasta B y en B no carga nafta por ende la cantidad de nafta cargada en B va a ser cero.

Consideremos ahora en $G'$ el siguiente camino $P''$ que une (A,0) con (B,0):

$$(A,0) \to (A, gasto_1) \to (c_2, gasto_1 - l_G(A, c_2)) \to (c_2, gasto_2) \to ... \to (c_{k-1}, gasto_{k-1}) \to (B,0)$$

$l_G(c_i, c_j)$ es el peso de la arista que va desde $c_i$ hasta $c_j$ en $G$, dichas aristas existen porque las recorre el camino $P$ en $G$.

Veamos ahora cuál es el costo del camino $P''$ en $G'$, llamémoslo $C_{P''}$:

$$C_{P''} = l_{G'}((A,0),(A,gasto_1)) + l_{G'}((A,gasto_1),(c_2,gasto_1 - l_G(A,c_2))) +$$
$$l_{G'}((c_2,gasto_1 - l_G(A,c_2)),(c_2,gasto_2)) + ... + l_{G'}((c_{k-1},gasto_{k-1}),(B,0))$$

donde $l_{G'}(x,y)$ es el peso de la arista que va desde $x$ hasta $y$ en $G'$.

Ahora bien, considerando que en $G'$ las aristas que tienen peso mayor estricto que cero son las de la pinta $((c_i, n_0), (c_i, n_1))$, o sea las que se mueven en una misma ciudad y sólo cargan nafta allí, y el resto de las aristas de $G'$ tiene peso igual a cero, llegamos a que:

$$C_{P''} = gasto_1 + gasto_2 + ... + gasto_{k-1} = \sum_{i=1}^{k-1} gasto_i = C_G$$

La última igualdad es por cómo definimos al camino $P''$. Con esto obtenemos que $C_{P''} = C_G$.

Luego, $P''$ es un camino que une (A,0) con (B,0) en $G'$ de costo menor que el camino $P'$, pues por lo visto antes y por hipótesis $C_{P''} = C_G < C_{G'}$. Esto es **ABSURDO** pues encontramos un camino en $G'$ desde (A,0) hasta (B,0) que tiene costo menor que el camino de costo mínimo.

Análogamente llegamos a un absurdo si suponemos que $C_G > C_{G'}$.

Por lo tanto pasa que $C_G = C_{G'}$.

## 2.3. Pseudocódigo

Dividimos esta sección según la forma de calcular el camino mínimo del algoritmo.

### 2.3.1. Uno a Muchos

Los algoritmos subyacentes utilizados en la resolución del problema *HIPERAUDITADOS* que calculan camino mínimo de un vértice a todos son tres: Bellman-Ford, Dijkstra y Dijkstra con priority queue. A continuación explicamos brevemente la idea de cada algoritmo junto con su invariante y su complejidad.

- **Bellman-Ford**

  La lógica del algoritmo de *Bellman-Ford* es la siguiente:

  Sea $s$ el nodo fuente como en Dijkstra. Inicializamos en un arreglo $D$ de tamaño $n$ la distancia de $s$ a él mismo con 0 y para el resto de los nodos en infinito. En un ciclo global, se recorren todas las aristas del grafo y dada una arista $(u, v)$ (suponiendo que estamos en un grado dirigido), si la distancia que tenemos guardada temporalmente en $D$ de $s$ a $u$ más el peso de $(u, v)$ es menor a la distancia temporal en $D$ de $s$ a $v$, entonces actualizamos $D[v]$ con el valor de la suma mencionada. Se puede ver que en cada iteración, para cada $v$, $D[v]$ tiene guardada la mejor distancia de $s$ a $v$ con a lo sumo $i$ aristas, como en un camino puede haber a lo sumo $n-1$ aristas, entonces es claro que el ciclo global debe iterar $n-1$ veces.

  **Invariante** *Bellman-Ford* tiene dos invariantes relevantes que se cumplen para cada nodo $u$ en cada iteración del ciclo externo:

  - Existe un camino desde $s$ hasta $u$ de valor $D[u]$ excepto que $D[u]$ valga infinito.
  - Luego de $i$ iteraciones del ciclo externo, para todos los caminos existentes de $s$ a $u$ de a lo sumo $i$ aristas, el largo del camino es mayor o igual a $D[u]$.

  **Complejidad** $O(n * m)$

- **Dijkstra y Dijkstra con Priority Queue**

  La lógica del algoritmo de *Dijkstra* es la siguiente:

  Sea $s$ el nodo desde el cual se quiere calcular los caminos mínimos contra el resto de los nodos del grafo $G$.

  1) Construimos un arreglo D representando todas las distancias de n al resto de los vértices con lo cual, al final del algoritmo, $D[v]$ tendrá el camino mínimo de n a v para todo vértice v de $G$. A su vez, en un conjunto V tenemos a los que todavía no le fueron calculados su distancia mínima. Al inicio, $D[s] = 0$ y $D[v] = \infty$ para el resto de los vértices, $V$ tiene a todos los nodos distintos de n.

  2) Si V es no vacío, seleccionamos $w \in V$ tal que $D[w]$ es la mínima distancia de todo el arreglo D y continuamos al paso 3, de ser V vacío pasamos al paso 4.

  3) Removemos a $w$ de $V$ y, iterativamente, para cada nodo $x$ adyacente a $w$ actualizamos su distancia temporal a $n$ si el camino de $n$ a $x$ pasando por $w$ es menos costoso que el calculado hasta ahora, es decir si se cumple

  $$peso((w, x)) + D[w] < D[x]$$

  entonces $D[x] = peso((w, x)) + D[w]$. Volvemos al paso 2).

  4) El algoritmo finaliza y para todo nodo $v$, $D[v]$ tiene exactamente el camino mínimo de $s$ a $v$

  **Invariante** En el paso iterativo iésimo tenemos calculados los caminos mínimos de los $i$ nodos más cercanos a $s$.

  **Complejidad** Si el conjunto $V$ de los nodos con camino mínimo se representa con un arreglo de tamaño $n$, que es lo que se hace para *Dijkstra* sin cola de prioridad, la complejidad del algoritmo es $O(n^2)$. Si $V$ se representa con un árbol binario de búsqueda, que es lo que hacemos en *Dijkstra* con cola de prioridad, la complejidad del algoritmo es $O(mlog(n))$.

Para tales algoritmos, consideramos el siguiente pseudocódigo:

**Algoritmo 2:** Ejercicio 2: uno a todos

**input :** Grafo G,int n,lista(int) precioNafta
**output:** conjunto ordenado de triplas de la pinta $(ciudad_1, ciudad_2, costoMinimo)$

```
1 listaAdy ← generarLista(G,n,precioNafta)                    // O(n^2)
2 crear matriz caminoMinimo
3 for i ← 0 to tamaño(listaAdy) do
4 |   en la fila i-ésima de la matriz caminoMinimo pongo el resultado que da
  |     algoritmoCMunoAtodos(listaAdy,i+1,tamaño(listaAdy))
5 end
6 solucion ← convertirSolución(caminoMinimo,n)                // O(n^2)
7 return solucion
```

### 2.3.2. Uno a Uno

- **A***
  Como vimos en las clases prácticas y de laboratorio de la materia, *Dijkstra* puede tomarse como un algoritmo que calcula camino mínimo de un nodo $s$ a otro $d$ si se finaliza su ejecución una vez calculada la distancia de $d$ en $D[d]$.$A*$ generaliza *Dijkstra* añadiendo una heurística de manera que el algoritmo no pierda tiempo explorando direcciones que se desvían del camino al nodo destino. La heurística es una función $h(x)$ definida para cada nodo x que estima la distancia de $x$ al nodo destino. En cada paso iterativo, en vez de tomar el nodo con mínima distancia al vértice fuente, se toma el nodo que minimice la suma entre la distancia al vértice fuente y la distancia estimada del nodo al nodo destino. Notar que si $h(x)$ es 0 obtenemos el algoritmo de *Dijkstra* de uno a uno. Solo vamos a usar $A*$ en los casos que nuestro grafo cumpla con la desigualdad triangular, ya que en caso contrario no podemos asegurar la correctitud del algoritmo

  **Complejidad** La complejidad de $A*$ es la misma que la de *Dijkstra* ya que no podemos asegurar que la heurística nos direccione en el sentido correcto. Es decir implementado con cola de prioridad es $O(mlog(n))$.

### 2.3.3. Muchos a Muchos

Los algoritmos utilizados para la resolución del problema *HIPERAUDITADOS* que calculan camino mínimo desde todos los vértices hacia todos son dos: *Floyd-Warshall* y *Dantzig*.

- **Floyd-Warshall**

  *Floyd-Warshall* es un ejemplo de algoritmo de programación dinámica. La función central subyacente de este algoritmo es $caminoMinimo(i, j, k)$, dicha función devuelve el camino mínimo de $i$ a $j$ utilizando únicamente los vértices $1, ..., k$ del grafo. Hay un caso base y uno recursivo, el caso base es $caminoMinimo(i, j, 0)$ es simplemente la arista $(i, j)$, el recursivo es $caminoMinimo(i, j, k) = \text{mín}(caminoMinimo(i, j, k - 1), caminoMinimo(i, k, k - 1) + caminoMinimo(k, j, k - 1))$

  Básicamente, si $k$ no es un vértice intermedio en el camino mínimo de $i$ a $j$ considerando únicamente los $k$ primeros vértices, entonces el camino mínimo de $i$ a $j$ tomando los primeros $k-1$ vértices es exactamente igual a aquel. En cambio, si $k$ es un vértice intermedio, entonces el camino mínimo puede ser dividido en dos partes, cada uno de los cuales usa los vértices en $\{1, ..., k - 1\}$ y en el cual $k$ es un punto intermedio.

  En este algoritmo se utiliza una matriz $D$ tal que $D[i][j]$ contiene el camino mínimo de $i$ a $j$

  **Invariante** Luego de $k$ iteraciones del ciclo más externo del algoritmo, la matriz $D[i][j]$ contiene el camino mínimo de $i$ a $j$ considerando los vértices de 1 a $k$ aparte de $i$ y $j$.
  **Complejidad** $O(n^3)$

- **Dantzig**

  El algoritmo de *Dantzig* es similar al de *Floyd* pero calcula en otro orden. En el paso késimo, en una matriz de $k \times k$ calcula los caminos mínimos del subgrafo que incluye a los primeros $1, ...k$ nodos, en la iteración siguiente agrega el nodo $k + 1$ a dicho grafo calculando los caminos mínimos entre este nuevo

vértice y el resto que ya están incluidos y se fija además si agregando a $k+1$ puede mejorar el camino mínimo previamente calculado entre pares de vértices del conjunto $\{1, ..., k\}$
**Invariante** Si $G_k$ es el subgrafo de $G$ que tiene a los vértices $1, ..., k$, entonces en la iteración késima tendremos calculados los caminos mínimos de todos a todos para el subgrafo $G_k$
**Complejidad** $O(n^3)$

Para tales algoritmos, consideramos el siguiente pseudocódigo:

**Algoritmo 3:** Ejercicio 2: todos a todos

```
   input  : Grafo G,int n,lista(int) precioNafta
   output: conjunto ordenado de triplas de la pinta (ciudad_1, ciudad_2, costoMinimo)
1  matAdy ← generarMatriz(G,n,precioNafta)                          // O(n^2)
2  caminoMinimo ← algoritmoCMtodosAtodos(matAdy)
3  solucion ← convertirSolución(caminoMinimo,n)                     // O(n^2)
4  return solucion
```

La función **convertirSolución** crea un conjunto ordenado de triplas (a,b,c) donde a y b son ciudades del problema original (nodos de G) tales que a < b, y c es el costo mínimo para ir de a hacia b en G. Dicha función tiene complejidad $O(n^2)$.

Las funciones **generarLista** y **generarMatriz** arman un nuevo grafo representado con listas de adyacencias y matriz de adyacencia, respectivamente. Tienen complejidad $O(n^2)$. A continuación se exponen sus respectivos pseudocódigos:

**Algoritmo 4:** generarLista

```
   input  : Grafo G,int n,lista(int) precioNafta
   output: Grafo G' asociado a G representado con lista de adyacencia
1  crear lista de adyacencia res de tamaño 61 * n        // la lista es un array de array de pares
                                                                   // O(n)
2  for i ← 0 to 61 * n do
3  |  for j ← 0 to 61 * n do
4  |  |  if cargo nafta en una ciudad then
5  |  |  |  ciudad_i ← coordX(i)                                   // O(1)
6  |  |  |  cantNafta_i ← coordY(i)                                // O(1)
7  |  |  |  cantNafta_j ← coordY(j)                                // O(1)
8  |  |  |  agrego el par (j, precioNafta[ciudad_i] * cantNafta_j − precioNafta[ciudad_i] * nafta_i) al final
   |  |  |   de res[i]                                             // O(1)
9  |  |  end
10 |  |  if me muevo de una ciudad a otra then
11 |  |  |  agrego el par (j, 0) al final de res[i]                // O(1)
12 |  |  end
13 |  end
14 end
15 return res
```

**Algoritmo 5:** generarMatriz

```
input : Grafo G,int n,lista(int) precioNafta
output: Grafo G' asociado a G representado con matriz de adyacencia
1  crear matriz res de tamaño 61 * n e inicializada en INF                          // O(n^2)
2  for i ← 0 to 61 * n do
3      for j ← 0 to 61 * n do
4          if i = j then
5              res[i][j] ← 0                                                         // O(1)
6          end
7          if cargo nafta en una ciudad then
8              ciudad_i ← coordX(i)                                                  // O(1)
9              cantNafta_i ← coordY(i)                                               // O(1)
10             cantNafta_j ← coordY(j)                                               // O(1)
11             res[i][j] ← precioNafta[ciudad_i] * cantNafta_j − precioNafta[ciudad_i] * nafta_i   // O(1)
12         end
13         if me muevo de una ciudad a otra then
14             res[i][j] ← 0                                                         // O(1)
15         end
16     end
17 end
18 return res
```

Como vimos en la sección anterior, en la parte del desarrollo, a partir de nuestro grafo $G$ de entrada, creamos un grafo $G'$ de $61*n$ nodos y con las características mencionadas en dicha sección (ésto es lo que hacen las funciones **generarLista** y **generarMatriz**). Ahora bien, las siguientes funciones muestran una correlación entre los nodos de $G'$ con la ciudad en la que estamos parados (nodo del grafo $G$) y la cantidad de nafta que tengo en ésa ciudad.

Sea $i \in nodos(G')$.

coordX(i) = parte entera de hacer $i/61$ = "ciudad o nodo de $G$"
coordY(i) = resto de hacer $i/61$ = "nafta que tengo en el tanque"

Veamos ahora qué significa ***cargo nafta en una ciudad*** y ***me muevo de una ciudad a otra.***

Sean $0 \leq i, j < 61 * n$, queremos ver cuándo hay una arista desde $i$ hasta $j$ en $G'$.

***Cargo nafta en una ciudad*** es verdadero cuando pasan las siguientes cosas:

- $i < j$
- "Queremos ir de una ciudad de $G$ a la misma ciudad", es decir, coordX(i) = coordX(j)
- "Queremos cargar de a un litro de nafta", es decir, coordY(i) + 1 = coordY(j)

***Me muevo de una ciudad a otra*** es verdadero cuando pasan las siguientes cosas:

- "Queremos ir de una ciudad de $G$ a otra ciudad diferente de $G$", es decir, coordX(i) $\neq$ coordX(j)
- "Hay una arista en $G$ entre las ciudades coordX(i) y coordX(j)"
- "Salimos de la ciudad coordX(i) de $G$ con coordY(i) litros de nafta y llegamos a la ciudad coordX(j) de $G$ con coordY(j) litros de nafta", es decir, coordY(i) - pesoArista(coordX(i),coordX(j)) = coordY(j)

Luego, si ***cargo nafta en una ciudad*** es verdadero o si ***me muevo de una ciudad a otra*** es verdadero entonces va a haber una arista desde $i$ hasta $j$ en $G'$.

## 2.4. Complejidad

La complejidad del algoritmo va a depender de qué función elija para calcular camino mínimo, entonces tenemos que las complejidades usando las distintas funciones son:

- **Dijkstra:** $O(n^3)$
- **DijkstraPQ:** $O(n * m * log(n))$
- **Bellman-Ford:** $O(n^2 * m)$
- **Floyd-Warshall:** $O(n^3)$
- **Dantzig:** $O(n^3)$

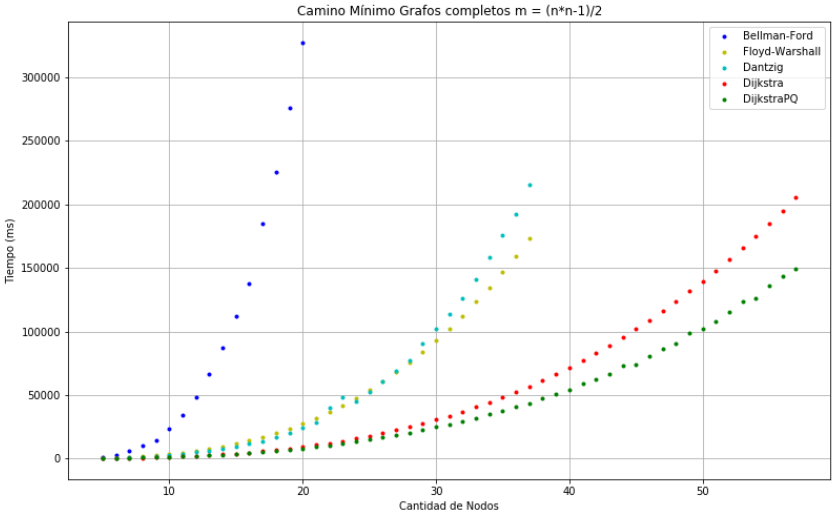
## 2.5. Experimentación

En base a una cantidad importante de test de casos generados aleatoriamente, graficamos los tiempos de ejecución de los diferentes algoritmos de camino mínimo. Mediante un scrypt de python generamos nuestras instancias y las corremos en nuestro código de c++ guardando los resultados en un archivo .csv. Luego graficamos estos resultados usando Jupyter Notebook y Pandas. Todos los experimentos se ejecutaron en las computadoras de los laboratorios del Departamento de Computación.
Para todos los experimentos dados a continuación vamos a suponer que excepto se aclare otra cosa:

- Los pesos de las aristas están distribuidos aleatoriamente entre $1 < a < 61$. No superan ese valor ya que para nuestro problema una arista con peso 61 sería igual a una con peso $infinito$ ya que nunca la podríamos recorrer. (nuestro auto solo tiene capacidad para 60 litros de nafta).
- Los pesos de los nodos fueron distribuidos aleatoriamente entre $1 < p < 100$.
- Dependiendo la complejidades de los algoritmos algunos no se prueban para todo el dataset debido al alto coste temporal asociado.
- Cada caso de test se ejecuto 5 veces y se usa la mediana para representarlos en el gráfico.

### 2.5.1. Grafos Completos

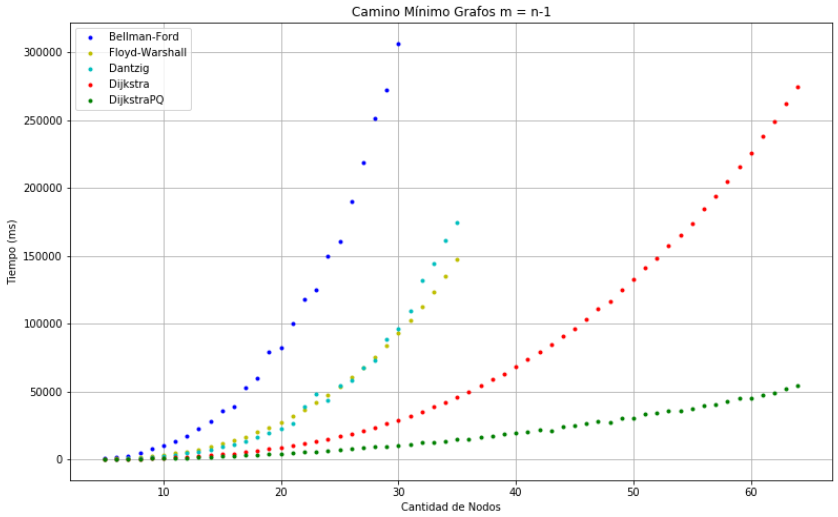
El primer experimento que hicimos fue para comparar los diferentes Algoritmos de Camino Mínimo en el caso de un grafo representando un conjunto de ciudades totalmente conectadas.
Para este experimento generamos datasets representando Grafos completos ($m = n * (n-1)/2$) variando la cantidad de nodos.



Como en este caso $m \in O(n^2)$ era de esperar que *Bellman-Ford* tenga peor complejidad que el resto de los algoritmos ya que queda $O(n^2 * m) = O(n^4)$. Sin embargo algo que se puede observar en la figura es que ambos algoritmos de *Dijsktra* (normal y con priority queue) tienen una perfomance mucho mejor que los algoritmos Matriciales. Tambien sorprende que *DijkstraPQ* tenga mejor tiempo de ejecución que *Dijkstra* ya que sus complejidades teóricas en este caso serian $O(n*m*log(n))$ y $O(n^3)$ Si bien *DijkstraPQ* tiene mejor complejidad que estos, *Dijkstra* tiene la misma complejidad en nuestro análisis inicial y debería tener una constante mas grande ya que los algoritmos matriciales solo hacen las cuentas mínimas para calcular la matriz final.

### 2.5.2. Árboles

Para este experimento generamos datasets representando grafos con una cantidad de aristas acotada ($m = n - 1$) variando la cantidad de nodos. En este experimento en contraste del anterior, tenemos una cantidad de aristas muy baja en relación al n, es decir es un grafo mucho mas esparzo, debido a esto deberíamos ver una mejora considerable en los algoritmos que usan listas de Adyacencias para representar el grafo (*Bellman-Ford* y las 2 versiones de *Dijkstra*).



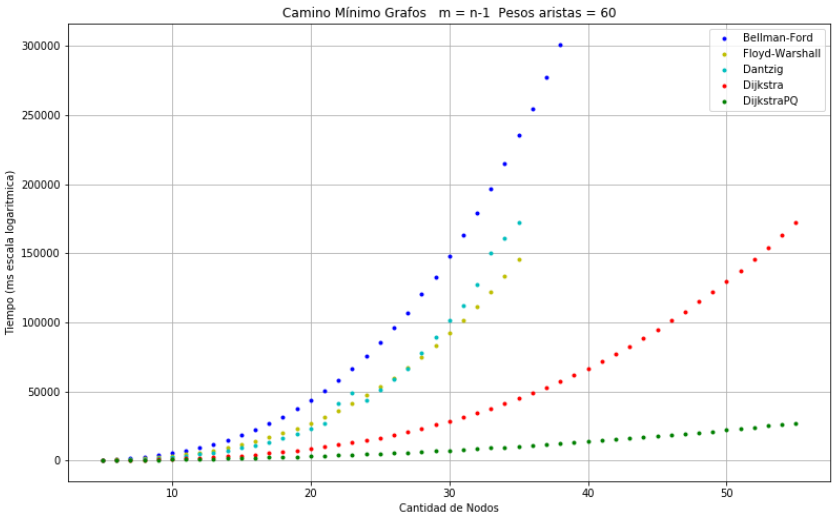

Como era de esperar podemos notar una mejora considerable en los algoritmos que que usan listas de adyacencia para representar nuestros grafos, ya que estos son esparsos. Las mejoras en *Bellman-Ford* no son suficientes para alcanzar el rendimiento de los algoritmos matriciales. *DijkstraPQ* muestra una mejora sobresaliente con respecto al resto.
Podemos notar que nuestros algoritmos matriciales (*Floyd-Warshall*, *Dantzig*) no presentan variaciones en estos gráficos. Esto es debido a que su complejidad esta asociada solamente con la cantidad de nodos del grafo y no tienen ninguna relación con la cantidad de aristas.
Si bien nuestros algoritmos que se basan en listas de adyacencias mejoraron sus tiempos de computo considerablemente, hasta ahora solo habíamos variado controladamente la cantidad de aristas. En los próximos experimentos también variaremos el peso de las aristas.

Como en el experimento anterior, en éste generamos datasets representando grafos que son árboles, es decir, $(m = n - 1)$ y variando la cantidad de nodos. Los pesos en los nodos fueron distribuidos aleatoriamente entre $1 < p < 100$ y los pesos de las aristas fueron fijados todos en 60.
Fijamos los pesos de las aristas en 60 porque consideramos lo siguiente:

nuestro algoritmo que resuelve el problema toma el grafo de entrada y le aplica la función **generarLista** o **generarMatriz** dependiendo de qué algoritmo de camino mínimo vamos a aplicar luego y nos genera un nuevo grafo $G'$ de $61 * n$ nodos. Como fijamos el peso de todas las aristas de nuestros grafos de entrada en 60, en el nuevo grafo $G'$ las aristas entre dos nodos diferentes de $G$ van a ser de la siguiente manera: $(c_i, 60) \rightarrow (c_j, 0)$ para

$c_i$ y $c_j$ nodos del grafo de entrada. Ésto se debe a que siempre vamos a tener que cargar la máxima cantidad de nafta posible en una ciudad (es decir 60 litros) para poder desplazarnos hacia otra ciudad pues la cantidad de nafta necesaria para atravesar dos ciudades en nuestro grafo de entrada es de 60 litros.

Nuestra hipótesis para este experimento es que los algoritmos que utilizan listas de adyacencia van a tener un mejor rendimiento en comparación con los que usan matriz de adyacencia.




Como se puede apreciar en el gráfico los tiempos de los algoritmos matriciales (*Floyd-Warshall*, *Dantzig*) se mantuvieron similares a los del experimento anterior. Mientras que los tiempos de *Bellman-Ford* comparándolos con los del experimento anterior mejoraron notoriamente, aunque esta idea de fijar los pesos de las aristas en 60 no le permitió a éste algoritmo aproximarse al algoritmo de *Dijkstra*, siendo estos dos últimos de igual complejidad para las instancias que estamos considerando en este experimento ($O(n^3)$ pues $m = n - 1$).

### 2.5.3. A* versus Dijkstra

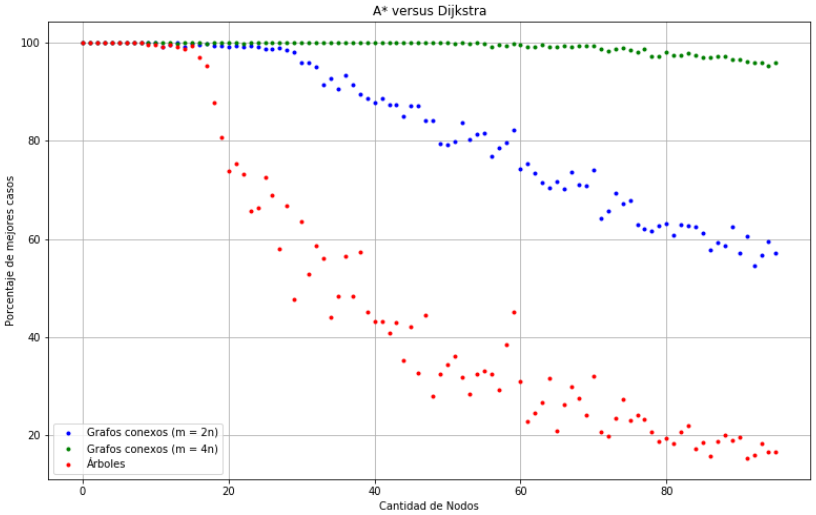
Dado que A* es un algoritmo que se puede aplicar a grafos que cumplen requisitos muy particulares, para poder experimentar con este algoritmo tuvimos que generar instancias de una forma muy específica y alternativa a la que hicimos para el resto de los experimentos del problema 2. Como A* lo consideramos para grafos en donde los nodos tienen una ubicación en el plano y en donde los pesos de las aristas son exactamente la distancia euclídea de los nodos en el plano, a partir de una entrada de n nodos y m aristas pasada como se indica en el enunciado, configuramos el siguiente grafo: dado un nodo $v \in \{1, \ldots, n\}$, definimos la función

$$f(v, n) = (cociente(v, \sqrt{n}), resto(v, \sqrt{n}))$$

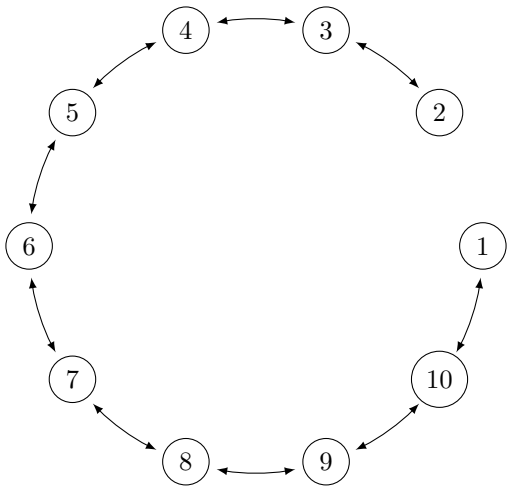
Luego, $f$ asocia a cada nodo $v$ del grafo con un par (x,y) del plano bidimensional en donde $x$ corresponde al cociente de dividir a $v$ por $\sqrt{n}$ e $y$ corresponde al resto de dicha división. De esta forma nos aseguramos de que los nodos queden bien distribuidos y que no estén todos alineados en una misma recta o en unas pocas rectas.

Una vez obtenida dicha distribución de los nodos en el plano bidimensional, a través de los parámetros de entrada obtuvimos la información de cuáles nodos estaban conectados entre sí con la excepción de que ahora, de estar conectados dos nodos $v$, $w$, el peso de la arista entre ellos es exactamente la distancia euclídea entre $f(v)$ y $f(w)$. Luego, la función estimadora $h$ de A* la tomamos como la distancia euclídea en el plano real.

Dado que, como ya explicamos antes, Dijkstra es un caso particular de A*, decidimos experimentar contrastando estos dos algoritmos y, para cada instancia $s$ de grafo del dataset, el experimento consistió en calcular del número de caminos mínimos de todos los nodos a todos, en cuántos casos A* supera en tiempo a Dijkstra y mostrar el porcentaje de dicho cálculo. Como ya a partir de la relación $m = 4n$, A* daba como mínimo en el noventa por ciento de los casos mejor que Dijkstra, decidimos no seguir aumentando el número de aristas. A continuación se muestran los gráficos de los porcentajes para tres casos: árboles, $m = 2n$ y $m = 4n$.




La decisión de probar el experimento en un comienzo con árboles e ir incrementando la cantidad de aristas partiendo de esta base está fundada en la siguiente hipótesis: dados dos nodos $v$ y $w$, si se quiere calcular el camino más corto de $v$ a $w$, A* siempre va a intentar tomar atajos e ir lo más directo posible de $v$ a $w$. Ahora, este algoritmo no tiene en cuenta que tomando las rutas más directas, al final quizás esta ruta es un callejón sin salida porque no hay una conexión del último nodo de dicha ruta a $w$, por lo que va a tener que volver sobre sus pasos y recalcular nuevamente si esta en este caso. En un grafo donde hay muy pocas formas de llegar de un nodo a otro (cuantas menos aristas haya, menos posibilidades), es muy probable que pase esta situación. Podemos ver en el siguiente ejemplo que pasa lo mencionado si calculamos el camino mínimo del nodo $v = 9$ al nodo $w = 1$. Tomando la función estimadora como $|a - b|$ siendo a y b los 2 nodos considerados en la iteración actual. Suponiendo que las aristas no tienen peso.




Podemos ver que Dijkstra llegaría a la solución en la segunda iteración y A* recorrería todo el grafo y llegaría a la solución recién en la iteración número 10. A medida que aumentamos la cantidad de aristas disminuye la cantidad de callejones sin salida en nuestro grafo por lo que aumenta el porcentaje de victorias de A* sobre Dijkstra como pudimos comprobar en los gráficos expuestos.

## 2.6. Conclusiones

1. **Algoritmos Matriciales**
   Como los algoritmos matriciales solo dependen del tamaño de la matriz (la cantidad de nodos en este caso), solo varían sus tiempos de ejecución al variar estos. Pudimos comprobar que esto se cumple con nuestra experimentación ya que variando el m (cantidad de aristas) de nuestra matriz no vario el tiempo de ejecución de estos algoritmos.

2. **Algoritmos Dijkstra**
   Los algoritmos de Dijkstra fueron los que se mostraron con mejor perfomance en todos los casos de experimentación, incluso para los casos en que $m \in n^2$. Particularmente Dijkstra con Priority queue fue el que dio mejores resultados y es el que recomendaríamos para resolver este problema.

3. **Bellman-Ford**
   El algoritmo de BellmanFord tuvo los peores tiempos de ejecución de todas las opciones, incluso en los casos en que nuestros grafos eran árboles ($m = n - 1$). Si bien era esperable que Bellman-Ford fuera peor que el resto no se esperaba que su rendimiento fuera tan malo.

4. **A*** Como ya analizamos en el experimento pertinente, A* es más rápido que Dijkstra para grafos conexos no muy esparsos (vimos que en árboles, Dijkstra lo superaba en la mayoría de los casos). Dado que a su vez, en el resto de la experimentación, Dijkstra fue el que arrojó mejores tiempos de ejecución siempre que el resto de los algoritmos, podemos concluir que para las instancias de grafos que cumplen con las condiciones necesarias para aplicar el algoritmo de A*, este será el más eficiente de todos los algoritmos vistos en este problema siempre y cuando tengamos grafos con por lo menos $m = 2n$.

## 2.7. Futuros Experimentos

Debido al tiempo limitado con el que se contó para la realización del trabajo dejamos para futuro análisis:

- Implementar los experimentos en casos mas extensos para verificar que nuestras hipótesis se mantienen aun para casos cuyo tiempo de ejecución sea muy grande.
- Probar diferentes funciones estimadoras para A* y comparar sus rendimientos.
- Analizar complejidades espaciales de nuestros algoritmos.
- Considerar el caso de grafos con nodos con pesos negativos y como afecta nuestra implementación.
- Probar otras heurísticas diferentes para calcular camino mínimo.