DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº2

Yéndose por las ramas

27 de Mayo de 2018 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bonaccini, Adrián	207/04	abonaccini@dc.uba.ar
Catalano, Arístides	279/10	aristidescata@gmail.com
Musso, Bruno Martín	676/11	brunomusso91@hotmail.com
Romera, Joaquin	183/16	jromera@dc.uba.ar

Entrega	Agustina Ciraco	I
Reentrega		

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

Índice

1. Introducción

En el siguiente trabajo se abordará la resolución de dos problemas relacionados con búsquedas de mínimos en grafos. En el primero se optimizará el costo de instalación de una red de conexiones entre distintas ciudades. En el segundo se dará cuenta del mínimo consumo de combustible que hará falta para recorrer la distancia entre cualquier par de ciudades del problema. Será imprescindible entonces presentar un panorama amplio de las herramientas que nos permitan llegar a soluciones óptimas en tiempos polinomiales.

En principio, para resolver el primer problema (*Hiperconectados*) y generar la red de conexión entre ciudades más barata alcanzaría con obtener un *árbol generador mínimo*. Este proporciona, dado un *grafo conexo*, un árbol que contiene todos los nodos del grafo original y la mínima cantidad de ejes para mantenerlo conexo, cuyos pesos sean además mínimos. Se han implementado varios algoritmos de los cuales luego valoraremos su desempeño en términos de complejidad temporal: *Prim*, *Kruskal* y *Kruskal con Path Compression*. Sin embargo, es necesario dar respuesta a un problema menos inmediato que el de la obtención de dicho *AGM*.

En la búsqueda de una solución, los mencionados algoritmos serán utilizados como subprocesos de otro algoritmo que pretende dar repuesta al problema de decidir si los caminos que conectan ciudades pertenecen, *siempre*, *a veces* o *nunca*, a la red menos costosa.

Para resolver el segundo problema (*Hiperauditados*), se tendrán en cuenta dos aspectos: por un lado, modelar como un grafo el problema de minimizar el costo de recorrer las ciudades y, por el otro, obtener el camino mínimo entre todo par de nodos. Para modelar el problema se considerará el consumo del combustible de un punto al otro y su precio en cada ciudad del recorrido.

Si bien ambos problemas están conectados en tanto pretenden resolver la búsqueda de mínimos en grafos -árboles generadores o caminos-, será necesario ahondar en dos lógicas que fueron necesarias para la resolución de dichos problemas:

- ***Hiperconectados***: teóricamente el problema podría resolverse si fuera posible analizar todos los posibles *AGM* del grafo con el que se trabaja. Dicha exploración no pertenece al campo de los tiempos polinomiales y por eso se descarta en función de una estrategia que hace uso de la definición misma de *AGM*.
- ***Hiperauditados***: el esfuerzo se centra en el modelado del grafo sobre el cual correrán los algoritmos de camino mínimo. Si bien las implementaciones de los mismos son familiares, la solución del problema no tiene la misma complejidad que los subalgoritmos sobre los cuales se modela el mismo. Dicho procedimiento será identificado como el pasaje del *grafo* de entrada del problema a un *grafo de estados*.

2. Changelog

- Cambios estéticos al diseño del informe
- Se revisaron errores de ortografía y se mejoró la redacción en general.
- Las pruebas experimentales de la complejidad de cada algoritmo se detallan en la sección de Experimentación.
- Se agregó la lista de archivos entregados y su descripción en la sección 1.2.
- Se realizaron optimizaciones de código que repercutieron mínimamente en los pseudocódigos y cálculos de complejidad.
- A* tiene una sección nueva detallando cuándo puede ser utilizado, la heurística elegida y su comparación con Dijkstra PQ.
- Nuevos experimentos agregados a la sección de Experimentación.
- Entre los archivos entregados se agregó README.md explicando cómo compilar y correr los programas.
- Se realizaron los siguientes cambios teniendo en cuenta las correcciones recibidas:
  - En la sección 2.2.4 se aclaró que el uso de los algoritmos para buscar AGMs fueron utilizados sin modificaciones y que tienen las complejidades vistas en las clases teóricas.
  - Se detalló cómo varía la complejidad en la sección 2.4 según el subalgoritmo elegido.
  - Los gráficos correspondientes a la experimentación sobre las complejidades de Hiperconectados están ahora en la sección de Experimentación. A su vez fueron detallados con mayor profundidad.
  - Se mejoró el análisis de correctitud de Hiperconectados y su redacción.
  - Se demostraron los casos con mayor formalidad en las secciones 2.5.1, 2.5.2, y 2.5.3.
  - El análisis de la complejidad de Hiperauditados se movió a la sección de Experimentación y se mejoró su desarrollo.
  - La sección de Experimentación se modificó íntegramente, detallando mejor cómo se armaron los casos de pruebas, el entorno para realizarlas y explicando las hipótesis y las expectativas de cada experimento.

3. Archivos

3.1. Principales

- ./Makefile *(se incluye para compilar todos archivos)*
- ./README.md *(detalla cómo correr los programas Hiperconectados e Hiperauditados y sus aridades correspondientes)*
- ./commonTypes.h *(header de tipos comunes definidos para todas las funciones)*
- ./generator.cpp *(generador de grafos para ambos problemas)*
- ./euclideangraphs.cpp *(generador de grafos para A*)*
- ./hiperauditados.cpp
- ./hiperauditados.h
- ./hiperconectados.cpp
- ./hiperconectados.h
- ./astarvsdpq.cpp

3.2. Hiperauditados

- ./astar.cpp
- ./astar.h
- ./bellmanford.cpp
- ./bellmanford.h
- ./dantzig.cpp
- ./dantzig.h
- ./dijkstra.cpp
- ./disjktra.h
- ./dijkstraold.cpp
- ./dijkstraold.h
- ./dijkstrapq.cpp
- ./dijkstrapq.h
- ./floydwarshall.cpp
- ./floydwarshall.h

3.3. Hiperconectados

- ./kruskal.cpp
- ./kruskal.h
- ./kruskalpc.cpp
- ./kruskalpc.h
- ./prim.cpp
- ./prim.h

3.4. Otros

- ./grafos.sh
- ./eucligrafos.sh

4. Hiperconectados

4.1. Introducción

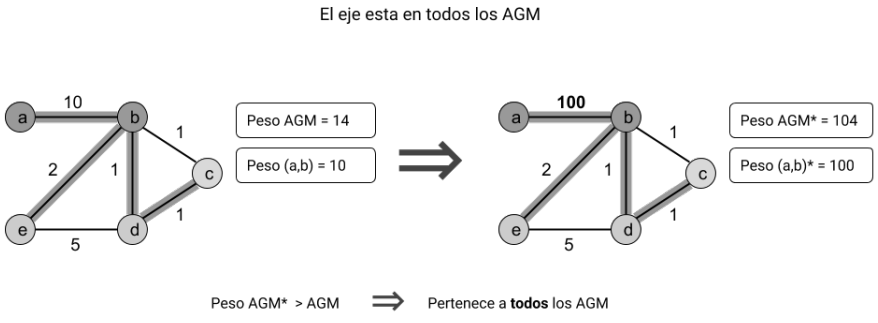
Este ejercicio consiste en determinar si, dado un conjunto de v ciudades y e potenciales conexiones con un costo asignado, cada conexión pertenece a *toda*, *alguna* o *ninguna* red mínima denominada Anexión General Minúscula, siendo esta una red que abarca todas las ciudades con un costo mínimo.

4.2. Solución

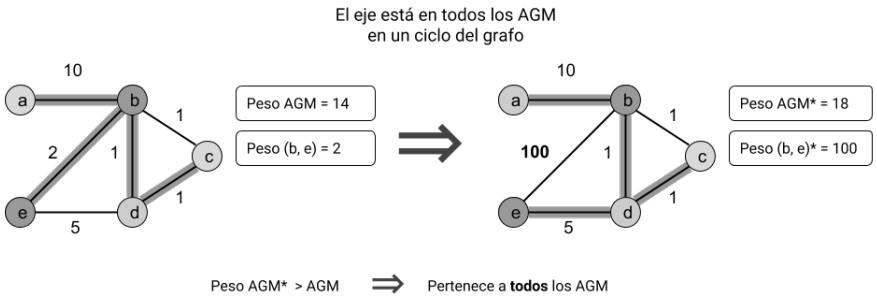
Sea $G = (V, E)$ el grafo no orientado que modela el mapa de conexiones entre ciudades, donde cada $v \in V$ representa una ciudad y cada eje $e = (v, w) \in E$ representa una ruta entre la ciudades v y w. Además cada e tiene un peso asociado, que es el costo de viajar entre las ciudades que une. Este costo siempre es un valor positivo.

4.2.1. Está en todos los AGM

Para saber si una conexión pertenece a todos los AGM se partirá de la siguiente premisa: si un eje está en **todos** los AGM sucede que al modificar su peso, también se modificará el peso del AGM. En consecuencia, para detectar si un eje e está presente en todos los casos, se le asigna un valor mayor al peso máximo del grafo y se calcula nuevamente el AGM para el grafo modificado, al que llamaremos AGM*. Si el valor de AGM* es mayor al del AGM original, esto indica que e estaba incluido en todos los AGM. En caso contrario, puede pasar que no esté en ninguno o que esté en algunos.



En la figura se ve que el eje e=(a,b) debe estar incluido en todos los AGM, ya que si no fuera así, quedaría mas de una componente conexa. En el ejemplo se ve que es lo que sucede cuando cambiamos el peso y obtenemos nuevamente el AGM. Como siempre esta incluido el peso del AGM termina siendo mayor.

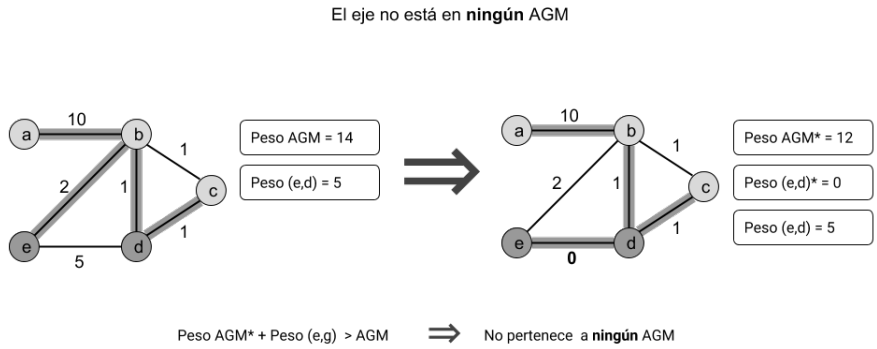


En este ejemplo, (b,e) está en un ciclo, con lo cual para un peso suficientemente grande, se va a encontrar una mejor opción, (d, e) en este caso. De todas formas el peso del AGM* termina siendo mayor que el del AGM original., entonces (b,e) tiene que estar en todos los AGM.

4.2.2. No está en ningún AGM

Para saber si una conexión no pertenece a ningún AGM se tuvo en cuenta lo siguiente: el eje e debe pertenecer a un ciclo del grafo, de lo contrario el eje estaría en todos los AGM y por lo tanto sería equivalente al caso anterior. Se procede a asignarle un valor mínimo al eje e y se vuelve a calcular el peso del AGM. Sabemos que

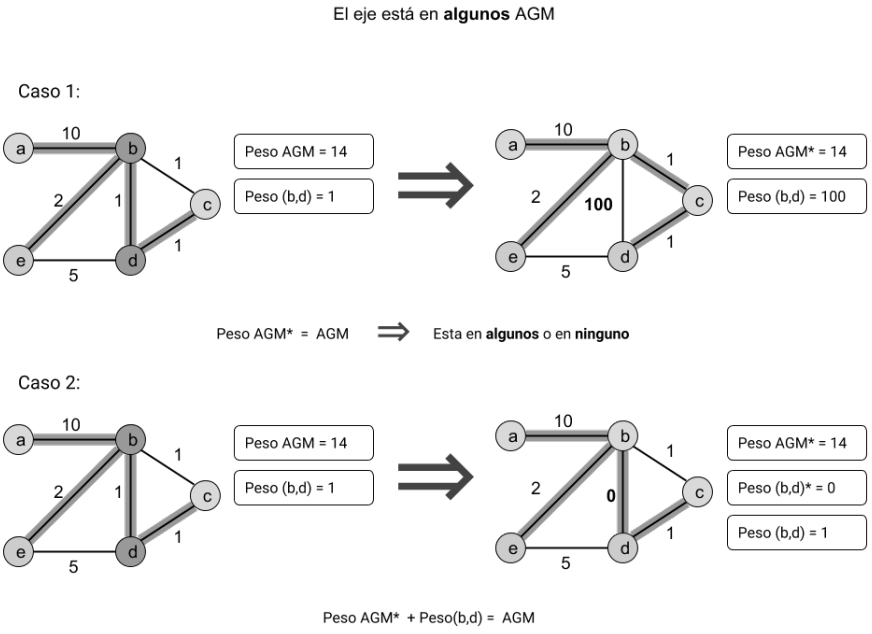
el eje ahora está incluido en el nuevo AGM, al cual llamamos AGM*. Luego se suma el peso original de e al peso de AGM* para poder saber el peso del AGM incluyendo a e. Si dicho peso es mayor al del AGM original, el eje e no se incluye en **ningún** AGM.



En el ejemplo, se ve que el eje e=(e,d) no forma parte de ningún AGM. Entonces se le asigna peso 0 para "forzar" a incluirlo en un nuevo AGM*. Luego se suman los pesos del AGM* obtenido con el peso original del eje. Como la suma es mayor los peos del AGM original, entonces se puede decir que (e,d) no esta en ningún AGM.

4.2.3. Está en algún AGM

Si las pruebas no fueron concluyentes ni para el primer ni el segundo caso, entonces el eje pertenecerá a **algunos** AGM.



En el ejemplo, vemos que sucede con un eje que está en algunos AGM. Para el primer caso, si se aumenta el peso del mismo no modifica el peso total del AGM*.
Luego en el segundo caso, cuando se fuerza a incluir ese eje y luego se suma el peso, tampoco pasa que esta suma sea mayor al AGM original.

4.2.4. Obtener cada AGM

Para poder resolver el problema de cómo obtener un AGM se utilizaron algoritmos cuya correctitud y complejidad fueron dadas en las clases teóricas y los laboratorios de la materia.

- Algoritmo de Prim.

- Algoritmo de Kruskal.
- Algoritmo de Kruskal con *Path Compression*.

Asimismo estos algoritmos fueron implementados sin modificaciones que afecten su complejidad. Las complejidades y las implementaciones de estos sub-algoritmos se detallan en la sección en la que se calcula la complejidad de Hiperconectados.

4.3. Pseudocódigo

El siguiente pseudocódigo describirá cómo resolver el problema de *Hiperconectados* donde la función *getAGM* llama a alguno de los algoritmos referidos anteriormente (Prim, Kruskal o Kruskal con Path Compression). Estos devuelven la suma de los pesos de todas las aristas del *AGM*. Si bien los algoritmos no devuelven el *AGM*, han sido modificados para que en cada iteración además de obtener los ejes que pertenecen al *AGM*, también calculen la suma de los pesos de cada eje que formará el árbol. De esta forma no se modifica la complejidad asintótica de los algoritmos.

```
1:  function HIPERCONECTADOS(array[edge] edgeList, int v, int e, int alg)
2:      array[string] solution ← CreateArray(e)                          ▷ O(e)
3:      int formerAGMCost ← getAGM(edgeList, v, e, alg)                  ▷ O(alg)
4:      for int i ← 1; i ≤ e; ++i do
5:          int edgeCost ← edgeList[i].weight                            ▷ O(1)
6:          edgeList[i].weight ← ∞                                       ▷ O(1)
7:          int excludingEdgeCost ← getAGM(edgeList, v, e, alg)          ▷ O(alg)
8:          edgeList[i].weight ← 0                                       ▷ O(1)
9:          int includingEdgeCost ← getAGM(edgeList, v, e, alg)          ▷ O(alg)
10:         if excludingEdgeCost > formerAGMCost then
11:             solution ← solution.push_back("toda")                    ▷ O(1)
12:         else
13:             if includingEdgeCost > formerAGMCost − edgeCost then     ▷ O(1)
14:                 solution ← solution.push_back("ninguna")             ▷ O(1)
15:             else
16:                 solution ← solution.push_back("alguna")              ▷ O(1)
17:             end if
18:         end if
19:         edgeList[i].weight ← edgeCost                                ▷ O(1)
20:     end for
21:     return solution
22: end function
```

4.4. Complejidad Temporal

Estos son los costos no triviales del pseudocódigo:

- Línea 2: La creación del arreglo *solution* para almacenar el resultado de cada eje: O(e)
- Línea 3: La complejidad de la llamada inicial al algoritmo elegido para obtener el peso del AGM del grafo depende del algoritmo elegido. En el caso de Prim es O(e log(v)), para Kruskal (con y sin Path Compression) es O(v + e log(v))
- Línea 4: Se itera para cada eje: O(e) y luego:
  - Líneas 7 y 9: En ambas se llama al algoritmo para obtener el peso del AGM con el grafo modificado, tienen la misma complejidad que la línea 3.

Como para estos problemas estamos trabajando con grafos conexos tenemos que $e \geq v - 1$. Luego podemos acotar la complejidad de ambos Kruskal de la siguiente manera: $O(v+e\ log(v)) = O(e+e\ log(v)) = O(e\ log(v))$. Entonces tenemos que la complejidad de Hiperconectados es:

$$\mathrm{O}(\textit{Hiperconectados}) = \mathrm{O}(e) + \mathrm{O}(e\ \log(v)) + \mathrm{O}(e)\ 2\ \mathrm{O}(e\ \log(v)) = \mathrm{O}(e^2 \log(v))$$

4.5. Análisis de Correctitud

Para demostrar la correctitud de nuestra solución a Hiperconectados, debemos probar que para cada eje del grafo el algoritmo devuelve si el mismo está en **todos**, en **algunos** o en **ningún** *AGM*. Es importante notar que se trata de una tricotomía, es decir, cada eje pertenece a una sola de estas tres categorías. Dicha característica es crucial para la justificación de la correctitud de la solución propuesta.

Análisis del pseudocódigo descripto en la sección **2.3**:

- En la línea 3, se calcula el peso de algún *AGM* dado un grafo de entrada. Para calcular los *AGM* se utilizan algoritmos presentados y demostrados en el contexto de la materia por lo cual asumimos su correctitud.
- A partir de la línea 4 comienza un ciclo que itera una cantidad de veces igual a la cantidad de ejes en el grafo, de esta forma podemos asegurar que cada eje va a ser analizado. Luego en el cuerpo del ciclo, por cada eje se almacena su peso original para restaurarlo después de hacer las comparaciones y los cálculos.
- En la línea 6 se le asigna a la arista siendo analizada un peso que representa un valor mayor al peso del AGM, de esta manera cuando se calcule el nuevo *AGM* en la linea 7, o bien la arista será excluida o hará que el peso del nuevo *AGM* (*excludingEdgeCost*) sea mayor que el original.
- En la línea 8 se asigna peso 0 al eje siendo analizado, de forma tal que cuando se vuelva a calcular el peso del *AGM* (*includingEdgeCost*), el eje esté incluido en el nuevo *AGM*.

Entre las líneas 10 y 18 se decide a qué categoría pertenece el eje que se está analizando. Para facilitar este análisis, se lo separa en los tres casos descriptos en la sección **2.2**:

4.5.1. Caso 1

En este caso vemos qué pasa cuando al eje se le asigna un peso grande (mayor que el peso del AGM). Si calculamos nuevamente un AGM sobre el grafo con el peso nuevo pueden pasar 2 cosas:

- El peso del AGM nuevo es mayor que el peso del AGM original.
  Hay dos explicaciones posibles: el eje une dos componentes conexas del grafo, es decir, no hay forma de unir estas dos componentes si no es por este eje; o el eje está en un ciclo y es el menos costoso de todos los ejes del mismo -si hubiera uno mejor estaría en el AGM original y el peso no hubiera variado-.
- El peso del AGM nuevo es igual al peso del AGM original.
  En este resultado el eje no está en todos los AGM, puede ser que esté en algunos o que no esté en ninguno.

4.5.2. Caso 2

En este momento ya podemos descartar que el eje esté en **todos** los AGM. Además, por esa misma razón sabemos que está en algún ciclo del grafo. Ahora le asignamos peso 0 al eje, calculamos el peso del nuevo AGM y lo comparamos con el peso del AGM original. Luego le sumamos el peso original del eje para obtener el peso del AGM que lo incluye. Entonces tendremos dos escenarios distintos:

- $includingEdgeCost + pesoEjeOriginal > AGMoriginal$.
  Como la suma es mayor al AGM original entonces el árbol generador nuevo no es un árbol generador mínimo del grafo original. Luego, el eje no está en **ningún** AGM.
- $includingEdgeCost + pesoEjeOriginal = AGMoriginal$.
  Si el nuevo peso sigue siendo igual al del AGM original, entonces se trata de un eje que puede estar o no (ya se descartó que esté en todos por el caso 1).

4.5.3. Caso 3

El último caso se obtiene cuando el eje no cumple con ninguno de los dos anteriores: si el eje no cumple con el primero, entonces no está en todos los AGM y además tiene que estar en algún ciclo del grafo; en el segundo caso, descartamos que el eje no esté en ningún AGM ya que no varía el peso del mismo cuando se lo fuerza a incluir ese eje. Luego al ser una tricotomía solo puede pasar que el eje analizado esté en **algunos** AGM.

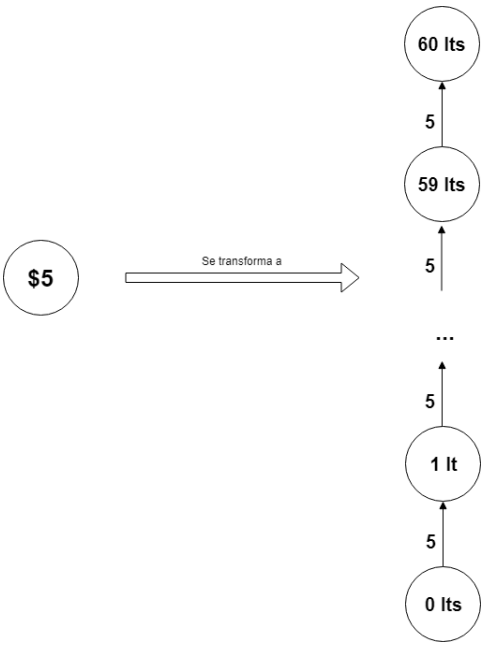
5. Hiperauditados

5.1. Introducción

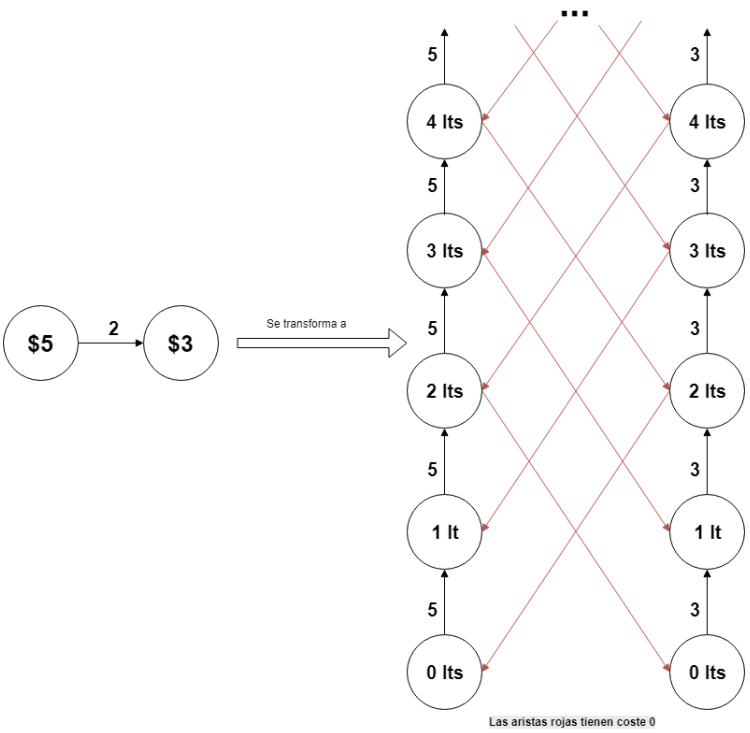
Este problema busca encontrar la manera más eficiente de recorrer todas las v ciudades una vez ya instalada la red del problema anterior, teniendo en cuenta que cada ciudad posee su propio valor para la nafta y que el vehículo en el cual se realizará el recorrido tiene un tanque con capacidad máxima de 60 litros.

Como cada ciudad posee un precio diferente para el combustible, el camino que gaste menos no será necesariamente el más corto. Podría suceder que haya ciudades conectadas donde ir de una a la otra requiera pocos litros de nafta, pero donde su costo sea muy alto; o lo opuesto: que la ruta que las conecte demande un mayor gasto en litros pero que la nafta esté muy barata en la ciudad de partida. Para solucionar esto, el grafo original se transformará de la siguiente forma:

- Por cada vértice se crearán 61 vértices que representaran la cantidad de nafta que se posee (desde 0 que equivale al tanque vacío hasta 60 que equivale al tanque lleno). Estos estarán conectados por ejes dirigidos con el costo de la nafta de dicha ciudad entre sí. A estos los llamaremos *vértices estado*, ya que cada uno indica un estado de la cantidad de nafta en el tanque y tendrá un eje dirigido hacia el siguiente estado con el costo de cargar un litro más de nafta.



- Por cada par de vértices vinculados en el grafo original ahora habrá un eje dirigido saliendo de cada vértice original por cada estado factible. Es decir, por cada estado del vértice de partida -con suficiente combustible- habrá un eje al vértice de llegada en un estado que representa la cantidad de litros de nafta utilizados para transitar la ruta que conecta ambas ciudades.



De esta manera, el nuevo grafo representará todas las soluciones posibles con sus vértices estado, facilitando la forma de obtener todas las acciones posibles a realizar. Los ejes que conectan los vértices estado que pertenecen al mismo vértice original representan el cargar nafta, y cada vez que se los recorre se aumenta el costo del camino con lo que costó cargar esa cantidad de nafta. A su vez, cada uno de éstos tienen un camino de costo 0 hacia cada vértice que estaba conectado en el grafo original. El mismo lo lleva a la otra ciudad en el vértice estado que representa el gasto de nafta realizado para llegar a dicha ciudad. Finalmente la solución del problema se reduce a encontrar un camino mínimo entre cada par de vértices del nuevo *grafo estado*.

5.2. Pseudocódigo

En el algoritmo de *Hiperauditados* se asume que el tanque empieza lleno en el vértice de partida. Luego de transformar el grafo de entrada en el *grafo estado* correspondiente se busca un camino mínimo entre cada par de vértices estado. Por último se calcula a qué ciudades pertenecen cada par de nodos de los caminos mínimos y se elige el de mínimo costo que llega a uno de los 61 estados posibles de la ciudad destino. A diferencia de *Hiperconectados*, la complejidad del algoritmo depende fuertemente de qué subalgoritmo se use para calcular los caminos mínimos. Esto se encuentra representado como *GetMinCosts* en el pseudocódigo.

```
1:  function HIPERAUDITADOS(array[int] litreCostByCity, array[edge] edgeList, int v, int e, int alg)
2:      int states ← 61                                                          ▷ O(1)
3:      array[edge] stateGraph ← CreateStateGraph(litreCostByCity, edgeList, states)   ▷ O(e)
4:      int nodesSG ← v · states                                                 ▷ O(1)
5:      int edgesSG ← |stateGraph|                                               ▷ O(1)
6:      int from ← states - 1                                                    ▷ O(1)
7:      matrix[int] solSG ← GetMinCosts(stateGraph, nodesSG, edgesSG, alg)       ▷ O(ChosenAlg)
8:      array[int] sol ← StateGraphToGraph(sol)                                  ▷ O(v²)
9:      return sol
10: end function                                                                 ▷ Total: O(ChosenAlg)
```

```
1: function CreateStateGraph(array[int] litreCostByCity, array[edge] edgeList, int states)
2:     array[edge] sol ← CreateArray()                                          ▷ O(|edgeList|)
3:     for nat i ← 2; i ≤ states; ++i do                                        ▷ O(1)
4:         int j ← i - 1                                                        ▷ O(1)
5:         for nat k ← 1; k ≤ litreCostByCity; ++k do                           ▷ O(|litreCostByCity|)
6:             sol ← sol.push_back(<j + states · k, i + states · k, litreCostByCity[k]>)   ▷ O(1)
7:         end for
8:         for edge edge in edgeList do                                         ▷ O(|edgeList|)
9:             int nodeA ← edge.a                                               ▷ O(1)
10:            int nodeB ← edge.b                                               ▷ O(1)
11:            int weight ← edge.p                                              ▷ O(1)
12:            if i ≥ weight then                                               ▷ O(1)
13:                sol ← sol.push_back(<i + nodeA · states, i − weight + nodeB · states, 0>)   ▷ O(1)
14:                sol ← sol.push_back(<i + nodeB · states, i − weight + nodeA · states, 0>)   ▷ O(1)
15:            end if
16:        end for
17:    end for
18:    return sol
19: end function                                                                ▷ Total: O(|edgeList|)
```

5.3. Complejidad Temporal

Para poder encarar el análisis de la complejidad temporal del algoritmo, se detallará la transformación del grafo necesaria para la resolución del problema junto con el análisis del pseudocódigo y el costo de su utilización.

5.3.1. CreateStateGraph

La complejidad del algoritmo *CreateStateGraph* es lineal en la cantidad de ejes del grafo de entrada:

- Línea 3: en el contexto de este problema *states* siempre es 61, por lo que este ciclo itera una cantidad de veces constante, y es O(1)
- Línea 5: este ciclo itera por el tamaño de *litreCostByCity* creando los ejes posibles que conectan a los vértices estado que representan las cantidades posibles de nafta en cada vértice del grafo original. O(|*litreCostByCity*|)
- Línea 8: este ciclo itera por cada eje del grafo original creando los ejes que conectan los distintos estados de cada par de vértices conectados en el grafo original, por lo que cuesta O(|*edgeList*|).

Como *litreCostByCity* tiene el mismo tamaño que la cantidad de vértices del grafo y *edgeList* como mínimo vale la cantidad de vértices -1, tenemos que (|*litreCostByCity*|) $\in$ O(|*edgeList*|), dejando el costo de *CreateStateGraph* en O(|*edgeList*|).

5.3.2. Hiperauditados

Para *Hiperauditados* existen solo 2 costos no triviales:

- Línea 3: La llamada a *CreateStateGraph* que cuesta O(|*edgeList*|).
- Línea 7: La llamada a *GetMinCosts* que depende cota del algoritmo utilizado, sus implementaciones respetan las complejidades vistas en las clases teóricas de la materia:

$Dijkstra$	$O(v^2 + e)$
$Dijkstra\ Priority\ Queue$	$O(e \cdot log(e))$
$A*$	$O(e \cdot log(e))$
$Bellman - Ford$	$O(v \cdot e)$
$Floyd - Warshall$	$O(v^3)$
$Dantzig$	$O(v^3)$

Los subalgoritmos de camino mínimo de un nodo a muchos se corren una vez por cada nodo, multiplicando su complejidad por: $O(v \cdot states)$.

- Línea 8: La llamada a *StateGraphToGraph* implica recorrer la matriz de $(v \cdot states)^2$ para fijarse cuál de todos los vértices estados de cada par de ciudades tiene el mejor gasto en combustible, como *state* es una constante, recorrer una matriz de v^2 cuesta $\mathrm{O}(v^2)$.

Entonces los posibles valores de complejidad son:

$$O(Hiperauditados) = \begin{cases} \mathrm{O}(v^2) + O(e) + O(v \cdot states) \cdot O(Dijkstra) & Dijkstra \\ \mathrm{O}(v^2) + O(e) + O(v \cdot states) \cdot O(DijkstraPQ) & DijkstraPQ \\ \mathrm{O}(v^2) + O(e) + O(v \cdot states) \cdot O(A*) & A* \\ \mathrm{O}(v^2) + O(e) + O(v \cdot states) \cdot O(BellmanFord) & Bellman - Ford \\ \mathrm{O}(v^2) + O(e) + O(FloydWarshall) & Floyd - Warshall \\ \mathrm{O}(v^2) + O(e) + O(Dantzig) & Dantzig \end{cases} \quad (1)$$

Reemplazando las complejidades de cada subalgoritmo y $O(v \cdot states)$ por $O(v)$ -*states* es una constante-:

$$O(Hiperauditados) = \begin{cases} \mathrm{O}(v^2) + O(v) \cdot O(v^2 + e) & Dijkstra \\ \mathrm{O}(v^2) + O(v) \cdot O(e \cdot log(e)) & DijkstraPQ \\ \mathrm{O}(v^2) + O(v) \cdot O(e \cdot log(e)) & A* \\ \mathrm{O}(v^2) + O(v) \cdot O(v \cdot e) & Bellman - Ford \\ \mathrm{O}(v^2) + O(v^3) & Floyd - Warshall \\ \mathrm{O}(v^2) + O(v^3) & Dantzig \end{cases} \quad (2)$$

Entonces la complejidad para cada subalgoritmo queda igual a:

$$O(Hiperauditados) = \begin{cases} \mathrm{O}(v^3 + e) & Dijkstra \\ \mathrm{O}(v \cdot e \cdot log(e)) & DijkstraPQ \\ \mathrm{O}(v \cdot e \cdot log(e)) & A* \\ \mathrm{O}(v^2 \cdot e) & Bellman - Ford \\ \mathrm{O}(v^3) & Floyd - Warshall \\ \mathrm{O}(v^3) & Dantzig \end{cases} \quad (3)$$

5.4. Análisis de Correctitud

Para hablar de correctitud hay que explicar por qué este modelo -al que llamamos *grafo estado*- en conjunto con la implementación de los algoritmos propuestos resuelven el problema inicial. Una vez que se obtenga la cantidad de litros necesarios para poder partir de una ciudad a la otra, se tienen 2 opciones:

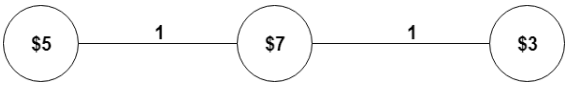
- Avanzar a la próxima ciudad y restarle al tanque los litros utilizados, o
- Cargar otro litro de nafta (si no se llegó al máximo de capacidad) y volver a tomar alguna de las 2 opciones posibles.

Cada decisión tomada implica elegir una opción y descartar otros subproblemas. Por ejemplo, si se elige avanzar a la próxima ciudad una vez que se tenga la mínima cantidad de nafta para llegar, si en esa ciudad la nafta está más cara y el tanque se encuentra vacío al haber llegado, estamos en un caso donde la mejor opción era seguir cargando nafta antes de partir por ser más barata, en vez de pagar un mayor costo para seguir avanzando.

Al no haber forma de anticipar cuál decisión es la correcta en cada paso, habrá que contemplarlas todas. Para no caer en aplicar Fuerza Bruta o alguna otra estrategia similar, aquí es donde el grafo estado brinda una solución para encarar dicho problema: al generar un vértice estado para cada posible valor del tanque de combustible del auto y el costo de cargar un litro más se reflejará en el eje orientado que conecte un vértice estado con el siguiente. Luego reconectar los posibles vértices estado de cada ciudad con el de la próxima ciudad llegando al que represente el gasto en combustible que proporcionaba el eje que las conectaba en el grafo inicial.

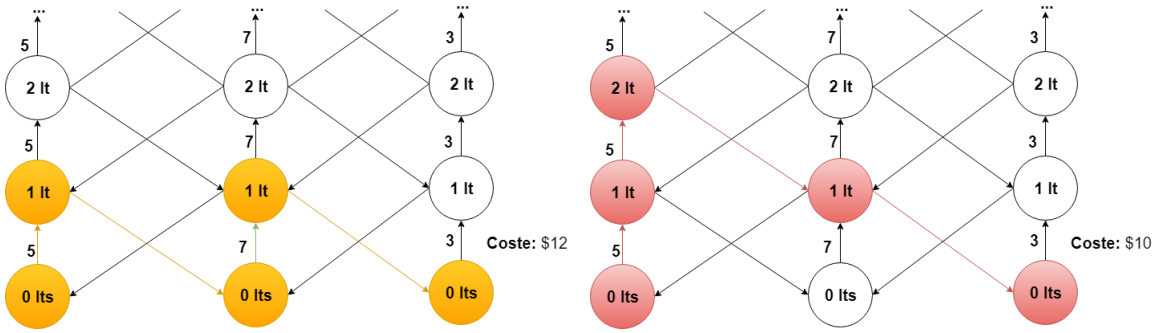
Como el objetivo no es gastar la menor cantidad de nafta, sino gastar la menor cantidad de plata posible, los ejes que conecten distintas ciudades no poseerán ningún costo asociado, o en otras palabras, valdrán 0.

Analizaremos a continuación un pequeño ejemplo que mostrará cómo esta variación al grafo original proporciona los caminos mínimos.



En un ejemplo tan sencillo está claro que la mejor forma de ir de la ciudad con costo $5 a la de costo $3 implica cargar 2 litros en la primera ciudad en vez de cargar 1 en cada ciudad hasta llegar a la de costo $3, ya que tendríamos como gasto $12 en vez de $10.

Los algoritmos de camino mínimo no sabrían cómo resolver esto de manera eficiente sin aplicar modificaciones ya que, primero, se basan en los pesos de los ejes, los cuales suman 2, y después, porque no está modelada la posibilidad de cargar nafta varias veces por ciudad, o sea, cargar varios litros. En cambio al utilizar el grafo estado que se genera en base a éste, se pueden obtener dichos caminos (entre otros):



Ahora sí se puede observar cómo se pueden generar ambas instancias descriptas anteriormente donde la mejor opción era cargar los 2 litros en la primera ciudad y luego cómo aplicar cualquier algoritmo de camino mínimo se resuelve aplicándolo al grafo estado.

Si bien es un ejemplo muy pequeño, lo mismo sirve para cualquier tipo de grafo, ya que el grafo estado modela todas las posibles acciones a realizar y sus correspondientes conexiones entre ciudades y los ejes que tienen peso simbolizan el gasto de nafta para moverse y los de costo 0 simbolizan el gasto de nafta para alcanzar una ciudad desde otra.

Una vez transformado el grafo y aplicado el algoritmo elegido, basta buscar los caminos mínimos para todos los vértices estado de cada par de ciudades y fijarse el costo mínimo entre los 61 vértices estado que representan la ciudad de destino, ya que nuestra matriz resultante está armada para todos los vértices estado y en realidad solo nos interesa el mínimo de todos los que pertenecen a la misma ciudad.

6. Experimentación

6.1. Metodología

Se realizaron distintas experimentaciones para dar soporte a las hipótesis expuestas respecto de las complejidades de los algoritmos presentados anteriormente. Para reunir una cantidad significativa de datos se desarrolló un generador de grafos con las siguientes particularidades:

- Se trabaja con grafos conexos y se generan para cada valor posible de cantidad de nodos, todas las combinaciones posibles en cantidad de aristas: desde un *árbol* hasta un *completo*. Según el algoritmo, la cantidad de nodos será limitada a la capacidad de procesamiento y tiempo de ejecución.
- Se genera un árbol de v nodos. A cada uno de los $v - 1$ ejes se les asigna un peso al azar utilizando la función *rand* de C++.
- Los $e - v + 1$ ejes restantes se crean eligiendo al azar dos nodos del grafo y un peso, utilizando *rand* y evitando que haya valores repetidos.
- El peso de los ejes se asignó de la siguiente manera:
  - En *Hiperconectados* se eligió un valor entre 1 y e para evitar casos triviales (peso 0) y tener una distribución aceptable entre los valores posibles y la cantidad de ejes.
  - Para *Hiperauditados* el rango de los pesos estuvo entre 30 y 60 para forzar la necesidad de cargar el tanque de combustible al moverse entre varias ciudades. Se evitaron pesos mayores a 60 ya que por las restricciones del problema sería un eje imposible de transitar.
- Los precios del litro de nafta por ciudad están entre 1 y v. Nuevamente se evitó el 0 como precio posible por considerarlo un caso trivial.
- Se cambió el valor del *seed* con cada grafo creado para evitar grafos repetidos.
- Para comparar **A*** y **Dijkstra (con cola de prioridad)** los grafos se generaron con un procedimiento diferente que es explicado en la sección correspondiente.

Los archivos fueron compilados utilizando *G++* como compilador y utilizando los siguientes flags: *-std=c++11 -O3 -Ofast*. Asimismo los programas se corrieron en una computadora que cuenta con *Ubuntu Linux 16.04 LTS* corriendo sobre un *Intel i7 con 16 GB de memoria ram*.

Para medir el tiempo de ejecución se utilizó la función *chrono* de C++. Específicamente se tomó el tiempo entre la llamada a la función y el momento en que retorna el resultado. Cada uno de los problemas aquí resueltos fueron ejecutados para cada uno de los grafos generados una cantidad de veces que iba entre las 100 y las 1000 promediando los tiempos obtenidos con el fin de reducir el ruido en cada muestra.

No siempre resultó posible probar todas las combinaciones de valores de nodos y aristas por limitaciones de tiempo que presentan las ejecuciones de los algoritmos. El límite en la cantidad de nodos para los grafos generados irá variando según el algoritmo y la hipótesis. Para facilitar las pruebas y obtener resultados que varíen la experimentación, se agruparon los grafos dependiendo de lo que se intentara probar de la siguiente manera:

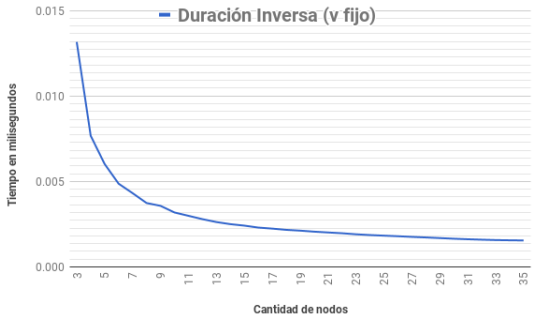
- **Árboles**: Grafos conexos cuya cantidad de aristas es equivalente a $nodos - 1$.
- **Completos**: Grafos conexos cuya cantidad de aristas es equivalente a $\frac{nodos*(nodos-1)}{2}$.
- **Todos**: Grafos conexos cuya cantidad de aristas varía entre $nodos - 1$ y $\frac{nodos*(nodos-1)}{2}$.

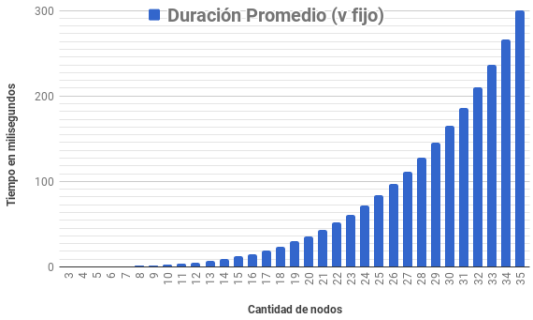
6.2. Hiperconectados

6.2.1. Complejidad

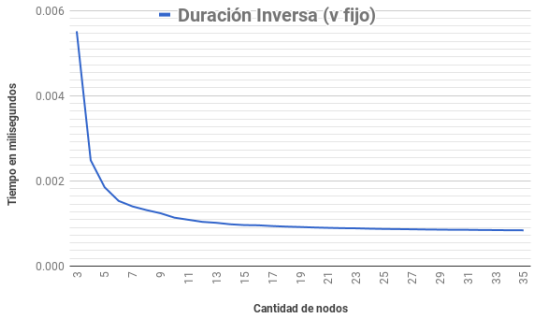
El análisis teórico de la complejidad de los algoritmos para Prim, Kruskal y Kruskal con Path Compression les otorga una cota de O(e*log(v)). A continuación se encuentran los gráficos que dan soporte empírico a las hipótesis expuestas. Para obtenerlos se construye a modo de función inversa aquella que se obtiene de dividir el tiempo de ejecución del algoritmo para una cantidad v de nodos (resultado función $g(x)$) por el resultado del cálculo $e^2\log(v)$ (promediando los valores para todos los grafos con v nodos). Como se puede observar al graficar la función $f(x) = g(x)/(e^2 log(x))$ se obtuvieron los resultados esperados de acuerdo a la complejidad que fue calculada. Nótese como luego de los primeros valores obtenidos, los gráficos de la función $g(x)$ para cada subalgoritmo muestran que los valores graficados tienden a una curva de apariencia monótona y constante.

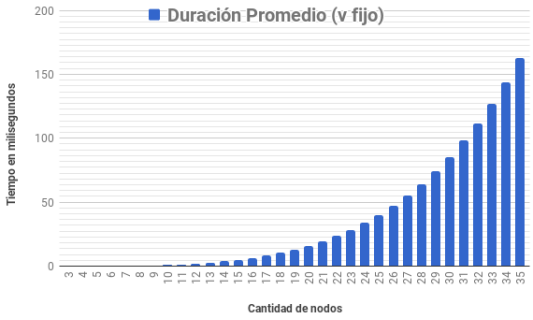
- Prim:



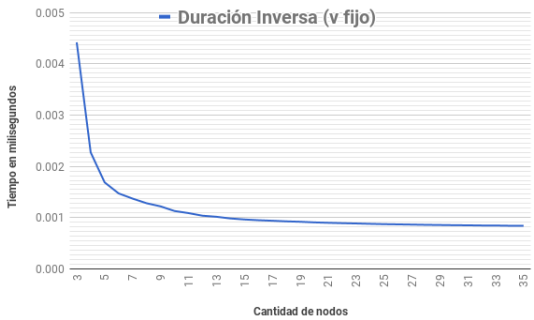


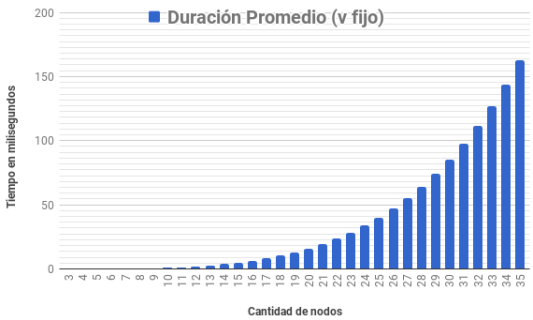
- Kruskal:





- Kruskal con Path Compression:



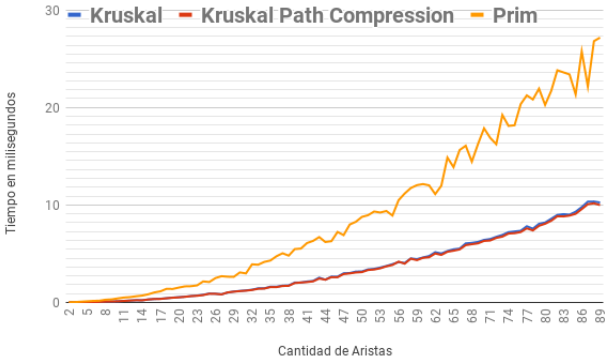


A partir de cierto valor comienza a verse la relación entre la cota de complejidad teórica y el tiempo de ejecución promedio que hace que el cociente entre ambas curvas tenga por producto la función inversa previamente detallada. A medida que se incrementa el tamaño de la muestra, en los tiempos de ejecución empieza a pesar la relación entre las variables y se observa que la curva en el gráfico se estabiliza. En consecuencia para los casos observados la cota teórica parece funcionar bien con estos algoritmos.

6.2.2. Análisis

Para cada grafo generado se obtuvo un tiempo promediado de ejecución según el subalgoritmo que se eligiera (cada grafo fue ejecutado 500 veces por cada variante). Luego los resultados se agruparon según subalgoritmo, subconjunto de grafos y promedio en base a la cantidad de vértices y luego en base a los ejes. Si bien asintóticamente la complejidad no varía, al comparar los tiempos de ejecución de cada subalgoritmo según el tipo de grafo siendo analizado (árboles, completos y todos) mostrará qué curvas son las menos pronunciadas.

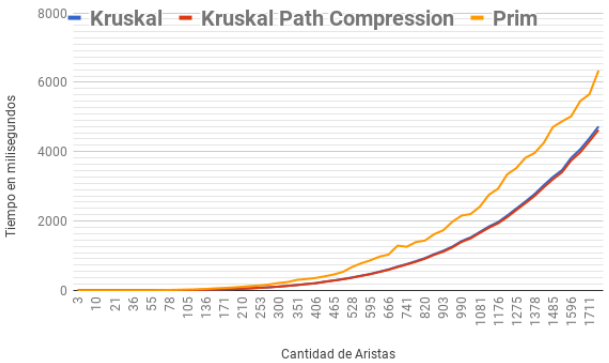
Para Hiperconectados los mejores casos (menores tiempos de ejecución) están dados por los grafos que son un árbol ya que en sí mismos constituyen un AGM. Entonces no existe arista que pertenezca al grafo y no a su AGM (ya que el camino posible entre todo par de vértices es único). Al tener la menor cantidad de ejes posibles no solo el ciclo principal que itera sobre cada arista tiene menor cantidad de iteraciones sino que también el ordenamiento es menos costoso. A continuación el gráfico con los tiempos de ejecución para árboles:



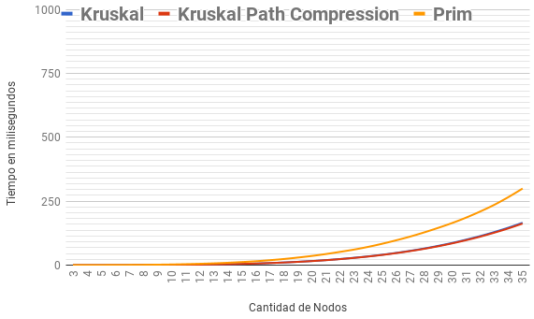
Si bien la diferencia es ínfima, ambos implementaciones del algoritmo de Kruskal obtienen tiempos similares que prácticamente superponen sus curvas en el gráfico. Esto se debe a que la parte más costosa de ambas implementaciones está en el ordenamiento de la lista de ejes de menor a mayor. En nuestras implementaciones no encontramos que la optimización por *Path Compression* mejore sustancialmente los valores, esto es esperable si se considera que en ambas variantes el resto de las operaciones son comunes y de mayor complejidad.

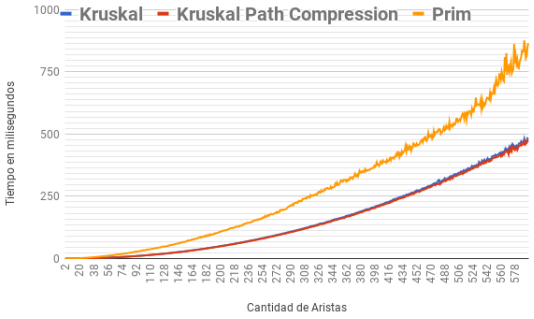
Cabe destacar que los tres subalgoritmos analizados comparten la misma cota asintótica y sin embargo *Prim* es el que más tiempo toma en ejecutar. Esto se debe a que realiza una cantidad mayor de cálculos que dotan a su complejidad de una constante mayor a la de los otros dos. En cada iteración del ciclo principal de dicho subalgoritmo, para obtener eficientemente el mínimo eje que conforme el AGM, se itera tantas veces como sea necesario extrayendo elementos de la cola de prioridad hasta dar con aquel que no genere ciclos.

Continuamos con las ejecuciones realizadas sobre los grafos completos donde hay $\frac{nodos*(nodos-1)}{2}$ cantidad de ejes. Al incrementar el número de aristas y tener potencialmente muchas mas combinaciones de AGMs, los tiempos crecen respecto del caso de los árboles pero las curvas respetan las cotas propuestas y vuelve a destacarse el comportamiento anterior de *Prim* y de las implementaciones de *Kruskal*. Se obtuvieron los siguientes gráficos para estos casos:

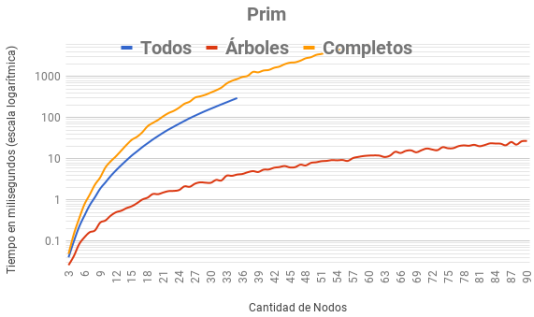


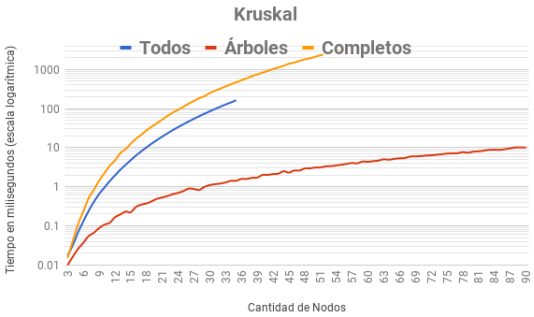
Finalmente se presentan los gráficos que corresponden a los tiempos de ejecución para el conjunto de grafos que consideramos en la categoría *todos*. Para las dos categorías previamente presentadas, cada grafo generado era único por la cantidad de nodos. En este caso se generaron para v cantidad de nodos, un grafo por cada combinación de v nodos y e aristas desde $e = v - 1$ hasta un grafo *completo*. Los resultados observados son consecuentes con los casos previos: Prim obtuvo los peores tiempos de ejecución mientras que no hubo una diferencia notable entre las implementaciones de Kruskal. Se presentan dos gráficos: uno correspondiente a la cantidad de nodos y otro a la cantidad de aristas, dado que la cota de complejidad depende de ambas variables -v y e-. En el primero se promedian los valores obtenidos agrupando los grafos por cantidad de nodos y en el segundo se procede de la misma manera con la cantidad de aristas.

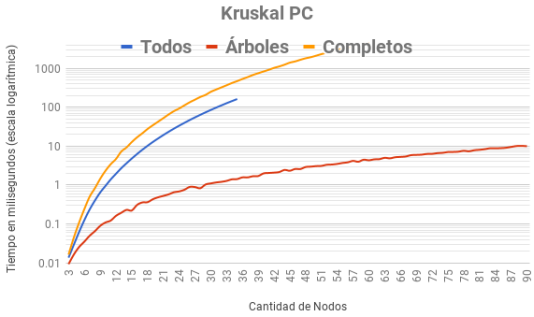




A continuación se incluyen los gráficos agrupados por subalgoritmo y comparando las curvas que corresponden a cada categoría de los grafos utilizados como entrada. Los mismos muestran una curva mas corta en todos los casos que corresponde a la ejecución de la categoría *todos*. Como esta tiene la mayor cantidad de grafos por cantidad de nodos los tiempos de ejecución se volvían impracticables, por lo que no era posible extenderla a la misma cantidad de nodos que las demás -para conseguir ejecutar todos los grafos de 41 nodos era necesario hacerlo 820000 veces por cada algoritmo lo cual volvía la experimentación inviable-.







Los tiempos de los grafos de tipo *árboles* son claramente menores que los de los otros dos tipos, como se explicó anteriormente, al tener la menor cantidad de ejes se realizan menos operaciones. Lo contrario sucede con los grafos completos. El promedio de todos los grafos intermedios se ubica entre las otras dos categorías como era de esperar, sin embargo observamos que está ubicada más cerca de la categoría de completos; podemos interpretar de este hecho que los grafos con mayor cantidad de ejes tienen mayor peso en el promedio de tiempos.

6.3. Hiperauditados

6.3.1. Complejidad

Cada grafo generado obtuvo un tiempo promediado de ejecución según el subalgoritmo que se eligiera (cada grafo fue ejecutado 50 veces por variante). Luego los resultados se agruparon según subalgoritmo, subconjunto de grafos y promedio en base a la cantidad de vértices y luego en base a los ejes.

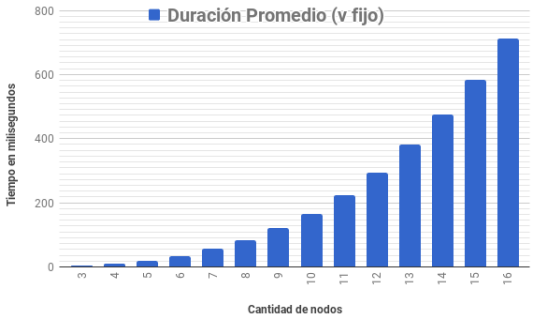
De la misma manera que se trabajó con los subalgoritmos de *Hiperconectados*, todo el análisis está hecho en base a las complejidades de los 6 subalgoritmos pedidos por la cátedra (Dijkstra, Dijkstra con Priority Queue, A*, Bellman-Ford, Floyd-Warshall y Dantzig). A continuación se exhiben las experimentaciones con las que buscamos respaldar las hipótesis respecto de las cotas de complejidad teórica. Para cada uno de ellos se exhibe el promedio de tiempos obtenido para una cantidad fija de aristas (o nodos según corresponda) y luego el resultado de la función inversa.

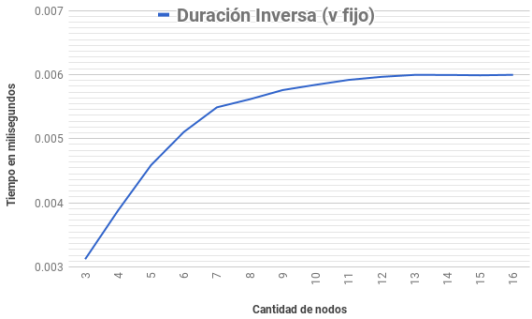
El análisis teórico de la complejidad de los subalgoritmos les otorga una cota distinta según el caso. Para comparar la complejidad teórica con los resultados obtenidos se construye a modo de función inversa aquella que se obtiene de dividir el tiempo de ejecución del algoritmo para una cantidad de v nodos (resultado función $g(x)$) por el resultado del cálculo de la cota de complejidad (promediando los valores para todos los grafos con v nodos). Al graficar la función $f(x) = g(x)/O(x)$ se obtuvieron los resultados esperados de acuerdo a la complejidad que fue calculada para los casos de *Dijkstra*, *Dijkstra PQ*, *Bellman-Ford* y *Dantzig*. Nótese como luego de los primeros valores obtenidos, los gráficos de la función $g(x)$ para cada uno de estos subalgoritmos muestran que los valores graficados tienden a distintos grados de estabilización alrededor de una constante.

Cabe destacar que las curvas obtenidas para el resto de los subalgoritmos no muestran una curva con una tendencia similar y sin embargo no contradicen las hipótesis presentadas. La capacidad computacional y temporal para extender la experimentación a una cantidad mayor de nodos limita la observación al recorte presentado. Es el caso como el de *Floyd-Warshal*, donde el gráfico de barras muestra que los tiempos de ejecución son mayores y dificultan la evaluación de grafos aún mas grandes.

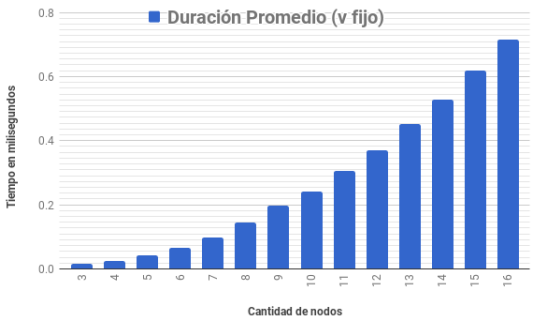
Si bien solo se puede observar una tendencia del comportamiento que respalda la hipótesis, tamaños de entrada más grandes colaborarían como refuerzo. De la misma manera que lo acontecido con la experimentación sobre *Hiperconectados*, los tiempos que suponen extender la cantidad de nodos y aristas de los grafos utilizados se escapan al marco propuesto.

- Dijkstra y función inversa: $f(x) = g(x)/(v^3 + e)$

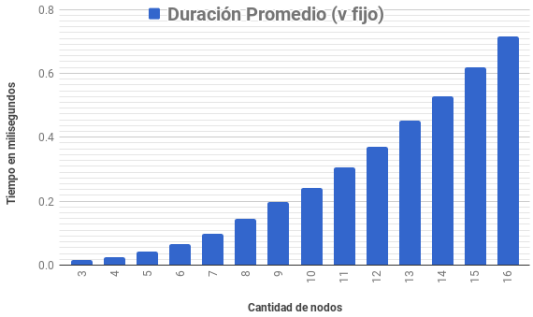




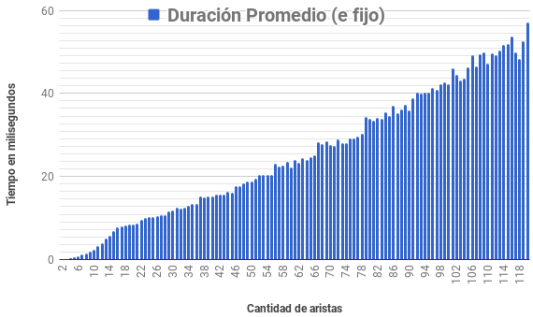
- Dijkstra con Priority Queue y función inversa: $f(x) = g(x)/(v * e * log(e))$

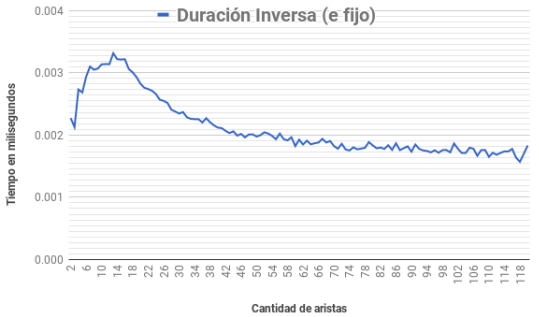


- A* y función inversa: $f(x) = g(x)/(v * e * log(e))$

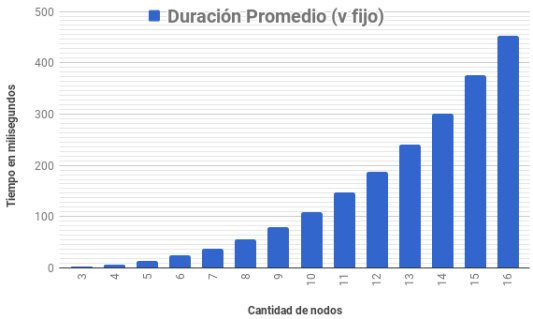


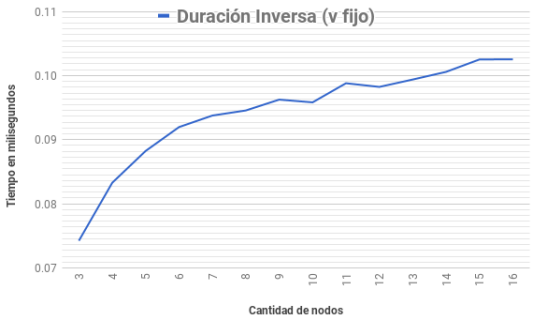
- Bellman-Ford y función inversa: $f(x) = g(x)/(v^2 * e)$



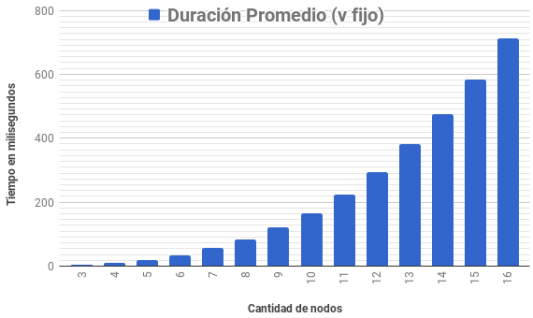


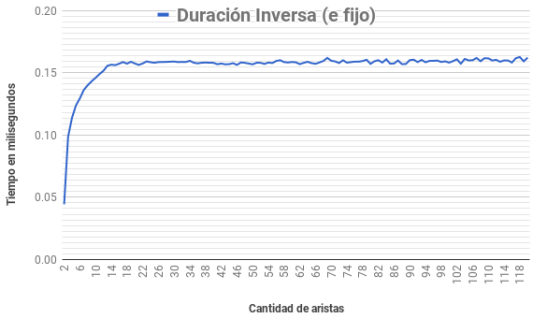
- Floyd-Warshall y función inversa: $f(x) = g(x)/(v^3)$





- Dantzig y función inversa: $f(x) = g(x)/(v^3)$





Como en esta parte de la experimentación no hay hipótesis extras sobre los grafos utilizados, A* se comporta de la misma manera que Dijkstra PQ ya que sus implementaciones son idénticas exceptuando el uso de la heurística.

6.3.2. Análisis

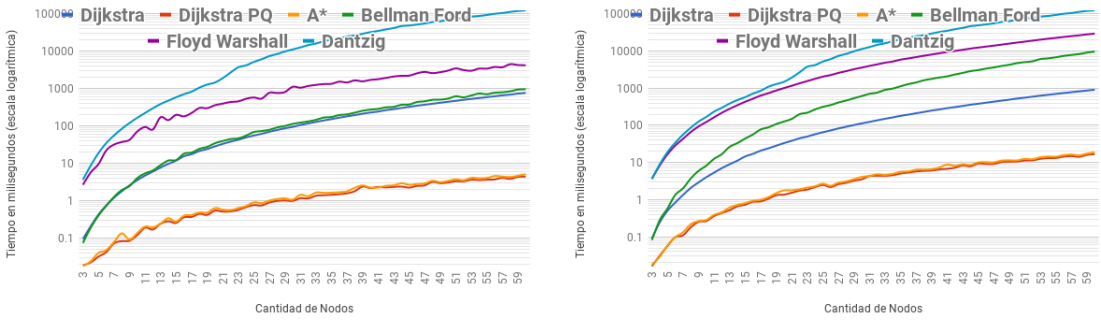
Al igual que para *Hiperconectados* se categorizaron y agruparon los grafos generados en tres: **Árboles**, **Completos** y **Todos**. La única diferencia con los grafos previamente generados es que se restringió el rango de valores para los pesos de los ejes. De manera de tener cierto control en la carga de combustible durante el recorrido, al peso del eje se le restringió su valor, aunque aún con aleatoriedad, en el rango: *[30,60]*. Considerando que el tamaño del tanque de combustible es de 60 litros, se hace necesario recurrir a cargar combustible para recorrer el grafo aunque no necesariamente en cada ciudad. Por otro lado, tener grafos donde los ejes tengan pesos mayores a 60 los haría virtualmente inconexos (sería equivalente a que el eje no existiera por las limitaciones del tanque).

Para cada grafo se reutilizarán los mismos experimentos que en *Hiperconectados* con la intención de probar las hipótesis de los valores de complejidad expresados teóricamente. Se comparará el tiempo de ejecución en base a los vértices y en base a los ejes, repitiendo para cada grafo evaluado 50 veces el experimento para calcular un promedio de tiempos y minimizar las posibles variaciones.

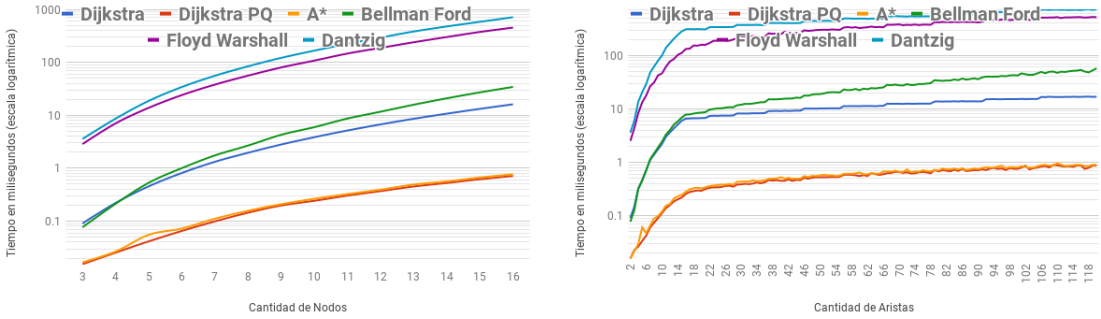
Hay que considerar que los grafos generados y utilizados para nuestras experimentaciones lucen de menor tamaño que en el caso del algoritmo anterior. Para resolver *Hiperauditados* el grafo utilizado como entrada es reconvertido en un grafo de mayor tamaño lo cual aumenta considerablemente los tiempos de ejecución y el consumo de recursos. Donde antes había sido posible explorar hasta grafos completos de 40 nodos, aquí la limitación será de 16 nodos. Y no solo el tamaño del grafo que se utiliza para resolver el problema es mayor, los subalgoritmos a su vez tienen cotas de complejidad también mayores.

Veamos a continuación los tiempos de los distintos subalgoritmos para cada tipo de grafos:

- Árboles (izquierda) y Completos (derecha)



- Todos (en función de la cantidad de nodos a la izquierda, y de aristas a la derecha):



Podemos observar en los experimentos realizados que el órden de performance entre los subalgoritmos se mantiene independientemente del tipo de grafo para estas cantidades de nodos. A su vez podemos notar que los tiempos de ejecución utilizando Dantzig no varían significativamente entre árboles y grafos completos, mientras que Floyd-Warshall empeora notablemente para completos. Otra resultado a destacar es que los algoritmos de Bellman-Ford y Dijkstra tienen un desempeño similar para árboles, pero para completos Bellman-Ford requiere mucho más tiempo incluso acercándose a Dantzig y Floyd-Warshall. La explicación para estos comportamientos

se puede encontrar analizando más minuciosamente la complejidad en los casos extremos: árboles y grafos completos. Las cotas de complejidad cambian de la siguiente manera:

- Para los árboles, como e = v - 1, tenemos que O(e) = O(v), entonces:

$$O(Hiperauditados) = \begin{cases} \mathrm{O}(v^4) & , Dijkstra \\ \mathrm{O}(v^2 * log(v)) & , DijkstraPQ \\ \mathrm{O}(v^2 * log(v)) & , A* \\ \mathrm{O}(v^3) & , Bellman - Ford \\ \mathrm{O}(v^3) & , Floyd - Warshall \\ \mathrm{O}(v^3) & , Dantzig \end{cases} \tag{4}$$

  Por lo que ordenándolos por cota quedan:

$$(DijkstraPQ, A*) < (Bellman - Ford, Floyd - Warshall, Dantzig) < Dijkstra$$

- Y para los completos, como $e = \frac{v(v-1)}{2}$, tenemos que $O(e) = O(v^2)$, entonces:

$$O(Hiperauditados) = \begin{cases} \mathrm{O}(v^5) & , Dijkstra \\ \mathrm{O}(v^3 * log(v)) & , DijkstraPQ \\ \mathrm{O}(v^3 * log(v)) & , A* \\ \mathrm{O}(v^4) & , Bellman - Ford \\ \mathrm{O}(v^3) & , Floyd - Warshall \\ \mathrm{O}(v^3) & , Dantzig \end{cases} \tag{5}$$

  Ordenándolos por cota quedan:

$$(Floyd - Warshall, Dantzig) < (DijkstraPQ, A*) < Bellman - Ford < Dijkstra$$

La razón por la que este ordenamiento por cota no se observa en los experimentos es que la cota no es lo suficientemente ajustada para algunos de los algoritmos. Si vemos en mayor detalle las implementaciones de los subalgoritmos tenemos los siguientes órdenes de complejidad (considerando $e = v^2$):

$$O(Hiperauditados) = \begin{cases} 3\ O(v) + 2\ O(v^2) & , Dijkstra \\ \mathrm{O}(v) + 2\ O(v^2 * log(v)) + 2\ O(v * d(v) * log(v)) & , DijkstraPQ \\ \mathrm{O}(v) + 2\ O(v^2 * log(v)) + 2\ O(v^2 * log(v)) & , A* \\ \mathrm{O}(v) + O(v^3) & , Bellman - Ford \\ \mathrm{O}(v^2) + O(v^2) + O(v) + O(v^3) & , Floyd - Warshall \\ \mathrm{O}(v^2) + O(v^2) + O(v^3) + O(v^3) & , Dantzig \end{cases} \tag{6}$$

Ordenándolos nuevamente por cota obtenemos:

$$(DijkstraPQ, A*) < Dijkstra < Bellman - Ford < Floyd - Warshall < Dantzig$$

Este orden por cota es más cercano al orden de performance observado en los experimentos.

6.3.3. Performance y mejoras sobre A* respecto de DPQ

A lo largo de los experimentos anteriores en *Hiperauditados*, puede verse que el desempeño de la implementación de A* no mejora notablemente con respecto a la de *DPQ* (Dijkstra con Priority Queue). Sin embargo, A* tiene un desempeño superior cuando se tiene una heurística para poder estimar la distancia hacia un objetivo -en particular cuando los grafos respetan la distancia euclídea se puede usar la distancia directa entre dos nodos como estimación-. En los grafos generados para los problemas anteriores no se dispone de la información necesaria para hacer esta estimación. Además la transformación del grafo original al *grafo de estados* rompería la propiedad euclídea si la hubiese. Debido a esto, para poder comparar el desempeño de este algoritmo y demostrar la hipótesis de su superioridad con respecto a Dijkstra PQ experimentalmente, se tomaron las siguientes decisiones:

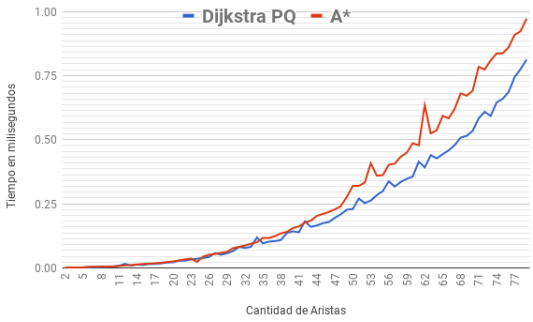
- Dado que $A*$ encuentra el camino mínimo entre dos nodos y que su implementación es casi idéntica a la de *DPQ*, se decidió comparar únicamente ambos algoritmos, modificando el segundo para detenerlo cuando llega al objetivo.
- Se generaron los grafos con las siguientes restricciones:
  - El precio de la nafta es el mismo en todas las ciudades.
  - El grafo es euclídeo: dados los ejes (a,c), (a,b) y (b,c) cualquiera, siempre pasa que $d(a,c) \leq d(a,b) + d(b,c)$

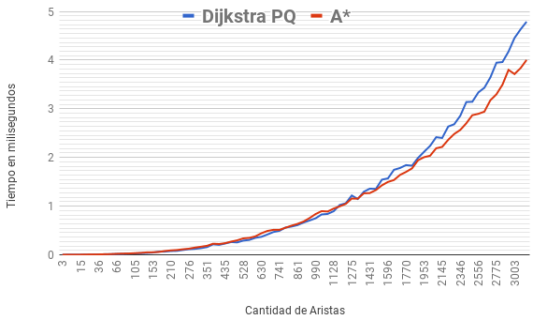
Con estas consideraciones, el problema para estas instancias de *Hiperauditados* se reduce a encontrar el camino mínimo entre dos ciudades cualesquiera. La estimación utilizada para la heurística de A* es la distancia al objetivo, este dato es ignorado por *Dijkstra PQ* en sus cálculos. Concretamente la diferencia entre ambas implementaciones es que para hacer las inserciones en la cola de prioridad A* utiliza esta heurística. Las pruebas se realizaron midiendo el tiempo como fue detallado en el inicio de esta sección.

Si bien se exponen todos los experimentos llevados a cabo, interesa destacar que en el caso de los árboles la hipótesis no se comprueba. El cálculo del camino mínimo mediante el uso de cola de prioridad será siempre superior al de una heurística cuando no existen combinaciones alternativas y el camino será uno solo por la estructura que impone el grafo. En cambio, hay que destacar el último de los gráficos que considera para cada cantidad de nodos el promedio del tiempo de ejecución para los grafos con todas las combinaciones de aristas posibles. Es ahí donde queda claro el beneficio del uso que hace A* de la heurística en este tipo de escenarios.

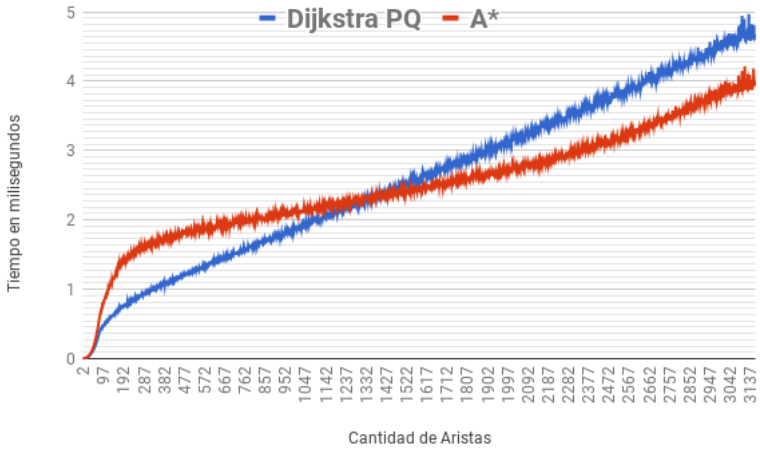
Veamos a continuación los tiempos de los algoritmos para cada tipo de grafos:

- Árboles (izq) y Completos (der)





- Todos


Dijkstra PQ
A*
Tiempo en milisegundos
Cantidad de Aristas
0
1
2
3
4
5
2
97
192
287
382
477
572
667
762
857
952
1047
1142
1237
1332
1427
1522
1617
1712
1807
1902
1997
2092
2187
2282
2377
2472
2567
2662
2757
2852
2947
3042
3137

7. Conclusiones y trabajo futuro

A lo largo de los experimentos realizados se trabajó extensivamente con algoritmos para grafos, buscando resolver distintos problemas que requirieron apoyarse fuertemente en las propiedades básicas de los grafos y en el uso de algoritmos clásicos. Para cada uno de estos problemas se pretendió corroborar empíricamente las complejidades calculadas para cada variante de los algoritmos. Asimismo, se midió el desempeño de los mismos frente a distintos tipos de grafos para poder compararlos entre sí.

Inicialmente planteamos que el problema *Hiperconectados* podría resolverse viendo todos los posibles AGMs para cada grafo de entrada si fuese posible computacionalmente pero, como sabemos, no se conocen algoritmos polinomiales para semejante tarea. La estrategia propuesta permitió resolver el problema en tiempo polinomial re-utilizando algoritmos conocidos como subrutinas: *Prim y Kruskal*. Para esto fue clave entender las propiedades básicas de los grafos y, en particular, de los AGMs. Una vez planteada la forma de resolución, su correctitud y una hipótesis acerca de su complejidad, pudimos realizar experimentos para corroborar o refutar nuestras expectativas -al mismo tiempo que se plantearon nuevos interrogantes y dificultades-.

Algunos resultados que podemos destacar de este problema incluyen la necesidad de comprobar la performance de un algoritmo más allá de su cota de complejidad y el inconveniente de trabajar con grafos cada vez más grandes. En cuanto a la primera observación, en los primeros experimentos vimos que si bien las tres variantes del algoritmo compartían una determinada cota, había uno que tenía una constante más alta y por lo tanto obtenía peores tiempos de ejecución. Por otro lado, no pudimos comprobar el impacto de determinadas optimizaciones como el agregado del *Path Compression* en *Kruskal*. Respecto del tamaño de los grafos observamos que determinados experimentos son inviables si no se hace un recorte del dominio del problema. Sin embargo, creemos que los experimentos fueron suficientes para reconocer la superioridad de *Kruskal* en nuestras implementaciones. Esta superioridad está en parte dada por la transformación que requiere *Prim* respecto de la representación del grafo de entrada (de lista de incidencias a lista de adyacencias). Para un trabajo futuro habrá que comparar Prim y Kruskal en un contexto más neutral para ambos, es de esperarse que uno sea mejor que otro dependiendo de la densidad de los grafos de entrada.

En *Hiperauditados* la dificultad principal estuvo centrada en la búsqueda de un modelo que pudiese representar el problema de una manera que facilitase su resolución. Una vez encontrado este modelo, obtener una solución óptima fue tan sencillo como utilizar un algoritmo clásico de camino mínimo. Para la experimentación repetimos el esquema de *Hiperconectados*: buscamos ver reflejada la complejidad calculada para cada uno de los subalgoritmos (*Dijkstra, Dijkstra PQ, A*, Bellman-Ford, Floyd-Warshall, Dantiz*) y luego comparar los tiempos de ejecución de cada variante entre sí para sacar conclusiones.

Dada la transformación del grafo de entrada a un *grafo de estados* para poder modelar *Hiperauditados* como un problema de camino mínimo, el tamaño de los grafos creció en la cantidad de nodos y ejes de manera que se dificultó la ejecución del algoritmo para grafos grandes. El tener esta limitación en el tamaño del grafo de entrada no permitió hacer análisis conclusivos acerca de la complejidad de todos los algoritmos, sin embargo ninguno de los experimentos refutó las hipótesis y en los subalgoritmos de *Dijkstra, Dijkstra PQ, A* y Dantzig* se obtuvieron resultados fieles a las cotas calculadas -dentro de los límites analizados-.

A la hora de compararlos entre sí en sus tiempos de ejecución pudimos observar la influencia de las constantes implícitas en la complejidad de cada algoritmo. Para poder encontrar el orden esperado en los tiempos de cada subalgoritmo fue necesario hacer un análisis más fino de la complejidad.

Para finalizar, experimentamos con un caso especial de grafos de entrada para poder comparar Dijkstra PQ con A*. Los resultados obtenidos confirman la importancia de usar algoritmos especializados cuando se tiene información adicional respecto de los datos de entrada. En este caso saber que los grafos respetan la distancia euclídea permite el uso de algoritmos como A* que en algunos casos pueden tener un desempeño mucho mejor.

En conclusión, una vez terminado este trabajo y los experimentos tenemos certeza de haber descubierto la importancia de las técnicas utilizadas y los conceptos explorados para nuestra formación en las ciencias de la computación. Si bien una etapa del trabajo estuvo basada en escenarios particulares, todos los resultados obtenidos impartieron conocimiento sobre el trabajo con grafos y distintas técnicas algorítmicas, y sobre cómo reflexionar para adaptar problemas a otros más sencillos. Esperamos poder investigar más en profundidad estos aspectos y conocer otros de igual importancia en el contexto de la materia.