DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Yéndose por las ramas

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2018

Grupo Estrellitas

Integrante	LU	Correo electrónico
Salinas, Pablo	456/10	salinas.pablom@gmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com
Berríos Verboven, Nicolás	46/12	nbverboven@gmail.com
Hofmann, Federico	745/14	federico2102@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo abordaremos dos problemas relacionados al planeamiento urbano, que denominaremos **Hiperconectados** e **Hiperauditados** y modelaremos utilizando grafos. A continuación, definiremos ambos problemas, junto con los modelos utilizados para resolverlos:

1.1. Hiperconectados

Este problema puede explicarse de la siguiente manera: se cuenta con un conjunto de ciudades, para las cuales se quiere crear una red de conexiones de Internet de manera tal que todos los pares de ciudades estén conectados por un camino de la red. Además de las ciudades, se cuenta con una lista de conexiones potenciales entre ellas. Cada conexión potencial con tres atributos: las dos ciudades que estaría conectando, junto con el costo de instalar efectivamente dicha conexión. El primer objetivo es crear la red de manera tal que el costo total de la red (definido como la suma de los costos de las conexiones efectivamente utilizadas para crear la red) sea mínimo. Modelaremos este problema con un grafo pesado, donde los nodos representan las ciudades a conectar y las aristas representan las conexiones potenciales, siendo el peso de cada arista el costo de establecer dicha conexión. Con el problema modelado de esta manera, el objetivo es exactamente hallar un Árbol Generador Mínimo (AGM) del grafo original. Como además se sabe que no hay una única manera de obtener un AGM de un grafo, el segundo objetivo es decidir, para cada arista que representa una conexión potencial, si pertenece a:

- Todos los posibles AGM del grafo.
- Algún posible AGM del grafo.
- Ningún posible AGM del grafo.

Este segundo problema viene motivado por lo siguiente: si bien el costo de cada posible enlace es un factor importante para definir si debe ser incluido en la red resultante, no es el único. Por ejemplo, una conexión potencial de costo muy bajo podría no ser deseable si se supiera que debe instalarse en una región de difícil acceso para su mantenimiento posterior por hallarse en una zona sísmica. Entonces, si se sabe que una conexión poco deseable por estos factores externos no pertenece a todos los AGM posibles, se puede elegir un AGM que no la contenga para evitar los problemas adicionales que traería aparejados.

Para resolver este problema, implementaremos tres soluciones distintas, basadas en los algoritmos de Prim, Kruskal y Kruskal con *Path Compression* para encontrar Árboles Generadores Mínimos. En las siguientes secciones de este informe exhibiremos nuestras soluciones para este problema, junto con sus detalles de implementación y cálculos de complejidad temporal.

1.2. Hiperauditados

Este problema puede explicarse de la siguiente manera: una vez establecida la red de costo mínimo, el objetivo es asegurarse periodicamente de que todas las conexiones están funcionando correctamente, para lo cual se debe visitar físicamente todas las ciudades, en lugar de usar herramientas remotas de diagnóstico como *ping*. Para ello, el objetivo es hallar la mejor forma de trasladarse en auto entre dos ciudades, minimizando la plata gastada en combustible. Con esto, se puede definir formalmente el problema: se cuenta con un conjunto de n ciudades y m rutas entre ellas. Cada ruta entre dos ciudades tiene un valor l_i que indica la cantidad de litros de nafta que necesita el vehículo para recorrerla, y cada ciudad tiene asociado un valor c_i que indica el precio de un litro de nafta en dicha ciudad, y lo que se busca saber es la cantidad mínima de plata necesaria para moverse entre un par de ciudades dadas, sabiendo que la capacidad del tanque de nafta del vehículo es de 60 litros. Para modelar este problema, nos valdremos nuevamente de grafos pesados; el modelado del problema utilizando grafos será explicado en la próxima sección de este informe. Al igual que para el problema anterior, implementaremos varias soluciones distintas, basadas en los siguientes algoritmos:

- Algoritmo de Dijkstra.
- Algoritmo de Dijkstra con *priority queue*.
- Algoritmo A* (Solo para grafos que cumplan la desigualdad triangular).
- Algoritmo de Bellman-Ford.
- Algoritmo de Floyd-Warshall.
- Algoritmo de Dantzig.

En las siguientes secciones de este informe exhibiremos, para cada una de nuestras soluciones a este problema, sus detalles de implementación, junto con la complejidad temporal de cada uno.

2. Desarrollo

En esta sección se explicarán los algoritmos diseñados para resolver los dos problemas enunciados en la sección previa.

2.1. Hiperconectados

En este problema, nuestro objetivo es el siguiente: para cada arista e del grafo, queremos saber si e pertenece a todos, alguno o ninguno de los posibles árboles generadores mínimos del grafo. Como hemos mencionado en la sección previa, nuestra solución a este problema se basará en los algoritmos de Prim y Kruskal (junto con su variante utilizando *Path Compression*). Si bien estos algoritmos permiten calcular, dado un grafo conexo, un árbol generador mínimo correspondiente a dicho grafo, no es cierto que se los pueda utilizar de forma directa para saber si una arista del grafo pertenecerá a todos, alguno o ninguno de los posibles árboles generadores mínimos del grafo. Sin embargo, mostraremos que sí es posible alterar al grafo original de manera tal de poder responder la pregunta sin alterar los algoritmos de Prim y Kruskal para ello.

Sea $G = (V, X)$ el grafo original, sea $e_i \in X$ una arista de G con peso w_i, y sea T el árbol generador mínimo generado por alguno de los métodos mencionados (la manera en que exhibiremos el método hace que sea cierto para cualquier método para calcular árboles generadores mínimos). Sea p el peso de T, es decir, la suma de los pesos de las aristas de T. Si consideramos el grafo G', con los mismos conjuntos de nodos y aristas, con la única diferencia de que el peso de e_i es $w_i + 1$, el peso de su árbol generador mínimo T' debe ser necesariamente igual a p o a $p + 1$. Es claro que no puede ser menor a p, pues si no el peso del árbol original también sería menor a p, eligiendo las mismas aristas para generarlo. De la misma forma, se puede ver que no puede ser mayor a $p + 1$: eligiendo las mismas aristas que para generar T, solo puede aumentar en 1, y esto ocurre únicamente si $e_i \in T$. Con esta idea en mente, afirmamos lo siguiente:

Lema 1. *$peso(T') = p + 1 \iff e_i$ está en todos los posibles árboles generadores mínimos de G.*

Demostración. $peso(T') = p + 1 \implies e_i$ está en todos los posibles árboles generadores mínimos de G. En efecto, si hubiera un árbol generador mínimo de G que no contuviera a e_i, ese árbol también sería un árbol generador del grafo G', con peso $p < p + 1 = peso(T')$, lo cual es absurdo por la definición de T'.

e_i está en todos los posibles árboles generadores mínimos de G $\implies$ $peso(T') = p + 1$. Por el contrarecíproco, es equivalente a probar que $peso(T') = p \implies$ existe árbol generador mínimo de G que no contiene a e_i. Si $e_i \in T'$ (recordando que el e_i de G' tiene aumentado en 1 el peso respecto del e_i de G), esto implicaría que, eligiendo las mismas aristas para obtener el árbol generador mínimo de G, el peso de T sería $p - 1$, lo cual es absurdo por la definición de p. Por lo tanto, T' no puede contener a e_i. Pero T' es un árbol generador mínimo de G que no contiene a e_i, lo cual es justamente lo que queríamos probar. □

Por lo tanto, hemos hallado una condición que nos permite determinar si una arista pertenece a todos los árboles generadores mínimos de un grafo. De manera análoga, podemos desarrollar otro método para determinar si una arista e_i no pertenece a ningún árbol generador mínimo de G: definiendo un grafo G', donde el peso de e_i es igual a $w_i - 1$, y definiendo T' como el árbol generador mínimo de G obtenido utilizando el mismo método que para obtener T, afirmamos lo siguiente:

Lema 2. *$peso(T') = p - 1 \iff e_i$ está en algún árbol generador mínimo de G.*

Demostración. $peso(T') = p-1 \implies e_i$ está en algún árbol generador mínimo de G. Es inmediato ver que e_i pertenece a algún árbol generador mínimo de G': si todos los árboles generadores mínimos de G' no contuvieran a e_i, se podría tomar cualquiera de ellos y obtener un árbol generador mínimo de G, el cual tendría peso $p-1$, pues estaría conformado por aristas de G' cuyos pesos serían iguales a los de G, lo cual es absurdo por la definición de p. Por lo tanto, el e_i de G' (es decir, con peso $w_i - 1$) pertenece a algún árbol generador mínimo de G'. Por ser árbol generador mínimo de G, su peso es $p - 1$. Si ahora se considera el mismo árbol, pero incrementando en 1 el peso de e_i (es decir, devolviéndolo a w_i), se tiene un árbol generador de G, cuyo peso es igual a p, con lo cual en realidad es un árbol generador mínimo de G, lo cual es justamente lo que queríamos probar: e_i pertenece a algún árbol generador mínimo de G.

e_i está en algún árbol generador mínimo de $G \implies peso(T') = p-1$. De manera análoga al lema anterior, se puede ver que todos los árboles generadores mínimos de G' deben tener peso p o $p - 1$. Entonces, lo que hay que probar es que $peso(T')$ no puede ser p. Como e_i (con su peso original w_i) pertence a un árbol generador mínimo de G, el cual tiene peso p, si se define un árbol con las mismas aristas, pero con el peso de e_i reemplazado por $w_i - 1$, se cuenta con un árbol generador de G', que tiene el peso mínimo posible para un árbol generador de G', por lo cual es árbol generador mínimo de G'. Esto implica que $peso(T')$ es también igual a $p - 1$, pues dos árboles generadores mínimos distintos no pueden tener pesos distintos (ya que, si ocurriera eso, uno de ellos no sería árbol generador mínimo). □

Por lo tanto, hemos hallado una condición que nos determinar si una arista pertenece a algún árbol generador mínimo de un grafo. Si combinamos esto con la condición para verificar si una arista pertenece a todos los árboles generadores mínimos de un grafo, sumado al hecho de que las condiciones que obtuvimos son equivalentes a lo que queremos verificar (y no condiciones suficientes pero no necesarias), nuestra forma de verificar si una arista pertenece a todos/algunos/ninguno de los árboles generadores mínimos de un grafo será la siguiente:

Algoritmo 1 Determinar, para cada arista de G, si pertenece a todos, alguno o ningún AGM de G.

```
1: T ← calcularAGM(G)
2: pesoOriginal ← peso(T)
3: para arista e ∈ G hacer
4:     aumentar en 1 el peso original de e
5:     T_aumentado ← calcularAGM(G)
6:     pesoAumentado ← peso(T_aumentado)
7:     decrementar en 1 el peso original de e
8:     T_decrementado ← calcularAGM(G)
9:     pesoDecrementado ← peso(T_decrementado)
10:    si pesoOriginal+1 = pesoAumentado entonces
11:        reportar que e pertenece a todos los AGM del grafo
12:    si no si pesoOriginal-1 = pesoDecrementado entonces
13:        reportar que e pertenece a algún AGM del grafo
14:    si no
15:        reportar que e no pertenece a ningún AGM del grafo
16:    fin si
17: fin para
```

Claramente, la complejidad total depende del algoritmo que se use para calcular los árboles generadores mínimos de las líneas 1, 5 y 8. Nuestra implementación de los métodos guardará una variable extra que llevará la suma de las aristas que se van agregando al árbol, de manera tal de poder devolver en O(1) el peso del árbol generador mínimo. A continuación, mostraremos en detalle los tres algoritmos implementados, junto con la complejidad total del algoritmo al usar cada uno de los métodos para calcular árboles generadores mínimos.

2.1.1. Prim

El algoritmo de Prim se basa en la siguiente idea: en cada iteración, agregar al árbol generador mínimo parcial, una arista de peso mínimo elegida del conjunto de aristas del grafo que no están en el árbol parcial, de manera tal que uno de sus extremos esté incluido en el árbol parcial y el otro no. De esta manera, en cada iteración, se tiene un subárbol del árbol generador mínimo total formado por j aristas de peso mínimo del grafo original, con lo cual cuando se ejecutaron $n-1$ iteraciones, se cuenta con un subárbol de $n-1$ aristas de peso mínimo del grafo original, es decir, un árbol generador mínimo.

Con esta idea en mente, el algoritmo de Prim para obtener un árbol generador mínimo de un grafo G es el siguiente:

Algoritmo 2 Algoritmo de Prim para obtener un árbol generador mínimo de un grafo G

```
1: listaAdyacencias ← vector de tamaño n, siendo cada posicion un vector vacío
2: para arista e ∈ G hacer  ▷ convertir la lista completa de aristas a la representación como lista de adyacencias
3:     agregar e.b al final de listaAdyacencias[e.a]
4:     agregar e.a al final de listaAdyacencias[e.b]
5: fin para
6: adentroDelArbol ← vector de booleanos de tamaño n, inicializado en false
7: previo ← vector de enteros de tamaño n, inicializado en -1
8: key ← vector de enteros de tamaño n, inicializado en +∞
9: previo[0] ← -1  ▷ arbitrariamente, se toma el vértice 0 como el inicial del árbol
10: key[0] ← 0
11: para i ∈ {0, .., n − 1} hacer
12:     verticeMinimo ← buscar el v ∈ {0, ..., n − 1} con key[v] mínimo y adentroDelArbol[v] = false
13:     adentroDelArbol[verticeMinimo] ← true  ▷ indico que verticeMinimo está en el árbol parcial
14:     para adyacente ∈ listaDeAdyacencias[verticeMinimo] hacer
15:         si adentroDelArbol[adyacente] = false ∧ peso(verticeMinimo, adyacente) <key[adyacente] entonces
16:             previo[adyacente] ← verticeMinimo
17:             key[adjacente] ← peso(verticeMinimo, adyacente)
18:         fin si
19:     fin para
20: fin para
21: aristas ← array vacio de aristas
22: para i ∈ {1, .., n − 1} hacer  ▷ se indexa desde 1 pues el vértice 0 tiene -1 como un antecesor artificial
23:     agregar (i, previo[i], costo(i, previo[i]) al final de aristas
24: fin para
25: devolver aristas
```

Como muestra el pseudocódigo, el array *adentroDelArbol* guarda, para cada vértice del grafo, un booleano que indica si el vértice está o no en el subárbol parcial. Así, se comienza árbitrariamente por el vértice 0 y luego, en cada iteración se agrega un vértice fuera del subárbol parcial que esté a distancia mínima de algún otro vértice presente en el subárbol, para lo cual se guardan los arrays *key*, que guarda la distancia mínima entre un vértice fuera del subárbol parcial y otro adentro del subárbol parcial, y *previo*, que guarda el identificador del vértice al que está conectado cada nodo fuera del subárbol parcial. De esta manera, se preserva el invariante: el conjunto de vértices cuyo valor en *adentroDelArbol* está seteado a *true* forma un subárbol del árbol generador mínimo, conectado entre sí por aristas de peso mínimo elegidas de forma golosa entre las que tienen un extremo dentro del subárbol y el otro afuera. De esta forma, cuando el ciclo termina, se tienen n-1 aristas que cumplen con esa propiedad; es decir, con un árbol generador mínimo. El ciclo del final cumple el propósito de transformar las estructuras de datos auxiliares utilizadas en el algoritmo a la lista de aristas que compone el árbol generador mínimo, con sus pesos originales. Esto demuestra la correctitud del algoritmo. A continuación, mostraremos el cálculo de complejidad asintótica del algoritmo de Prim:

- Convertir la lista de aristas a la lista de adyacencias tiene complejidad $O(n + m)$
- La inicialización de los arrays *adentroDelArbol*, *previo* y *key* tiene complejidad $O(n)$.
- El ciclo principal, encargado de agregar los vértices al árbol, se ejecuta $n - 1$ veces. En primer lugar, en cada iteración, recorre todas las adyacencias del vértice agregado. Si se considera por separado esa parte del ciclo principal, la cantidad total de veces que se ejecutará la lógica de recorrer las adyacencias es exactamente m. En segundo lugar, se busca linealmente el vértice con mínimo valor de key que esté afuera del árbol parcial (con lo cual, haciendo el mismo análisis que para las adyacencias, el costo total de esto es costo $O(n^2)$). Por lo tanto, el ciclo principal completo tiene complejidad $O(n^2 + m)$.
- El ciclo encargado de convertir el contenido de los arrays en una lista de aristas tiene complejidad $O(n)$.

Por lo tanto, la complejidad asintótica de nuestra implementación del algoritmo de Prim, obtenida como la suma de las complejidades de todos los pasos, es $O(n^2 + m)$. Notando además que para un grafo cualquiera vale que $m < n^2$, esa complejidad puede refinarse como $O(n^2)$.

2.1.2. Kruskal

La idea intuitiva del algoritmo de Kruskal es la siguiente: agregar, en cada iteración, una arista de peso mínimo que no forme un ciclo con las aristas previamente agregadas. De esta manera, una vez que se ejecutaron $n - 1$ iteraciones,

se cuenta con un árbol generador mínimo, pues se tiene un grafo de n vértices y $n-1$ aristas, sin ciclos (con lo cual ese grafo es en realidad un árbol) y de peso mínimo. Para poder responder, dado un estado parcial del árbol generado por el algoritmo de Kruskal y una arista candidata a ser agregada al árbol, si la arista candidata forma algún ciclo con alguna combinación de aristas presentes en el árbol parcial, recurriremos a una estructura de datos denominada *Disjoint Set*. Esta estructura de datos provee dos operaciones: Find(x) y Union(x, y). Find(x) devuelve un 'representante' del conjunto (disjunto con todos los otros) al que pertenece el elemento x; Union(x, y) hace la unión de los conjuntos disjuntos a los que pertenecen los elementos x e y, actualizando de forma consistente el representante de los dos conjuntos a unir. Para el caso concreto del algoritmo de Kruskal, utilizaremos *Disjoint Set* de la siguiente manera: los elementos serán los nodos del grafo original, y los conjuntos representarán la componente conexa en la que están los elementos. De esta forma, al elegir una arista de peso mínimo, se puede verificar si formará un ciclo comparando el valor de retorno de la función Find sobre sus dos extremos: serán iguales cuando la arista forme un ciclo, y distintos en caso contrario. Por último, en el caso de que los valores de retorno de Find de los dos extremos sean distintos (con lo cual sea seguro agregar la arista candidata), sólo restará unir los dos conjuntos, lo cual se corresponde con la operación de unir dos componentes conexas distintas mediante una arista que hace de puente entre ellas.

Con esta idea en mente, el algoritmo de Kruskal para obtener un árbol generador mínimo de un grafo G es el siguiente:

Algoritmo 3 Algoritmo de Kruskal para obtener un árbol generador mínimo de un grafo G

```
1: set ← Disjoint Set de n elementos
2: queue ← heap de aristas, donde las más prioritarias serán las de menor peso
3: para arista ∈ aristas de G hacer
4:     agregar arista a queue
5: fin para
6: aristas ← array vacio de aristas
7: mientras queue no esté vacío hacer
8:     arista ← obtener próxima arista de queue
9:     si Find(set, e.a) != Find(set, e.b) entonces
10:        Union(set, e.a, e.b)
11:        agregar arista al final de aristas
12:    fin si
13: fin mientras
14: devolver aristas
```

Mediante este pseudocódigo, podemos justificar la correctitud del algoritmo: en cada iteración, se extrae una arista del heap, que las devuelve en orden creciente según sus pesos. Si los dos extremos de la arista extraída del heap no pertenecen a la misma componente conexa (condición que se verifica con los valores de retorno de Find sobre los vértices) la arista se agrega al conjunto de aristas parcial; caso contrario, se descarta la arista. Con esto, sabemos que en cada paso el conjunto parcial de aristas no contiene ningún ciclo y tiene peso mínimo. Por ser conexo el grafo original, sabemos además que se puede elegir un subconjunto de las aristas originales para formar un árbol generador, con lo cual, después de recorrer todas las aristas del grafo original, en orden creciente por sus pesos, y agregar únicamente las que no formen ciclos en el conjunto parcial de aristas, lo que se tiene al final es un arbol generador mínimo, como queríamos ver.

Si bien este pseudocódigo explica algunos aspectos de la complejidad asintótica de este algoritmo (como por ejemplo el costo $O(m*log(m))$ de crear el heap y del ciclo de buscar aristas de peso mínimo que no formen ciclos), falta explicar en detalle los algoritmos de *DisjointSet*:

Algoritmo 4 DisjointSet: Inicialización

```
1: n ← cantidad de elementos del set
2: padre ← array de tamaño n
3: para i ∈ {0, .., n − 1} hacer
4:     padre[i] ← i
5: fin para
```

Algoritmo 5 DisjointSet: Find

```
1: si x = padre[x] entonces
2:     return x
3: si no
4:     return Find(padre[x])
5: fin si
```

Algoritmo 6 DisjointSet: Union

```
1: raizX ← Find(x)
2: raizY ← Find(y)
3: padre[raizX] ← raizY
```

Con estas implementaciones, se puede ver que las tres operaciones tienen costo O(n), donde por n, nos referiremos indistintamente a los n elementos del conjunto y a los n vértices del grafo. Es evidente que la inicialización tiene costo O(n), pues es un ciclo que a cada elemento le asigna a él mismo como su representante. La operación Find también tiene complejidad asintótica O(n) en el peor caso: si todos los elementos están en el mismo conjunto, podría darse una situación donde se ejecuten $n-1$ llamadas recursivas a Find hasta llegar al elemento que coincide con el representante del conjunto. Esto a su vez implica que la operación Union también tiene costo O(n), puesto que su implementación consiste en dos llamadas a Find más la operación de costo O(1) de sobreescribir la posición de un array. Pero entonces, sumando las complejidades de todas las operaciones del algoritmo:

$$O(n)+O(m*log(m))+O(m*(log(m)+3*n)) = O(n+m*log(m)+m*log(m)+m*n) = O(n+m*log(m)+m*n))$$

En este punto, se puede hacer algunas observaciones para refinar la cota. En primer lugar, $m > 1 \implies m*n > n$, con lo cual se puede descartar el factor n de la complejidad. Por lo tanto, se puede reescribir la complejidad anterior como:

$$O(m*log(m)+m*n) = O(m*(log(m)+n))$$

En segundo lugar, notando que $m < n^2$, se puede reescribir:

$$O(log(m)) = O(log(n^2)) = O(2*log(n)) = O(log(n))$$

Entonces, reemplazando de esa manera $log(m)$ por $log(n)$, se obtiene:

$$O(m*(log(m)+n)) = O(m*(log(n)+n)) = O(m*n).$$

Por lo tanto, el costo asintótico del algoritmo de Kruskal, con esta implementación de la estructura *DisjointSet*, es O($m*n$).

2.1.3. Kruskal con Path Compression

El algoritmo de Kruskal con Path Compression supone una mejora en el comportamiento asintótico de la versión original, mediante una implementación distinta de la estructura *Disjoint Set*. Esta implementación alternativa estará basada en la técnica de *Path Compression*. Esta técnica se basa en lo siguiente: en su versión original, el costo de la operación Find es O(n), pues si todos los vértices estuvieran en la misma componente conexa, las llamadas recursivas a Find(parent) podrían terminar recorriendo todos los vértices. Como lo que se busca al ejecutar Find(x) es el vértice representante de la componente conexa que incluye a x, la técnica de *Path Compression* consiste en actualizar los representantes de cada vértice involucrado en las llamadas recursivas a Find. Esta lógica puede expresarse mediante el siguiente pseudocódigo:

Algoritmo 7 DisjointSet: Find con *Path Compression*

```
1: si parent[x] ≠ x entonces
2:     parent[x] ← Find(parent[x])
3: fin si
4: devolver parent[x]
```

La técnica de *Path Compression*, como indica el pseudocódigo anterior, entra en juego en la ejecución de Find. Junto con esta heurística, suele implementarse otra, que se aplica en la operación Union. Esta técnica se denomina *Union By Rank*, que consiste en lo siguiente: para cada vértice x, se guarda un ranking numérico no negativo, que funciona como cota superior de la cantidad de aristas máxima entre x y una hoja de la componente conexa en la que está. Este ranking se inicializa en 0 para todos los vértices, pues cada vértice comienza en una componente conexa en la que sólo está él mismo. Luego, ante cada llamada a Union(x, y) , se consideran dos casos: si los rankings de los representantes de las componentes conexas que los contienen son distintos, se elige al de mayor ranking como representante de la unión de los dos; si son iguales, arbitrariamente se elige como nuevo representante a alguno de los dos y artificialmente se le incrementa el ranking en 1. Esta lógica puede expresarse mediante el siguiente pseudocódigo:

Algoritmo 8 DisjointSet: Union con *Union By Rank*

```
1: xRaiz ← Find(x)
2: yRaiz ← Find(y)
3: si ranking(xRaiz) >ranking(yRaiz) entonces
4:     padre(yRaiz) ← xRaiz
5: si no si ranking(yRaiz) >ranking(xRaiz) entonces
6:     padre[xRaiz] ← yRaiz
7: si no
8:     ranking(yRaiz) ← ranking(yRaiz) + 1
9:     padre[xRaiz] ← yRaiz
10: fin si
```

Si bien no haremos la demostración en este trabajo, se puede comprobar que, para m operaciones sobre un *DisjointSet* de n elementos, la combinación de las heurísticas *Path Compression* y *Union By Rank* tienen complejidad asintótica $O(m*\alpha(n))$, donde α es una función *similar* a la inversa de la función de Ackermann, la cual, según Cormen (2009) [1], cumple que a fines prácticos $\alpha(n) < 4$, con lo cual haremos la simplificación de considerar que m operaciones sobre un *Disjoint Set* tienen costo $O(m)$. A su vez, si consideramos que m operaciones sobre un *Disjoint Set* tienen complejidad $O(m)$, cometeremos el *abuso* de considerar $O(1)$ como el costo temporal amortizado para cada una de ellas al momento de calcular la complejidad del algoritmo de Kruskal utilizando esta variante de *Disjoint Set*.

Teniendo este resultado, el pseudocódigo de Kruskal utilizando *Path Compression* puede expresarse de una manera casi idéntica a su versión original:

Algoritmo 9 Algoritmo de Kruskal para obtener un árbol generador mínimo de un grafo G

```
1: set ← Disjoint Set de n elementos
2: queue ← heap de aristas, donde las más prioritarias serán las de menor peso
3: para arista ∈ aristas de G hacer
4:     agregar arista a queue
5: fin para
6: aristas ← array vacio de aristas
7: mientras queue no esté vacío hacer
8:     arista ← obtener próxima arista de queue
9:     si FindPathCompression(set, e.a) != FindPathCompression(set, e.b) entonces
10:        UnionByRank(set, e.a, e.b)
11:        agregar arista al final de aristas
12:    fin si
13: fin mientras
14: devolver aristas
```

Para calcular la complejidad, procederemos como antes: la inicialización del *Disjoint Set* tiene complejidad $O(n)$. La inicialización del heap de aristas tiene complejidad $O(m * log(m))$. Dentro del ciclo que arma el árbol generador mínimo (cuya lógica se ejecuta a lo sumo m veces), se ejecutan dos operaciones utilizando el *Disjoint Set*, para las cuales hemos establecido que tienen costo amortizado $O(1)$, y la operación de extraer el próximo elemento del heap, cuyo costo asintótico es $O(log(m))$. Por lo tanto, la complejidad temporal del algoritmo completo es $O(m * log(m))$. Como última observación, notando que $m < n^2$, $O(m*log(m)) = O(m*log(n*n)) = O(2*m*log(n)) = O(m*log(n))$.

Teniendo desarrollados los tres algoritmos, junto con sus complejidades temporales, ya estamos en condiciones de obtener la complejidad temporal del método para obtener la solución al problema de decidir, para cada arista, si pertenece a todos/alguno/ningún árbol generador mínimo del grafo. Si se utiliza el algoritmo de Prim para calcular

árboles generadores mínimos, la complejidad temporal de la solución al problema se obtiene como $O(n^2)$ (por calcular el árbol generador mínimo del grafo original) + $O(m * n^2)$ (por calcular, para cada arista e_i, dos árboles generadores mínimos correspondientes a los grafos con $peso(e_i) = w_i + 1$ y $peso(e_i) = w_i - 1$), con lo cual la complejidad total utilizando Prim es $O(m * n^2)$. De manera análoga, se puede ver que la complejidad utilizando Kruskal es $O(m^2 * n)$, y la complejidad utilizando Kruskal con *Path Compression* es $O(m * m * log(n)) = O(m^2 * log(n))$.

Esto concluye con el desarrollo de este problema. En las siguientes secciones de este informe se procederá a analizar de forma experimental la performance de los distintos métodos para resolver el problema.

2.2. Hiperauditados

En este problema, nuestro objetivo es el siguiente: dado un mapa de ciudades, y una lista de conexiones entre ellas, determinar el costo mínimo de plata en combustible que se debe gastar para llegar entre cualquier par de ciudades del mapa, sabiendo que la capacidad del tanque de combustible es de 60 litros y teniendo además, para cada ciudad, el costo de cargar combustible allí, junto con el costo en litros de combustible de recorrer una conexión entre dos ciudades.

Si el problema se modelara con un grafo donde los nodos representan ciudades y las aristas las conexiones entre ciudades, siendo el peso de las aristas el costo en litros de combustible, no estaríamos en condiciones de aplicar directamente los algoritmos de camino mínimo para resolverlos, lo cual constituye uno de los objetivos de este trabajo. Por lo tanto, para este problema, nos valdremos de una representación alternativa del problema a resolver, la cual también utilizará grafos como herramienta de modelado.

Como hemos explicado, nuestro objetivo es obtener la mínima cantidad posible de plata a gastar en combustible para conectar cualquier par de ciudades. Para esto, necesitamos modelar este problema mediante un grafo cuyas aristas representen, el costo en plata de *algo* relacionado a unir pares de ciudades. En este punto, haremos dos observaciones:

- La acción de viajar de una ciudad a una de sus ciudades vecinas no cuesta plata, asumiendo que se cuente con suficiente combustible para recorrer esa ruta. Esta acción lleva la situación a un estado donde la cantidad de combustible en el tanque disminuyó tanto como la cantidad de litros de nafta necesaria para recorrer esa ruta.
- La acción de cargar combustible en una ciudad es, de hecho, la única acción que cuesta dinero ejecutar, llevando la situación a un estado donde se gastó tanta plata como el producto entre la cantidad de combustible cargado y el costo del litro de nafta en la ciudad en cuestión.

Con estas dos ideas en mente, nuestro modelo consistirá en lo siguiente: un grafo con 61 niveles, donde cada nivel represente una cantidad de nafta disponible en el tanque en un estado dado del sistema. En cada nivel, los nodos serán las ciudades del mapa original. Por otro lado, las aristas se definirán de la siguiente forma:

- La ciudad c_i del nivel k estará conectada con la ciudad c_i del nivel $k + 1$ por una arista cuyo peso sea el costo de la nafta en c_i, para $k \in \{0, ..., 59\}$. De esta forma, la acción de recorrer esa arista representará cargar un litro de nafta en esa ciudad.
- La ciudad c_i del nivel k estará conectada con la ciudad c_j del nivel l si c_i y c_j son vecinas y el costo en litros de combustible de recorrer la ruta que las une es exactamente $k - l$. Como esta arista representará la acción de viajar de una ciudad a otra teniendo el combustible necesario para hacerlo, tendrá peso igual a 0.

En este punto, vale la pena notar que si no se dispone de la nafta suficiente para recorrer la ruta entre dos ciudades c_i y c_j, es decir, si $k < costo(c_i, cj)$, esta arista no será generada, pues observando la caracterización previa se puede ver que esta arista debería conectar una ciudad con otra que estaría en un nivel negativo del grafo.

Una vez construido este grafo, estamos en condiciones de resolver el problema original utilizando algoritmos de camino mínimo para hallar el costo mínimo (en plata) de viajar entre cualquier par de ciudades del mapa.

En base a todo lo anterior, se puede representar la solución a este problema mediante el algoritmo 10.

Algoritmo 10 Determinar el costo mínimo de viajar entre cualquier par de ciudades del mapa

```
1: n ← cantidad de vértices del grafo
2: m ← cantidad de aristas del grafo.
3: G ← grafo de 61 niveles, con n vértices en cada nivel.
4: para i ∈ {0, .., n − 1} hacer
5:     para j ∈ {0, .., 60} hacer
6:         agregar a G una arista entre c_i^j y c_i^{j+1} de peso l_i    ▷ subíndices:= número de ciudad; supraíndices:= nivel
7:     fin para
8: fin para
9: para conexión ∈ conexiones(mapa) hacer
10:    i ← extremo izquierdo de la conexión
11:    j ← extremo derecho de la conexión
12:    litros ← costo en litros de recorrer la conexión
13:    para k ∈ {0, .., 60} hacer
14:        si k - litros ≥ 0 entonces
15:            agregar a G una arista entre c_i^k y c_j^{k−litros} de peso 0
16:            agregar a G una arista entre c_j^k y c_i^{k−litros} de peso 0
17:        fin si
18:    fin para
19: fin para
20: aplicar un algoritmo de camino mínimo entre todo par de vértices sobre el grafo G
21: para i ∈ {0, ..−, n − 1} hacer
22:    para j ∈ {i + 1, ..., n − 1} hacer
23:        minimoCosto ← +∞
24:        para k ∈ {0, ..., 60} hacer
25:            si costo(c_i^0, c_j^k) <minimoCosto entonces
26:                minimoCosto ← costo(c_i^0, c_j^k)
27:            fin si
28:        fin para
29:        reportar que el costo mínimo entre c_i y c_j es el valor de minimoCosto
30:    fin para
31: fin para
```

En las líneas 25 y 26 se puede ver un detalle extra de nuestro modelado del problema: consideraremos que el vehículo comienza el viaje con el tanque vacío.

Una aclaración importante es que, para resolver este problema mediante el algoritmo A*, utilizaremos un modelo completamente distinto a este, valiéndonos además de una serie de hipótesis adicionales sobre el mapa de ciudades que este algoritmo recibe por parámetro, de manera tal de llevar el problema a las condiciones necesarias para que A* efectivamente resuelva el problema de camino mínimo.

2.2.1. Algoritmo de Dijkstra

Como este algoritmo resuelve la versión del problema con un único origen y múltiples destinos, y el objetivo es hallar el costo mínimo en litros para viajar entre todos los pares de ciudades, se utilizará el algoritmo de Dijkstra una vez por cada vértice del grafo que hemos definido. Para resolver dicho problema, el algoritmo 11 genera, en primer lugar, la lista de sucesores de G y una matriz D de $n \times n$ en la que se almacenarán los resultados. Luego, para cada vértice v de G ejecuta el algoritmo de Dijkstra y guarda el resultado en la v-ésima fila de D. Al finalizar, D posee en la posicíón ij la longitud del camino mínimo entre i y j en G, con $i, j \in V(G)$.

Para probar la correctitud de este algoritmo, debe verificarse que en, cada iteración del ciclo principal, el invariante del algoritmo de Dijkstra se mantiene. Sea v el vértice de G en alguna iteración del ciclo principal.

Lema 3. *En la k-ésima iteración, para todo $w \in V(G)$ tal que $w \notin noVisitados$, D_{vw} [1] contiene la longitud de los caminos mínimos desde v hasta w.*

[1] Dados dos vértices $v, w \in V(G)$, notaremos D_{vw} a la posición de D dada por el número de vértice de v y de w. Esto es así porque se considera que $v_0, v_1, ...v_{n-1}$ son los vértices de G y, por lo tanto, v y w corresponderán a algún v_i, con i entre 0 y $n - 1$. Por ejemplo, si $v = v_1$ y $w = v_4$, D_{vw} será equivalente a escribir D_{14}. Como $0 \leq i \leq n - 1$ y D es una matriz de $n \times n$, no se accede a una posición inválida.

Algoritmo 11 Algoritmo de Dijkstra para obtener los caminos mínimos entre todo par de vértices de un grafo G

```
1: Sucesores ← lista de sucesores de G
2: D ← matriz de n × n
3: para v ∈ V(G) hacer                          ▷ se ejecuta n veces el algoritmo de Dijkstra
4:     noVisitados ← vector vacío de vértices
5:     para w ∈ v(G) hacer
6:         si w ≠ v entonces
7:             D_vw ← peso de la arista (v, w) si (v, w) ∈ E(G) o +∞ en caso contrario
8:             agregar w a noVisitados
9:         si no
10:            D_vw ← 0
11:        fin si
12:    fin para
13:    mientras haya elementos en noVisitados hacer
14:        elegir u ∈ noVisitados tal que D_vu = mín_{i∈noVisitados} m_vi
15:        eliminar u de noVisitados
16:        para z ∈ Sucesores[u] hacer
17:            D_vz ← min(D_vz, D_vu+ peso de la arista (u, z))
18:        fin para
19:    fin mientras
20: fin para
21: devolver m
```

Demostración. Antes de entrar en el *mientras*, se agregan todos los vértices de G menos v en $noVisitados$ y se actualiza D_v con la distancia del camino directo desde v hasta cada uno de los demás. $D_{vv} = 0$, porque G no es un pseudografo y además no puede ser menor, porque las aristas de G tienen pesos no negativos. Entonces, como v es el único vértice que no pertenece a $noVisitados$, vale el lema antes de entrar al ciclo.

Suponemos ahora que la hipótesis se cumple para todas las iteraciones hasta alguna llamada k. Sea u el vértice perteneciente a $noVisitados$ tal que D_{vu} es mínimo en la iteración k.

Sea u el vértice que se elige en la iteración k y que es el primer tal que D_{vu} no es la longitud del camino mínimo entre v y u.

Suponemos que existe un mejor camino entre v y u. $u \neq v$, porque v nunca está en $noVisitados$ y $D_{vv} = 0$ es la longitud del camino mínimo entre v y v. Entonces, seguro hay algún elemento de $V(G)$ que no esté en $noVisitados$ antes de eliminar a u, y debe haber un camino entre v y u pues, si no, $D_{vu} = +\infty$ y no se cumpliría que D_{vu} no contiene la longitud del camino mínimo entre v y u. Como existe un camino entre u y v, en particular existe uno P de longitud mínima. Antes de eliminar u, P conecta un vértice que no está en $noVisitados$ -v- con uno que sí lo está -u-. Consideramos ahora el primer vértice $y \in P$ tal que $y \in noVisitados$ y llamamos x a su predecesor en P. Entonces, $P = v...x\ y...u$ y lo podemos descomponer en $P_1 = v...x$ y $P_2 = y...u$ (notar que y puede ser u y que x puede ser v).

A partir de aquí, llamaremos $\delta(v, u)$ a la longitud del camino mínimo entre v y u, con $u, v \in V(G)$.

Afirmamos que $D_{vy} = \delta(v, y)$ en la iteración k. Notamos que, en la k-ésima iteración, $x \notin noVisitados$. Entonces, como u es el primero tal que $D_{vu} \neq \delta(v, u)$ al ser eliminado de $noVisitados$, se tiene que $D_{vx} = \delta(v, x)$ cuando se eliminó x de $noVisitados$ (por invariante). Como en esa iteración se relajó la arista (x, y), vale la afirmación.

Como y se encuentra .ªntes"que u en P y los pesos de las aristas de G son no negativos (en particular los de P_2), llegamos a que $\delta(v, y) \leq \delta(v, u)$ y, entonces,

$$\begin{aligned} D_{vy} &= \delta(v, y) \\ &\leq \delta(v, u) \\ &= D_{vu}. \end{aligned}$$

Como u e y pertenecían a $noVisitados$ cuando se eligió a u, $D_{vu} \leq D_{vy}$ y, por lo tanto, $D_{vy} = \delta(v, y) = \delta(v, u) \leq D_{vu}$, lo cual es absurdo pus supusimos que D_{vu} no era la longitud del camino mínimo desde v hacia u. Luego, al eliminar u, vale que $D_{vu} = \delta(v, u)$. Como al relajar los ejes no es posible mejorar el camino mínimo (porque las aristas no tienen peso negativo), vale que que en la iteración k para todo $w \in V(G)$ tal que $w \notin noVisitados - u$, D_{vw} contiene la longitud de los caminos mínimos desde v hasta w. □

Luego, al concluir el ciclo, vale que D_{vw} contiene la longitud del camino mínimo desde v hacia w para todo $w \in V(G)$. Como v puede ser cualquier vértice de G, entonces el algoritmo es correcto.

Para el cálculo de la complejidad, se considera que $|V(G)| = n$ y $|E(G)| = m$.

La representación de G como lista de sucesores posee costo $O(m)$, ya que se recorren todos los ejes y la creación de la matriz D tiene complejidad $O(n^2)$.

Al comenzar cada iteración del ciclo principal, se agregan $n-1$ vértices a $noVisitados$ y se cargan los valores de los caminos directos entre v y todos los demás vértices en D_v. Como obtener el valor de un eje en particular consiste en acceder a la matriz con la que se representa D en para este algoritmo, esto cuesta $O(1)$. Entonces, el primer *para* cuesta $O(n)$.

El mínimo se obtiene mediante una búsqueda lineal en $noVisitados$, por lo que su complejidad es $O(n)$. Como se realiza $n-1$ veces, el costo total es $O(n^2)$. Para cada mínimo que se encuentra, se actualizan los valores de los caminos de todos sus adyacentes. Como G es dirgido y todas las operaciones son $O(1)$, esto tiene costo $d_{out_i}O(1)$, donde i es el vértice que se elimina en esa iteración. En total, se realizan $n-1$ de estas operaciones dentro del *mientras* y una más antes de entrar en él. El costo termina siendo $\sum_{i=0}^{n-1} d_{out_i}O(1) = mO(1) = O(m)$.

El costo total del algoritmo 11 resulta

$$\begin{aligned} O(m) + O(n^2) + n(O(1) + O(n) + O(n^2) + O(m)) &= O(m) + O(n^2) + n(O(n + n^2 + m) \\ &= O(m) + O(n^2) + O(n^3 + nm) \\ &= O(m + n^2 + n^3) \\ &= O(n^3) \end{aligned}$$

.

2.2.2. Algoritmo de Dijkstra con *priority queue*

El algoritmo 12 es similar al presentado en la sección anterior. En este caso, sin embargo, se utiliza una cola de prioridad para almacenar los valores de los caminos mínimos desde el origen hacia todos los vértices no visitados por el algoritmo de Dijkstra. De esta forma, se evita realizar una búsqueda lineal entre todos los vértices no visitados para encontrar aquel cuyo camino es mínimo y se reemplaza por un costo logarítmico en la cantidad de vértices. También se elimina la lista de sucesores, pues en cada iteración se recorren todos los vértices no visitados.

Algoritmo 12 Algoritmo de Dijkstra con cola de prioridad para obtener los caminos mínimos entre todo par de vértices de un grafo G

```
1: D ← matriz de n × n
2: para v ∈ V(G) hacer                          ▷ se ejecuta n veces el algoritmo de Dijkstra
3:     noVisitados ← vector de pares (l, vtx), donde l es la longitud del camino mínimo desde v hasta vtx
4:     para w ∈ v(G) hacer
5:         si w ≠ v entonces
6:             D_vw ← peso de la arista (v, w) si (v, w) ∈ E(G) o +∞ en caso contrario
7:             agregar (D_vw, w) a noVisitados
8:         si no
9:             D_vw ← 0
10:        fin si
11:    fin para
12:    transformar noVisitados en un minHeap     ▷ para la comparación se usan los valores de l de cada elemento
13:    mientras haya elementos en noVisitados hacer
14:        extraer u el mínimo de noVisitados
15:        para z ∈ noVisitados hacer
16:            r ← el vértice de u
17:            k ← el vértice de z
18:            si (r, k) ∈ E(G) entonces
19:                D_vk ← min(D_vr, D_vk+ peso de la arista (r, k)
20:                z ← (D_vk, k)
21:            fin si
22:        fin para
23:        restaurar la propiedad de minHeap de noVisitados
24:    fin mientras
25: fin para
26: devolver m
```

En primer lugar, se estudiará la corectitud del algoritmo.

Considérese una iteración del ciclo principal, llamando $v \in V(G)$ al vértice origen en dicha iteración. Como ahora se tiene información duplicada en D_v y en $noVisitados$, debe verificarse que el invariante del algoritmo de Dijkstra sigue valiendo.

Antes de entrar al ciclo, v sigue siendo el único vértice de G que no pertenece a $noVisitados$. Como $D_{vv} = 0$ y, para todo $w \neq v$, D_{vw} es el mismo que en el algoritmo 11, sigue valiendo el invariante antes de entrar al ciclo. Notamos que, para todos los $w \neq v$, se guarda en $noVisitados$ la misma información contenida en D_{vw} junto con el vértice al cual pertenece.

Resta ver que el resto de los elementos que no se eliminan de $noVisitados$ se actualizan correctamente.

A diferencia del algoritmo 11, en esta versión no se recorren todos los vértices adyacentes al elegido sino únicamente los que pertenecen a $noVisitados$. Al recorrer los sucesores en el algoritmo 11, podía ocurrir que estos hubieran sido visitados o no. Como el invariante se preservaba bajo esas condiciones, en particular también lo hara si solamente se revisan los adyacentes aún no visitados.

Por el invariante del algoritmo de Dijkstra sabemos que, para todo $w \notin noVisitados$ en una iteración, el valor D_{vw} corresponde a la longitud del camino mínimo entre v y w. Como la información de D_v se actualiza de la misma manera que antes, entonces esto se sigue manteniendo y, como cada vez que se modifica D_{vz} (donde z es cualquier vértice que no haya sido elegido en la iteración k) se reflejan estos cambios en $noVisitados$, entonces, si cada vez que extraigo un elemento, resulta ser el mínimo, se cumplirá el invariante en la iteración k.

El mínimo obtenido en la primera iteración del *mientras* es el correcto pues $noVisitados$ es un minHeap antes de entrar al ciclo. Como al final del *mientras* se reestablece la propiedad de minHeap de $noVisitados$, entonces en cada iteración el elemento que se extrae de $noVisitados$ es efectivamente el mínimo.

Luego, al concluir una iteración del ciclo principal, vale que D_{vw} contiene la longitud del camino mínimo desde v hacia w para todo $w \in V(G)$. Como v puede ser cualquier vértice de G, entonces el algoritmo es correcto.

Por último, interesa calcular la complejidad.

Construir la matriz de soluciones, al igual que en el caso anterior, tiene un costo de $O(n^2)$. Dentro del ciclo principal, como el vector $noVisitados$ se crea vacío, cuesta $O(1)$. Luego, sigue un ciclo que itera sobre todos los vértices de G inicializando las posiciones de D_v y llenando $noVisitados$. Obtener el peso de una arista y agregar al vector son operaciones que se realizan en tiempo constante; en el primer caso, debido a la representación de G como una matriz de distancias y en el segundo porque se amortiza el costo de una inserción por la cantidad agregada. Entonces, como se hacen n iteraciones, se paga en total $O(n)$ en este ciclo. Ordenar los elementos de $noVisitados$ para que cumplan con la propiedad de Heap cuesta $O(n)$.

A continuación, se analizará qué ocurre dentro del *mientras*. Extraer el mínimo cuesta $O(log(n))$ pues $noVisitados$ es un minHeap. Luego, se realizan n iteraciones con costo $O(1)$ para relajar los ejes de los sucesores del elemento elegido, lo cual termina arrojando una complejidad de $O(n)$. Por último, se vuelve a pagar $O(n)$ para transformar nuevamente a $noVisitados$ en un minHeap.

La suma de los costos de las operaciones que se realizan dentro del *mientras* es $n(O(log(n)) + 2O(n)) = nO(n + log(n)) = O(n^2 + log(n))$. En este punto es necesario hacer una observación, pues la complejidad teórica de esta porción del algoritmo de Dijkstra utilizando una cola de prioridad es $O((n + m)log(n))$. Sin embargo, esto toma en cuenta una estructura que provee una función para modificar la prioridad de un elemento en una posición arbitraria con complejidad $O(log(tam\_cola))$. Así, cada vez que se elige un mínimo, es posible recorrer únicamente los vértices adyacentes a este, sin necesidad de pagar un costo lineal en cada iteración. Como esto no es posible con las bibliotecas de la standard library de C++, y para evitar el tener que implementar una cola de prioridad que cumpliera con esta propiedad, se tomó la decisión de recorrer todos los elementos de $noVisitados$ en cada iteración, pagando el costo lineal que supone tanto esto como restaurar el invariante del heap.

Por todo lo anterior, la complejidad temporal del algoritmo 12 resulta

$$\begin{aligned} O(n^2) + n(O(1) + 2O(n) + O(nlog(n)) + 2O(n^2)) &= O(n^2) + n(O(n + nlog(n) + n^2)) \\ &= O(n^2) + nO(n^2) \\ &= O(n^2 + n^3) \\ &= O(n^3). \end{aligned}$$

2.2.3. Algoritmo A*

Como ya hemos mencionado, para la solución de este problema utilizando A* necesitaremos una hipótesis extra sobre el mapa de ciudades: que se cumpla la desigualdad triangular sobre el el costo de llegar de una ciudad a otra. Es importante notar que esto se refiere a que para cualquier par de ciudades, debe cumplirse que el costo (en plata) de viajar hasta allí directamente sea menor al costo de hacer un desvío por una ciudad intermedia. Claramente, el problema original hace que esto no vaya a cumplirse para cualquier mapa de ciudades: podría resultar más conveniente

pasar por una ciudad intermedia si la nafta resultara ser especialmente barata en dicha ciudad intermedia. Por lo tanto, para que se cumplan las hipótesis de A* le impondremos al mapa de ciudades una serie de restricciones extra:

- Para evitar que el paso por una ciudad intermedia resulte menos costoso por el precio del combustible, asumiremos que el costo del combustible es igual en todas las ciudades.
- Combinada con la condición anterior, asumiremos que vale la desigualdad triangular sobre la cantidad de litros de nafta necesaria para conectar dos ciudades. Esto quiere decir lo siguiente: sean ci, cj dos ciudades distintas del mapa. Entonces, $\forall c_k \neq c_i, c_j$, costoEnLitros$(c_i, c_j)$ <costoEnLitros(c_i, c_k) + costoEnLitros(c_k, c_j).

Con estas restricciones extra, a continuación mostraremos que no será necesario modelar el problema con el grafo en niveles que definimos antes. Como le hemos impuesto al mapa de ciudades estas restricciones adicionales, es equivalente minimizar la cantidad de plata necesaria para llegar desde c_i hasta c_j a minimizar los litros de combustible necesarios para dicho viaje, ya que el costo del combustible será igual en todas las ciudades. Por lo tanto, utilizaremos un grafo donde los vértices sean las ciudades y las aristas las rutas entre ellas, siendo el peso de cada arista el producto entre la cantidad de litros necesaria para recorrerla y el precio del combustible en cualquier ciudad.

Ahora, sólo resta definir qué función de subestimación utilizaremos para definir A*. Como la función de subestimación debe (justamente) subestimar el costo de viajar entre un par de ciudades, debemos hallar una cota inferior para el costo del viaje entre cualquier par de ciudades. Para ello, utilizaremos como subestimación del costo una función constante que devuelva el peso de la arista menos pesada del grafo definido para este algoritmo. Como este algoritmo asume que la desigualdad triangular vale sobre los pesos de las aristas, se sabe que el camino entre cualquier par de vértices el grafo siempre estará acotada por la arista menos pesada del grafo.

El siguiente pseudo-código corresponde al algoritmo de A* encargado de encontrar la distancia desde un nodo 'inicio' hacia un nodo 'fin'. El mismo es llamado una vez por cada nodo del grafo, para de esta forma completar la matriz de adyacencias con los costos mínimos para ir desde cualquier ciudad hacia cualquier otra.

Algoritmo 13 Algoritmo de A* para hallar la distancia mínima entre un par de vértices del grafo

```
1: nodosEvaluados ← conjunto de nodos evaluados (comienza vacío)
2: nodosNoEvaluados ← conjunto de nodos no evaluados
3: insertar(nodosNoEvaluados, nodoInicial)
4: distancias ← vector de distancias desde inicio hacia el resto, inicializado en infinito
5: distancias[nodoInicial] ← 0
6: subestimacionDistancia ← vector de subestimaciones de distancias a todos los nodos, inicializado en infinito
7: subestimacionDistancia[nodoInicial] = pesoAristaMinima
8: mientras nodosNoEvaluados no es vacío hacer
9:     nodoActual ← nodo no evaluado con menor valor en subestimacionDistancia
10:    si nodoActual = nodoFinal entonces
11:        devolverdistancias[nodo_actual]
12:    fin si
13:    borrar(nodosNoEvaluados, nodoActual)
14:    insertar(nodosEvaluados, nodoActual)
15:    para todas las rutas que inciden en nodoActual hacer
16:        si vecino del nodoActual no esta en nodosEvaluados entonces
17:            insertar(nodosNoEvaluados, vecino)
18:            distanciaTentativa ← distancias[nodoActual] + peso de la arista entre nodoActual y vecino
19:            si distanciaTentativa ≤ distancias[vecino] entonces
20:                distancia[vecino] ← distanciaTentativa
21:                subestimacionDistancia[vecino] ← nuevaDistancia + pesoAristaMinima
22:            fin si
23:        fin si
24:    fin para
25: fin mientras
```

Para verificar la correctitud de este algoritmo, al igual que en los algoritmos anteriores, mostraremos que el ciclo principal termina y que cuando esto sucede, el vector de distancias se completa con los costos minimos para viajar de una ciudad hacia todo el resto.

El ciclo principal termina cuando *nodosNoEvaluados* llega a ser vacío. Al comienzo, empieza con un solo elemento. Lo primero que se hace es borrarlo de la lista *nodosNoEvaluados* y agregarlo a *nodosEvaluados*, momento en el cual este último queda vacío. Luego, se recorren todos los adyacentes al nodo inicial verificando que estos no estén en la

lista *nodosEvaluados* y se los agrega a la lista de nodos no evaluados. De este modo, en la siguiente iteración, cuando se proceda con los adyacentes de otro nodo, los que ya fueron evaluados no volverán a serlo por la guarda recien mencionada.

También podemos ver que se recorren todos los nodos del grafo, ya que por el problema que estamos modelando, el grafo es conexo, es decir que todo nodo es adyacente de otro nodo, con lo cual todos pasarán por la lista *nodosNoEvaluados* y luego por *nodosEvaluados*.

Como los grafos que utilizaremos con este algoritmo cumplen con la condición de desigualdad triangular, veremos que la segunda guarda del ciclo actualiza correctamente las distancias mínimas entre un nodo y el resto. En este caso, actualizar correctamente dichas distancias mínimas es tomar aquella distancia resultante de llegar hasta el nuevo nodo por el camino mas corto. Y esto es exactamente lo que ocurre: en la primera iteración del ciclo, se guardará como distancias mínimas a los pesos de las aristas entre el nodo inicial y sus adyacentes. Luego se hará lo mismo con los adyacentes al nodo más cercano, sumando además la distancia del primer nodo hacia este otro. Este proceso se repite hasta recorrer todos los nodos.

Para analizar la complejidad de este algoritmo, notaremos que el algoritmo se ejecuta una vez por cada par $(inicio, fin)$, con $1 \leq inicio \leq n$ y $inicio + 1 \leq fin \leq n$, con lo cual se ejecuta $O(n^2)$ veces. Dentro de la ejecución para cada par $(inicio, fin)$, el ciclo que verifica que nodos no evaluados no esté vacío se ejecuta, a lo sumo, $O(n)$ veces. Dentro de este ciclo, se ejecuta una búsqueda lineal del nodo no evaluado con menor valor de subestimación, lo cual tiene costo $O(n)$, y luego se recorren todas las aristas del grafo, ejecutando acciones de tiempo constante (consideramos que las llamadas a funciones sobre los conjuntos toman tiempo constante pues la documentación de la STL indica que tienen costo amortizado $O(1)$), lo cual tiene costo $O(m)$. Por lo tanto, la ejecución para cada par $(inicio, fin)$ tiene complejidad $O(m * n + n^2)$. Pero entonces, recordando que la cantidad de pares $(inicio, fin)$ posibles es $O(n^2)$, obtenemos que la complejidad de ejecutarlo sobre todos los pares es $O(n^4 + m * n^3)$.

2.2.4. Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford (algoritmo 14) resuelve el problema del camino mínimo desde un vértice hacia el resto de un grafo. El mismo genera un arreglo π de n posiciones, las cuales indican el camino mínimo de ir desde el vértice $v \in V(G)$ hacia todos los demás, por lo que en un principio tendrán un camino infinito, dado que no se sabe cómo se puede llegar. En la posición de v tendrá el valor 0, ya que no hay aristas que conecten a un vértice consigo mismo.

Luego, se itera sobre todas las aristas $(a, b) \in E(G)$, $b \neq v$, relajando el valor de llegar al vértice b con el peso de la arista más el peso calculado de llegar al vértice a (que es donde nace la arista), es decir $\pi_b = l(a, b) + \pi_a$. Esto se efectuará hasta que no haya más cambios en π, o hasta haber ejecutado n iteraciones, ya que de encontrar circuitos con longitud negativa no existirá el caso en que no haya más cambios en π, pues cada vuelta extra sobre el circuito negativo ocasionará cambios sobre los valores de π. Por lo anterior, podemos decir que al terminar la iteración k el algoritmo determina el camino mínimo utilizando a lo sumo k aristas entre el nodo v y los otros nodos de $V(G)$.

Algoritmo 14 Algoritmo de Bellman-Ford para obtener los caminos mínimos desde un vértice de un grafo G

```
1:  N ← cantidad de vertices del grafo G
2:  v ← vertice del grafo G
3:  caminoMinimo ← lista de N posiciones
4:  para i ∈ {0, .., n − 1} hacer
5:      caminoMinimo_i ← +∞
6:  fin para
7:  caminoMinimo_v ← 0
8:  i ← 0
9:  mientras no hay cambios en caminoMinimo ó i < N hacer        ▷ Con ciclos negativos el ciclo corta en i = N
10:     caminoMinimo' ← caminoMinimo
11:     para (a, b) ∈ E(G) hacer
12:         si b ≠ v entonces
13:             peso ← peso de la arista (a, b)
14:             caminoMinimo_b ← min(caminoMinimo_b, caminoMinimo'_a + peso)
15:         fin si
16:     fin para
17:     i ← i + 1
18: fin mientras
19: devolver caminoMinimo
```

Mirando con atención el pseudocódigo del algoritmo podemos determinar la complejidad. En primer lugar, tenemos un ciclo de n iteraciones para inicializar π. Luego, en el segundo ciclo recorremos todos las aristas de G y calculamos el camino minimo de llegar a cada vértice, que en el peor caso, el cálculo podría no terminar nunca ya que de tener ciclos negativos π siempre va a cambiar. Pero como mucho calcularemos n caminos mínimos, entonces utilizamos a i para tener una cota superior en este caso. Por lo que la complejidad resulta ser O(n*m) siendo n la cantidad de vértices y m la de aristas.

A continuación, se mostrará la correctitud del algoritmo. Dado un grafo $G = (V, E)$, $v \in V$ y una función para calcular el peso de una arista $f : E \to \mathbb{R}$. Asumiendo que G no contiene ciclos negativos alcanzables desde v, después de completar todas las iteraciones del ciclo ubicado en la linea 9 del Algoritmo 14 tenemos el costo de ir a todos los vertices alcanzables desde v.

Y lo probamos acudiendo al la propiedad de relajación de camino que hacemos en las líneas 12, 13 y 14 del Algoritmo 14. Considerando cualquier vértice u alcanzable desde v podemos decir que $\pi = (u_0, u_1, .., u_k)$ donde $u_0 = v$ y $u_k = u$ cualquier camino mínimo de v a u. Porque π tiene al menos $|V| - 1$ aristas, como en cada iteración del ciclo en la linea 9 recorre todas las aristas con un subciclo, podemos decir que en esta relajando al menos un vertice u_i donde $0 \leq i \leq |V| - 1$. Esto se cumple ya que en cada iteración i del ciclo, el algoritmo completa el camino minimo desde el vértice u_0 al u_i recorriendo en este ciclo todas aristas.

Por último, como el Algoritmo de Bellman-Ford busca los caminos mínimos desde un vértice hacia todos los demás, utilizaremos el algoritmo 15 para poder buscar los caminos mínimos de todos los vértices del grafo G. Para eso, generaremos una matriz de $n \times n$ la cual almacenará los resutlados por fila. Luego, para cada vertice v de G, usaremos el algoritmo 14 (Bellman-Ford) y guardaremos el resultado en la v-ésima fila de la matriz. Al finalizar este procedimiento, la matriz posee en la posicíón ij la longitud del camino mínimo entre i y j en G, con $i, j \in V(G)$.

Algoritmo 15 Algoritmo de Bellman-Ford para obtener los caminos mínimos entre todo par de vértices de un grafo G

```
1: M ← matriz de n × n
2: para v ∈ V(G) hacer                ▷ Ejecuto n veces el algoritmo de Bellman-Ford
3:     M_v ← Corro el Algoritmo 14 para el vértice v
4: fin para
5: devolver M
```

2.2.5. Algoritmo de Floyd-Warshall

El algoritmo de Floyd-Warshall es una solución para el problema de camino mínimo para múltiples orígenes - múltiples destinos basado en programación diñamica. La idea intuitiva es la siguiente: si, para un índice $i \leq n$, se sabe resolver el problema de camino mínimo considerando como soluciones intermedias aquellas que utilizan los vértices $\{v_1, .., v_i\}$ como nodos intermedios, entonces la solución al problema original se obtiene cuando $i = n$.

En primer lugar, se demostrará la correctitud del algoritmo. Sea G un grafo y $V(G) = \{0, 1, ..., n-1\}$. Consideremos un subconjunto $\{0, 1, ..., k\}$ de vértices de G para algún k, con $0 \leq k < n-1$ Para cualquier par de vértices $j, k \in V(G)$, consideremos todos los caminos desde i hasta j cuyos vértices intermedios pertenecen a $\{0, 1, ..., k\}$, y sea P un camino simple de longitud mínima entre ellos.

- Si k no es un vértice intermedio de P, entonces todos los vértices intermedios de P se encuentran en $\{0, 1, ..., k-1\}$. Entonces, un camino de longitud mínima desde i hasta j con todos sus nodos intermedios en $\{0, 1, ..., k-1\}$ también es un camino de longitud mínima desde i hasta j con todos sus nodos intermedios en $\{0, 1, ..., k\}$.
- Si k es un vértice intermedio de $P = i...k...j$, puede descomponerse P en $P_1 = i...k$ y $P_2 = k...j$ de forma tal que $P = P_1 + P_2$. Como todo subcamino de un camino mínimo también es un camino mínimo, P_1 es un camino de longitud mínima desde i hasta k con todos los vértices intermedios en $\{0, 1, ..., k\}$. De hecho, como k no es un vértice intermedio de P_1, entonces todos los vértices intermedios de P_1 pertencen al conjunto $\{0, 1, ..., k-1\}$. Por lo tanto, P_1 es un camino de longitud mínima desde i hasta k con todos los vértices intermedios en $\{0, 1, ..., k-1\}$. Análogamente, P_2 es un camino de longitud mínima desde k hasta j con todos los vértices intermedios en $\{0, 1, ..., k-1\}$.

Habiendo caracterizado la solución óptima, se define $f(i, j, k)$ como el peso del camino mínimo entre i y j tal que todos los vértices intermedios se encuentran en el conjunto $\{0, 1, ..., k\}$ o el peso de la arista (i, j) si no existieran nodos intermedios. Entonces,

$$f(i,j,k) = \begin{cases} peso((i,j)) & \text{si } k = 0, \\ min(f(i,j,k-1), f(i,k,k-1) + f(k,j,k-1)) & \text{si } k \geq 1. \end{cases}$$

Algoritmo 16 Algoritmo de Floyd-Warshall para obtener la longitud de los caminos mínimos entre todos los vértices de un grafo G

```
1: D ← matriz de n × n
2: para v ∈ V(G) hacer
3:     para w ∈ V(G) hacer
4:         si w ≠ v entonces
5:             D_vw ← peso de la arista (v, w) si (v, w) ∈ E(G) o +∞ en caso contrario
6:         si no
7:             D_vw ← 0
8:         fin si
9:     fin para
10: fin para
11: para i ∈ V(G)} hacer
12:     para k ∈ V(G) hacer
13:         si i ≠ k y (i, k) ∈ E(G) entonces
14:             para j ∈ V(G) hacer
15:                 si j ≠ i, j ≠ k y (k, j) ∈ E(G) entonces
16:                     D_ij = min(D_ij, D_ik + D_kj)
17:                 fin si
18:             fin para
19:         fin si
20:     fin para
21: fin para
22: devolver m
```

Luego, el problema de encontrar la longitud de los caminos mínimos entre todo par de vértices de un grafo puede ser resuelto mediante programación dinámica. El algoritmo 16 corresponde a una versión *bottom-up* de la función recursiva. El resultado final, luego de haber computado todos los posibles, equivale a obtener $f(i, j, n)$, es decir la solución óptima para el problema de tamaño n. Es importante aclarar que en el algoritmo presentado se omite la comprobación de la presencia de ciclos de longitud negativa, ya que el grafo utilizado para modelar el problema no posee aristas de peso negativo.

Para hallar la complejidad temporal del algoritmo, considérese en primer lugar que tanto crear la matriz D como inicializarla con los pesos de las aristas entre todo par de nodos cuesta $O(n^2)$. El ciclo principal del algoritmo contiene otros dos ciclos anidados, con operaciones con costo $O(1)$ en su interior. Nótese que, como el grafo está representado como una matriz, se puede conocer si una arista pertenece al conjunto de aristas del grafo en tiempo constante. Como los 3 ciclos iteran n veces, la complejidad es $O(n^3)$. Por lo tanto, la complejidad total del algoritmo 16 es $O(n^2) + O(n^3) = O(n^3)$.

2.2.6. Algoritmo de Dantzig

Este algoritmo resuelve la variante del problema de camino mínimo con múltiples orígenes - múltiples destinos. Al igual que el algoritmo anterior, está basado en programación dinámica: si, para un índice $i \leq n$, se sabe resolver el problema de camino mínimo para el subgrafo inducido por los vértices $\{v_1, .., v_i\}$, está claro que la solución al problema original corresponde a la solución de ese problema asociado con $i = n$. A continuación, mostraremos el pseudocódigo para resolver el problema con esta idea:

Como podemos ver en el algoritmo 17, dado un grafo obtenemos la matriz de adyacencia L del mismo. Primero, para cada $1 \leq k \leq n-1$, consideramos el subgrafo inducido por los vértices $1, .., k$. Lo que haremos en cada iteración $1 \leq i \leq k$ será calcular el costo mínimo para llegar hasta los vértices que se encuentran exactamente una posición afuera de la submatriz de nuestro subgrafo, es decir, $L_{i,k+1} = min_{1 \leq j \leq k}(L_{i,j} + L_{j,k+1})$ y $L_{k+1,i} = min_{1 \leq j \leq k}(L_{k+1,j} + L_{j,i})$.

Luego, con los valores afuera de esta submatriz que representa el subgrafo inducido por esos vértices, recalcularemos el valor minimo de todos los vértices de nuestro subgrafo utilizando el camino mas corto entre el que ya habiamos calculado en nuestro subgrafo, o sumándole a este camino la nueva arista que calculamos en el paso anterior (las que se encuentran exactamente una posición afuera de la submatriz), es decir, $k + 1$. De esta manera, al finalizar cada iteración k estaremos agregando al subgrafo que teníamos al empezar la iteración el vértice $k + 1$, y recalculando los caminos mínimos utilizando la nueva arista (del vertice $k + 1$) o no como parte del camino.

La complejidad del algoritmo 17 es relativamente sencilla de calcular: se cuenta con un ciclo principal que recorre los valores de k desde 1 hasta $n - 1$. Dentro del ciclo principal, hay dos ciclos separados, donde cada uno recorre los

Algoritmo 17 Algoritmo de Dantzig para obtener los caminos mínimos entre todo par de vértices de un grafo G

```
1: L ← matriz de adyacencia de G (n × n)
2: para k ∈ {1, .., n − 1} hacer
3:     para i ∈ {1, .., k} hacer
4:         min_one ← ∞
5:         min_two ← ∞
6:         para j ∈ {1, .., k} hacer
7:             min_one ← min(min_one, L_{i,j} + L_{j,k+1})
8:             min_two ← min(min_two, L_{k+1,j} + L_{j,i})
9:         fin para
10:        L_{i,k+1} ← min_one
11:        L_{k+1,i} ← min_two
12:    fin para
13:    t ← ∞
14:    para i ∈ {1, .., k} hacer                ▷ se verifica que la diagonal no sea negativa
15:        t ← min(t, L_{k+1,i} + L_{i,k+1})
16:        si t <0 entonces
17:            reportar que hay circuitos de longitud negativa
18:        fin si
19:    fin para
20:    para i ∈ {1, .., k} hacer
21:        para j ∈ {1, .., k} hacer
22:            L_{i,j} ← min(L_{i,j}, L_{i,k+1} + L_{k+1,j})
23:        fin para
24:    fin para
25: fin para
26: devolver L
```

valores de i desde 1 hasta k. A su vez, cada uno de los dos ciclos interiores contiene un ciclo más, donde la variable j recorre los valores desde 1 hasta k, y adentro de los ciclos sobre j se ejecutan operaciones de costo O(1). Por todo lo anterior, sabiendo que $k < n$, se puede decir que cada ciclo sobre j se ejecuta a lo sumo n veces. De forma análoga, se puede hacer el mismo razonamiento sobre la cantidad de veces que se ejecuta cada ciclo sobre i, por lo cual cada ciclo sobre i tiene complejidad temporal O(n^2). Por lo tanto, el ciclo principal completo tiene complejidad O(n^3).

3. Experimentación

En esta sección, plantearemos una serie de experimentos, con el objetivo de contrastar empíricamente los resultados teóricos obtenidos en el desarollo de los problemas. Al igual que en las secciones anteriores, dividiremos la experimentación en dos subsecciones, cada una correspondiente a uno de los problemas desarrollados.

3.1. Hiperconectados

Comenzaremos la experimentación de este trabajo por los algoritmos empleados para resolver el problema Hiperconectados: Prim, Kruskal, y Kruskal con *Path Compression*. Para los tres algoritmos, mostraremos de manera empírica que la complejidad teórica obtenida se corresponde con su comportamiento real. Para lograr esto, tomaremos una serie de mediciones temporales para distintos valores de n (cantidad de vértices del grafo que representa las ciudades a conectar) y m (cantidad de conexiones potenciales entre las ciudades). Con el objetivo de poder visualizar los resultados obtenidos, reduciremos el problema de tener las dos variables n y m, definiendo a los valores posibles para la cantidad de aristas del grafo en función de la cantidad de vértices del grafo.

Para verificar que las complejidades obtenidas se correspondan al comportamiento real de los algoritmos al ejecutarlos, nos valdremos del siguiente lema:

Lema 4. *Un grafo con n vértices y más de (n-1)(n-2)/2 aristas es conexo.*

Como los tres algoritmos tienen como precondición que los grafos sean conexos, usaremos $m = \frac{(n-1)(n-2)}{2} + 1$. Por el lema anterior, esta definición de m nos garantizará que, para un grafo de n vértices, elegir esa cantidad de aristas generará un grafo conexo. En particular, esto nos permitirá elegir esa cantidad de aristas de manera pseudo-aleatoria

(es decir, conectar pares de vértices de manera no determinística) y obtener siempre grafos conexos. Con esto en mente, configuraremos los parámetros de este experimento de la siguiente forma:

- Consideraremos valores de n desde 10 hasta 80. Para valores de n más altos, el tiempo de ejecución de los experimentos se hace prohibitamente alto.
- Para cada valor de n, definiremos $m = \frac{(n-1)(n-2)}{2} + 1$.
- Para cada valor de n, generaremos 20 grafos con esos valores de n y m. Para ello, generaremos las $n(n-1)/2$ aristas posibles y elegiremos un subconjunto aleatorio de ellas mediante la funcion **random.sample** de Python. Luego, fijado el valor de n, tomaremos el tiempo como el promedio de las 20 ejecuciones distintas.
- Para cada arista de cada grafo definido como hemos indicado, elegiremos un peso aleatorio entre 1 y 100, generado mediante la función **random.uniform** de Python. Esta decisión fue tomada por lo siguiente: los pesos de las aristas no afectan la cantidad de operaciones que ejecuta ninguno de los algoritmos, con lo cual ignoraremos sus valores para analizar los tiempos de ejecución.

Con los parámetros configurados de esta manera, todavía resta definir cómo haremos para determinar si un algoritmo cumple tener la complejidad de peor caso obtenida durante el desarollo del problema. Tomemos, por ejemplo, la solución al problema utilizando Prim. Los resultados teóricos obtenidos indican que la solución al problema utilizando el algoritmo de Prim tiene complejidad $O(m * n^2)$. Siendo que nuestra definición de $m = \frac{(n-1)(n-2)}{2} + 1$ es $O(n^2)$, podemos decir que con los parámetros definidos de esta forma, la complejidad del algoritmo es $O(n^4)$. Por lo tanto, una manera intuitiva de verificar si esta complejidad efectivamente se cumple en la práctica podría ser graficar, para cada n, el tiempo de ejecución como lo hemos definido y verificar si el gráfico *tiene pinta* de ser un polinomio de orden 4. Como esto es difícil de verificar (además de estar *abierto a interpretación*), tomaremos una estrategia análoga, pero modificando los tiempos medidos. Concretamente: si nuestro desarrollo indica que el tiempo de ejecución debería estar en el orden de $O(n^4)$, dividiremos el valor de cada tiempo de ejecución obtenido por n^3, lo cual debería corresponderse visualmente con un polinomio de grado 1, es decir, con una recta, cuya pendiente dependerá de las constantes descartadas al calcular la complejidad de peor caso, por lo cual, no conoceremos de antemano. Para verificar si las mediciones alteradas de esta manera se corresponden con una recta, las ajustaremos con un polinomio de grado 1, utilizando la técnica de Cuadrados Mínimos y verificaremos que las mediciones modificadas no se *alejen demasiado* del polinomio de grado 1 obtenido.

Con los parámetros de entrada definidos de esa manera, a continuación exhibiremos las mediciones obtenidas.

3.1.1. Complejidad de Prim

Como ya hemos explicado, nuestra estrategia para verificar que el algoritmo de Prim se comporte en la práctica como indica la complejidad temporal obtenida consistirá en dividir los tiempos obtenidos, para cada valor de n, por n^3, y verificar que un polinomio de grado 1 ajuste bien los valores modificados. Con los parámetros de entrada configurados como hemos dicho, los resultados obtenidos fueron los siguientes:

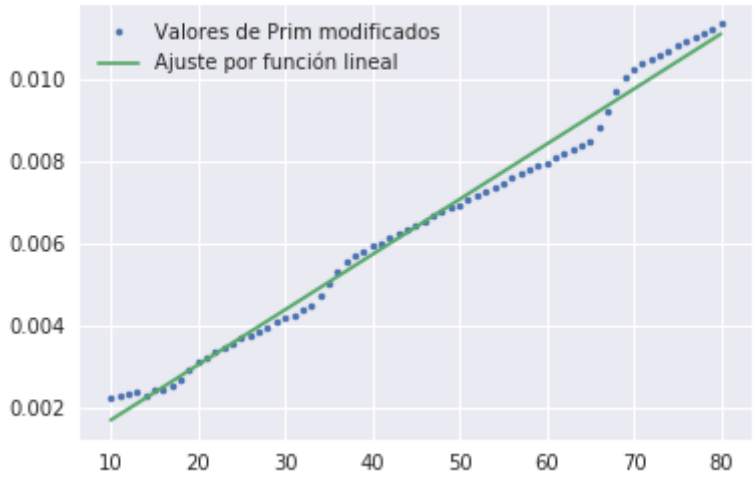

Figura 1: Valores de Prim escalados y ajustados por una función lineal

En el gráfico podemos ver que el polinomio de grado 1 ajusta bien los datos modificados (escalados por n^3 como hemos explicado previamente), con lo cual podemos afirmar con certeza que nuestro cálculo fue correcto.

3.1.2. Complejidad de Kruskal

De manera análoga a lo dicho para Prim, sabiendo que $m = O(n^2)$, la solución al problema utilizando Kruskal es $O(m^2 * n) = O(n^5)$. Por lo tanto, dividiremos los tiempos obtenidos, para cada n, por n^4 y graficaremos los resultados, esperando obtener alguna recta. Con los parámetros de entrada configurados igual que antes, los resultados obtenidos fueron los siguientes:

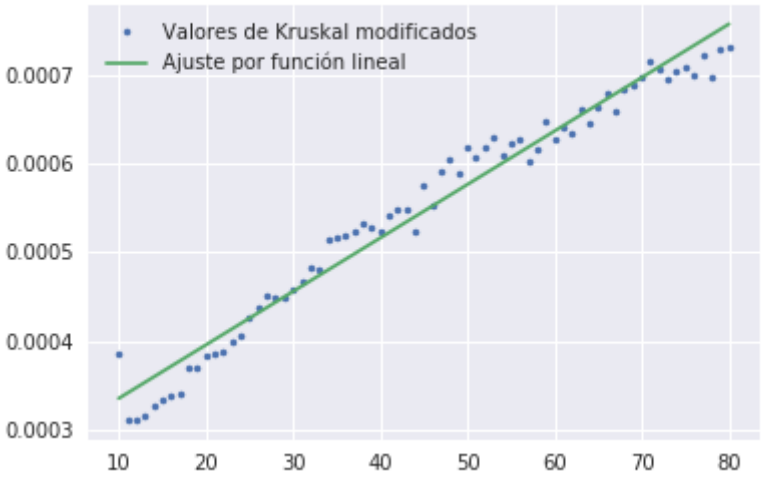

Figura 2: Valores de Kruskal escalados y ajustados por una función lineal

Si bien el ajuste no es tan bueno como el del gráfico mostrado para el algoritmo de Prim, se puede ver de todas formas que la nube de puntos se comporta de una manera similar a la función lineal obtenida con Cuadrados Mínimos, lo cual nos lleva a concluir que nuestro cálculo de complejidad asintótica para este algoritmo fue correcto.

3.1.3. Complejidad de Kruskal con *Path Compression*

De forma análoga a lo dicho para los dos algoritmos anteriores, podemos decir que la solución utilizando Kruskal con con *Path Compression* tiene complejidad $O(m^2 * log(n)) = O(n^4 * log(n))$. Por lo tanto, dividiremos los tiempos obtenidos, para cada valor de n, por $n^3 * log(n))$, esperando obtener un gráfico similar al de una recta. Con los parámetros de entrada configurados igual que antes, los resultados obtenidos fueron los siguientes:

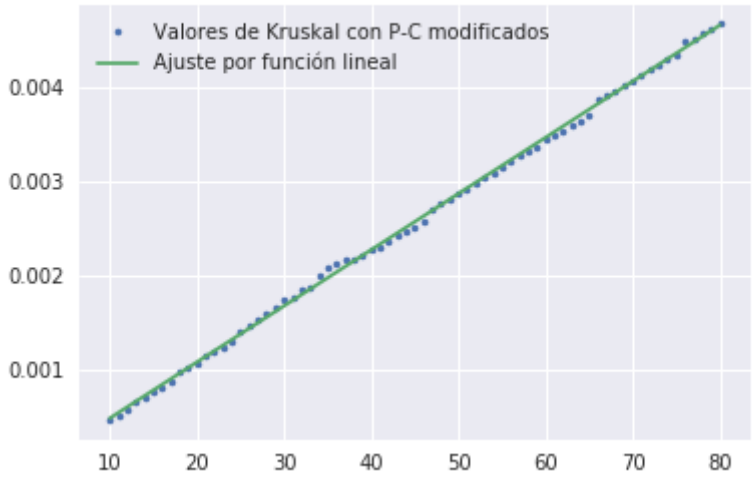

Figura 3: Valores de Kruskal con *Path Compression* escalados y ajustados por una función lineal

Al igual que para los dos algoritmos anteriores, podemos ver que un polinomio de grado 1 obtenido con Cuadrados Mínimos es un buen ajuste para nuestros datos, lo cual nos permite afirmar que nuestra complejidad teórica obtenida para este algoritmo es correcta.

Esto concluye con la parte experimental destinada a contrastar las complejidades teóricas obtenidas contra los comportamientos reales de los algoritmos. A continuación, plantearemos otro conjunto de experimentos, con el objetivo de comparar la performance de los tres algoritmos implementados.

En todos los casos, dado cualquiera de los tres algoritmos implementados, la complejidad temporal de la solución completa consiste en multiplicar por m la complejidad temporal de dicho algoritmo. Por lo tanto, al referirnos a las complejidades de los algoritmos para hacer un análisis y plantear los parámetros de la experimentación, en los próximos experimentos nos referiremos a la complejidad temporal de los algoritmos aislados, es decir, sin el factor m resultante de correr el algoritmo para cada arista.

Para cada comparación entre dos algoritmos, además de plantear experimentos que tomen en cuenta sus detalles de implementación, haremos una comparación de la performance en el caso promedio, el cual se configurará, para todas las comparaciones, de la siguiente manera:

- Se considerarán valores de n desde 10 hasta 70. Para cada valor de n, se tomarán 50 mediciones, con el resto de los parámetros configurados como se indica a continuación. El tiempo de ejecución, para cada valor de n, se obtendrá como el promedio de todas las mediciones.
- Para cada medición correspondiente a un valor de n, se particionará el rango $[(n-1), n*(n-1)/2]$ en 10 partes iguales, y se elegirá un valor de m por cada una de sus divisiones. Para cada valor de m, generaremos un grafo conexo con esa cantidad de aristas.
- Para cada medición correspondiente a un valor de n, los pesos de las aristas se generarán de forma aleatoria ya que, como hemos dicho antes, sus valores no afectan la cantidad de operaciones que ejecuta ninguno de los algoritmos.

3.1.4. Prim versus Kruskal

En primer lugar, compararemos la performance de los algoritmos de Prim y Kruskal. Como explicamos previamente, compararemos en primer lugar los tiempos de ejecución para el caso promedio. Para esta comparación, siendo que el

algoritmo de Prim tiene complejidad $O(n^2)$ y el de Kruskal $O(n * m)$, nuestra intuición inicial es que el algoritmo de Prim mostrará tiempos de ejecución menores o iguales al de Kruskal, ya que, como para un grafo conexo vale que $m \geq n - 1$, con lo cual, salvo para árboles (que sabemos que tienen $n - 1$ aristas), en general también valdrá que $n^2 < n * m$.

Con esta intuición en mente, los resultados obtenidos fueron los siguientes:

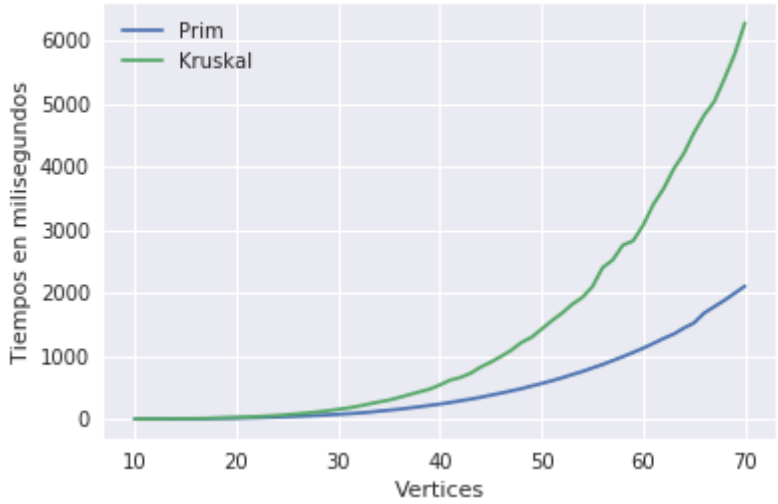

Figura 4: Comparación del caso promedio entre Prim y Kruskal

Los resultados obtenidos confirman nuestra intuición inicial: la solución al problema utilizando el algoritmo de Prim muestra una performance mucho mejor que la solución implementada con el algoritmo de Kruskal en el caso promedio, como lo hemos definido.

La intuición que nos llevó a pensar que este escenario se daría provino de analizar las complejidades de ambos algoritmos y notar que la cantidad de aristas de un grafo conexo puede ser muchísimo más grande que la cantidad de vértices, pues de hecho puede estar en el orden de $O(n^2)$. Sin embargo, también es posible definir grafos conexos de manera tal que m esté en el orden de $O(n)$: por ejemplo, se puede definir un árbol generador cualquiera del grafo original ($n - 1$ aristas) y a eso agregarle una cantidad de aristas que también sea lineal en n. Con esta configuración, se puede reescribir la complejidad de Kruskal como $O(n^2)$, la cual coincide con la complejidad del algoritmo de Prim. Esto motivará otra comparación entre estos dos métodos, la cual configuraremos de la siguiente manera:

- Al igual que antes, tomaremos valores de n entre 10 y 70. Para cada valor de n, tomaremos el tiempo de ejecución asociado como el promedio de todas las mediciones con el mismo valor de n.
- Para cada valor de n, tomaremos m en el rango $[(n - 1), 2 * n]$. De esta forma, garantizaremos que $m = O(n)$ y definiremos un grafo conexo de n vértices y m aristas.
- Al igual que antes, elegiremos valores aleatorios para los pesos de las aristas.

Al restringirnos a un subconjunto de grafos conexos para los cuales se cumple que los algoritmos de Prim y Kruskal tienen la misma complejidad asintótica, ya no estamos en condiciones de predecir que ninguno de los dos vaya a exhibir mejor performance que el otro. Con esta configuración, los resultados obtenidos fueron los siguientes:

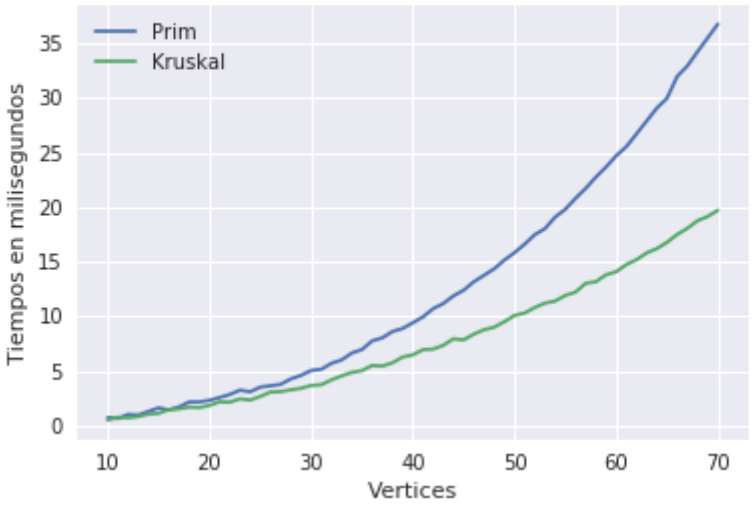

Figura 5: Comparación entre Prim y Kruskal con *Path Compression*

Los resultados obtenidos con esta configuración le dan la ventaja a la solución utilizando el algoritmo de Kruskal. Si bien no podemos dar una explicación certera para esto (pues ambos algoritmos tienen la misma complejidad asintótica), sí podemos aventurar alguna especulación al respecto: es posible que, incluso sin las heurísticas de *Union by Rank* y *Path Compression*, el uso de un heap para extraer las aristas en orden creciente que hace Kruskal suponga una mejora muy grande respecto de la búsqueda lineal para obtener la arista de peso mínimo ejecutada por Prim. Además, si bien la operación de *Union* que hace este algoritmo tiene costo de peor caso $O(n)$, no es cierto que todas las llamadas vayan a caer en el peor escenario posible: al unir por primera vez dos aristas desconectadas (es decir, estando cada una inicialmente en su propia componente conexa) el costo de dicha operación es $O(1)$, pues cada una es su propio representante, por lo cual no se hacen n iteraciones para obtenerlo. De manera análoga, podemos extender este análisis sobre *Union* y observar que las primeras llamadas son mucho menos costosas, con lo cual también podría ocurrir que las primeras invocaciones a *Union* amorticen el costo de las últimas, haciendo que el tiempo global de ejecución de Kruskal sea menor al de Prim.

3.1.5. Prim versus Kruskal con *Path Compression*

De manera análoga a la comparación anterior, comenzaremos los experimentos con estos dos algoritmos comparando el caso promedio, de la manera en que lo hemos definido antes. En este caso, no es evidente si alguno de los dos algoritmos debería mostrar mejor performance que el otro: al ser sus complejidades $O(n^2)$ y $O(m * log(n))$ respectivamente, ninguna de las dos parece ser estrictamente menor que la otra. Si bien es claro que vale que $log(n) < n$, esto no impica que Kruskal con *Path Compression* vaya a correr más rápido, pues un valor desmesuradamente grande de m bien podría sobrecompensar los demás factores de la complejidad al momento de medir los tiempos de ejecución. Más aún, sabiendo además que nuestro valores considerados para m caen dentro del rango $[(n-1), n*(n-1)/2]$ (ya que los hemos elegido así por garantizar que son grafos de entrada que cumplen la precondición de ser conexos), consideramos que el caso promedio que hemos definido podría registrar tiempos menores para Prim que para la versión modificada de Kruskal. Con estas ideas en mente, los resultados obtenidos fueron los siguientes:

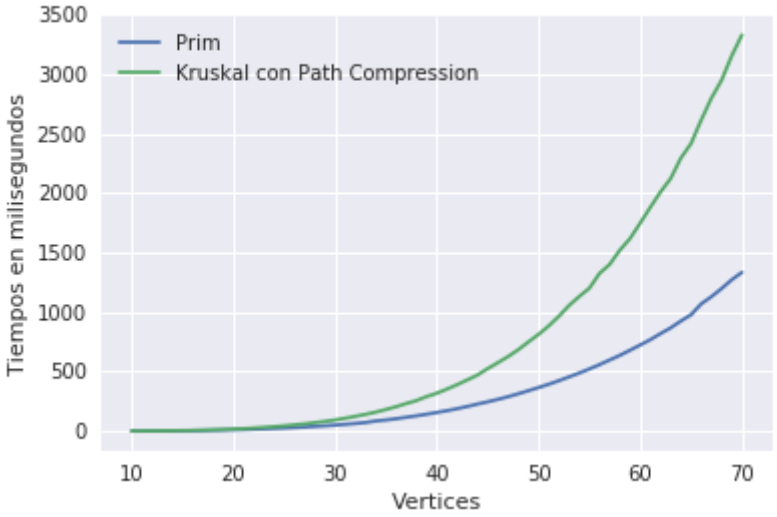

Figura 6: Comparación entre Prim y Kruskal con *Path Compression*

Esto confirma nuestras intuiciones iniciales sobre el caso promedio: los tiempos obtenidos para la solución utilizando Prim son mucho menores, para nuestra definición del caso promedio.

Razonando de forma análoga a la comparación entre Prim y la versión original de Kruskal, podemos restringir la experimentación a un subconjunto de grafos conexos que cumplan que m esté en el orden de $O(n)$, pues en este escenario podemos reformular la complejidad de la versión de Kruskal con *Path Compression* como $O(n * log(n))$, la cual sí cumple con ser estrictamente menor que el $O(n^2)$ de Prim. De forma análoga a lo hecho para la comparación entre Prim y la versión original de Kruskal, definiremos de esta manera nuestro nuevo subconjunto de grafos:

- Al igual que antes, tomaremos valores de n entre 10 y 70. Para cada valor de n, tomaremos el tiempo de ejecución asociado como el promedio de todas las mediciones con el mismo valor de n.
- Para cada valor de n, tomaremos m en el rango $[(n-1), 2*n]$. De esta forma, garantizaremos que $m = O(n)$ y definiremos un grafo conexo de n vértices y m aristas.
- Al igual que antes, elegiremos valores aleatorios para los pesos de las aristas.

Con estos valores para los parámetros de entrada, nuestra intuición indica los tiempos de ejecución de la solución utilizando el algoritmo de Kruskal con *Path Compression* deberían ser menores que utilizando el algoritmo de Prim. Con esto en mente, los resultados obtenidos fueron los siguientes:

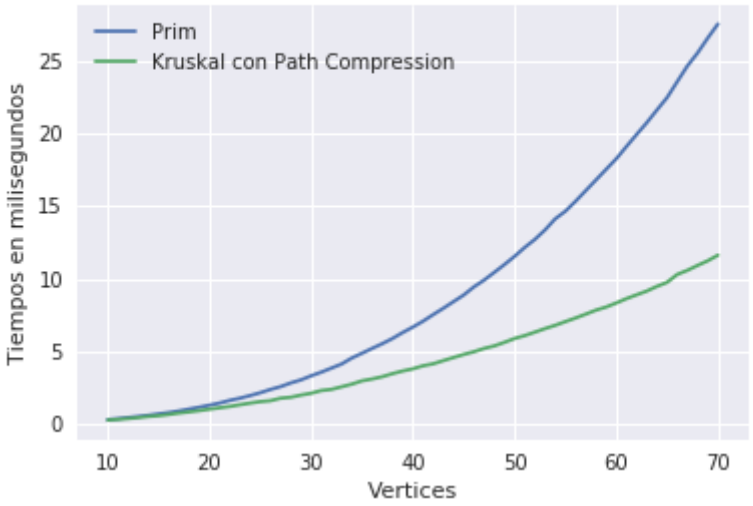

Figura 7: Comparación entre Prim y Kruskal con *Path Compression*, con la cantidad de aristas lineal en la cantidad de vértices

Los resultados obtenidos confirman nuestras intuiciones iniciales: al llevar el valor de m a algo en el orden de O(m), el algoritmo de Kruskal utilizando *Path Compression* muestra mejores resultados que el algoritmo de Prim.

3.1.6. Kruskal versus Kruskal con *Path Compression*

Para cotejar la diferencia de performance entre las dos versiones de Kruskal, nos enfocaremos únicamente en la comparación de sus tiempos de ejecución para el caso promedio. Esta decisión viene motivada por lo siguiente: nuestra intuición indica que, al tratarse de dos algoritmos casi idénticos, donde la única diferencia entre ellos es que las heurísitcas de *Union by Rank* y *Path Compression* permiten obtener cotas de complejidad mucho mejores, deberíamos obtener tiempos de ejecución estrictamente menores para la versión de Kruskal utilizando *Path Compression*. Esto puede expresarse de otra forma: con las cotas de complejidad obtenidas para los dos algoritmos, no deberíamos ser capaces de construir instancias donde la versión original de Kruskal reporte mejores tiempos de ejecución que la versión utilizando *Union by Rank* y *Path Compression*, con lo cual en este caso solamente haremos experimentos que comparen el caso promedio entre ellos. Con esta idea en mente, y los parámetros de los experimentos configurados igual que antes, los resultados obtenidos fueron los siguientes:

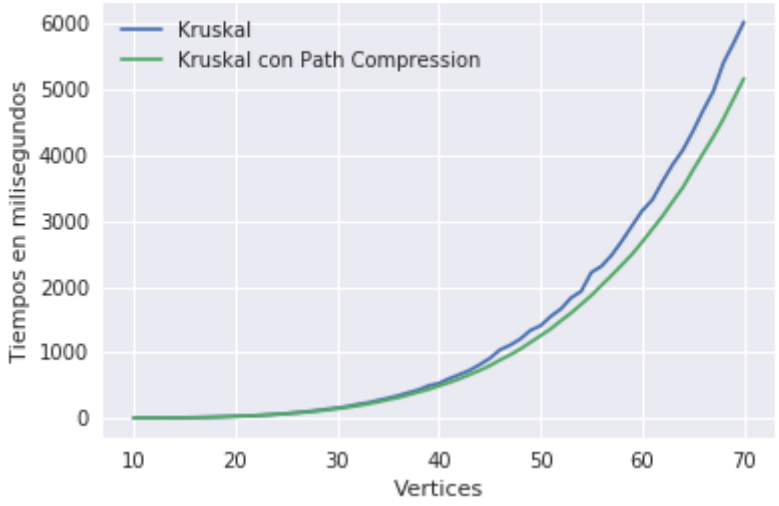

Figura 8: Comparación entre Kruskal y Kruskal con *Path Compression*

Los resultados obtenidos confirman nuestras intuiciones originales: los tiempos observados para las dos implementaciones le dan ventaja a la versión de Kruskal utilizando *Path Compression*, lo cual era esperable siendo que la versión del algoritmo con las heurísticas *Union by Rank* y *Path Compression* representan una implementación estrictamente mejor del mismo algoritmo.

3.2. Hiperauditados

Para comenzar la experimentación sobre los distintos algoritmos diseñados para resolver el segundo problema, haremos lo mismo que para el primero: verificar que sus complejidades teóricas se condigan con los tiempos de ejecución de los mismos en la práctica. Los algoritmos a analizar en este caso son: Dijkstra, Dijkstra con *priority queue*, A* (solo para grafos que cumplen la desigualdad triangular), Bellman-Ford, Floyd-Warshall y Dantzig.

De la misma forma que antes, definiremos a la cantidad de rutas entre ciudades (aristas del grafo), en función de la cantidad de ciudades (nodos). Es decir, definiremos una función f, tal que $f(n) = m$, con n nodos y m aristas. A partir de esto, podremos expresar los tiempos de ejecución de los algoritmos sólo en función de n, facilitando de este modo la visualización de los resultados. La función f será nuevamente $f(n) = \frac{(n-1)(n-2)}{2} + 1$, la cual, por el lema 4, generará grafos conexos. Los grafos generados para los siguientes experimentos serán de la forma:

- Consideraremos valores de n desde 10 hasta 30.
- Para cada n, definiremos a m con la funcion f.
- Para cada n generaremos 5 grafos eligiendo las aristas de forma aleatoria, utilizando la función *random.sample* de Python. Luego, tomaremos el tiempo de ejecución como el promedio de las 5 ejecuciones.
- A cada arista le asignaremos un peso aleatorio entre 1 y 60, para que tenga sentido respecto al problema que estamos modelando: la cantidad de litros necesaria para recorrer una ruta no puede ser menor a 1 ni mayor a la capacidad en litros del tanque del auto.

Al igual que en los experimentos del problema 1, bajaremos los grados necesarios de las complejidades de los algoritmos para poder compararlos con una función lineal. Para esto, dividiremos el valor de cada tiempo de ejecucion obtenido por un polinomio del grado necesario para obtener complejidad de orden lineal, de acuerdo al algoritmo que estemos analizando.

3.2.1. Complejidad A*

Como la complejidad de este algoritmo es $O(n^4 + n^3 * m)$, a partir del m devuelto por la función f, la complejidad teórica es $O(n^4 + n^3 * n^2) = O(n^5)$. Por lo tanto, dividir a los tiempos de ejecución medidos por n^4 debería llevarlos a orden lineal. Sin embargo, al tomar las mediciones de los tiempos de ejecución, nos topamos con que los tiempos de ejecución divididos por n^3 se ajustaban mejor por una recta que al dividirlos por n^4. Esto se podrá visualizar en el gráfico que mostraremos a continuación.

Como este algoritmo tiene un tiempo de ejecución mucho menor a todos los demás, decidimos realizar este experimento para un rango de cantidad de nodos de 10 a 80. Además, el grafo utilizado fue diferente a los que se usaron para el resto de los algoritmos: el precio de la nafta era el mismo en todas las ciudades y la cantidad de litros necesarios para recorrer cualquier ruta era constante (para cumplir con la condición de desigualdad triangular).

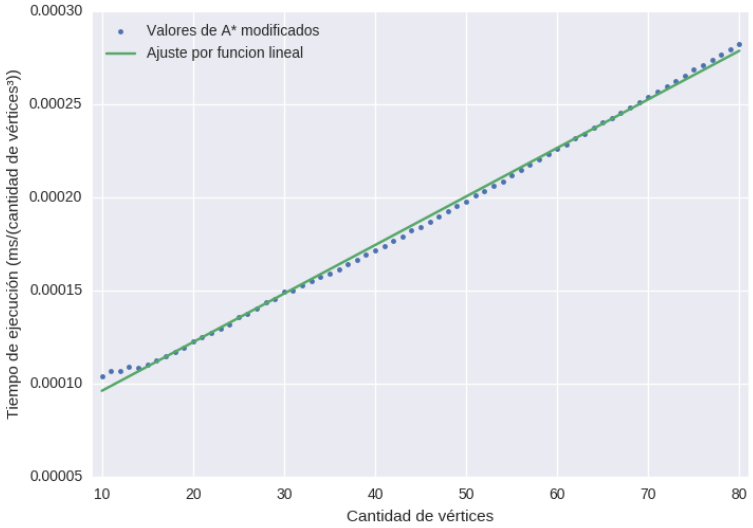

Figura 9: Valores de A* escalados y ajustados por una función lineal

Como se puede observar, los tiempos medidos para A* y escalados por n^3 se ajustan a la recta. Esto nos lleva a un resultado esperable, pero *novedoso*: en la práctica, si el grafo cumple con las condiciones necesarias, A* funciona mucho mejor que lo que sugiere su complejidad de peor caso.

3.2.2. Complejidad Dijkstra

Dado que la complejidad teórica calculada para el algoritmo 11 resultó $O(n^3)$, siendo n el número de vértices del grafo, se dividieron los tiempos de ejecución por n^2 antes de ajustarlos por una recta con Cuadrados Mínimos. Como puede verse en la figura 10, los datos manipulados de esta forma se ajustan a una función lineal y, por lo tanto, la complejidad experimental se ajusta a la teórica.

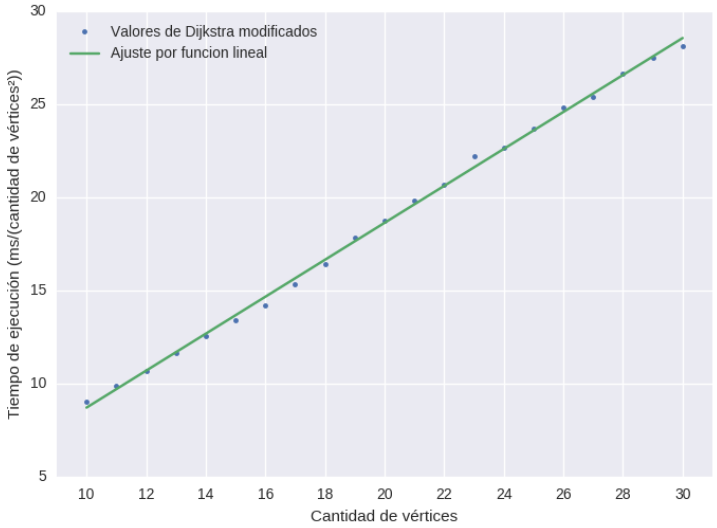

Figura 10: Tiempo de ejecución del algoritmo de Dijkstra en función de la cantidad de vértices. Los tiempos se escalaron por la cantidad de vértices al cuadrado. Se muestra el ajuste de los datos a la recta $y = ax + b$, con $a = 0{,}99$, $b = -1{,}21$ y $R^2 = 0{,}99$. El método utilizado para la aproximación fue el de cuadrados mínimos.

3.2.3. Complejidad Dijkstra utilizando priority queue

De forma análoga a los otros algoritmos, para verificar que la complejidad de la variante del algoritmo de Dijkstra utilizando una *priority queue* se condice con su comportamiento en la práctica, graficaremos los tiempos de ejecución medidos divididos, para cada valor de n, por n^2:

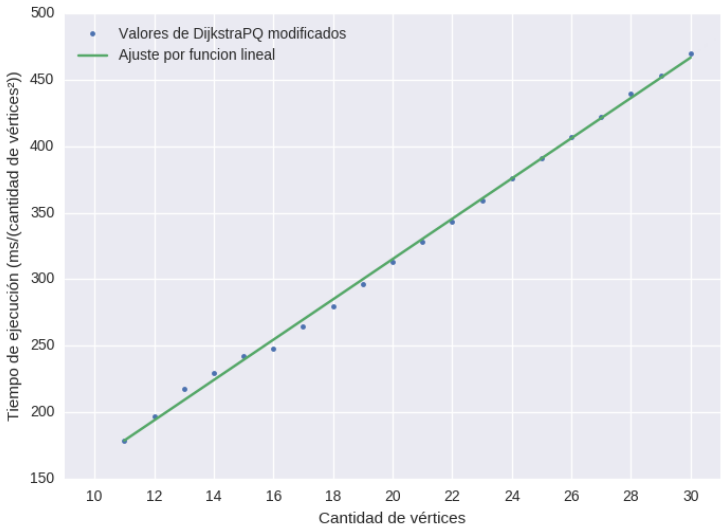

Figura 11: Tiempo de ejecución del algoritmo de Dijkstra con cola de prioridad en función de la cantidad de vértices, escalados por n^2 y ajustados por una recta

Los resultados obtenidos confirman que nuestro cálculo de la complejidad teórica fue correcto, ya que los valores obtenidos se ajustan bien mediante el polinomio de grado 1 obtenido mediantes Cuadrados Mínimos.

3.2.4. Complejidad Bellman-Ford

La complejidad teórica del algoritmo de Bellman-Ford resultó $O(n^2m)$ por lo que, al dividir el tiempo de ejecución por n^2 y graficarlos en función de n debería obtenerse algo similar a una recta. Esto es lo que se ve en la figura 12 y, por lo tanto, puede afirmarse que la complejidad experimental se ajusta a la teórica.

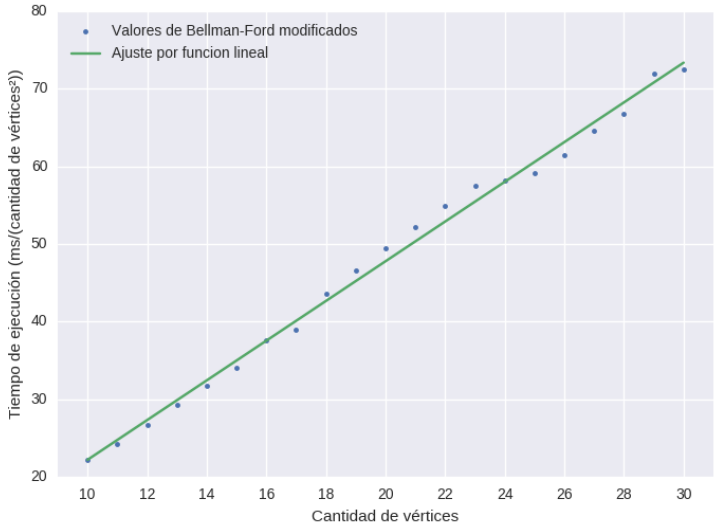

Figura 12: Tiempo de ejecución del algoritmo de Dijkstra en función de la cantidad de vértices. Los tiempos se escalaron por la cantidad de vértices al cuadrado. Se muestra el ajuste de los datos a la recta $y = ax + b$, con $a = 2{,}56$, $b = -3{,}39$ y $R^2 = 0{,}99$. El método utilizado para aproximar los datos fue el de cuadrados mínimos.

3.2.5. Complejidad Floyd-Warshall

Para verificar que la complejidad del algoritmo de Floyd-Warshall utilizaremos una función $f : \mathbb{R} \to \mathbb{R}$ la cual es n^2 para dividir los tiempos de ejecución medidos en n^3 para llevarlos a un orden lineal.

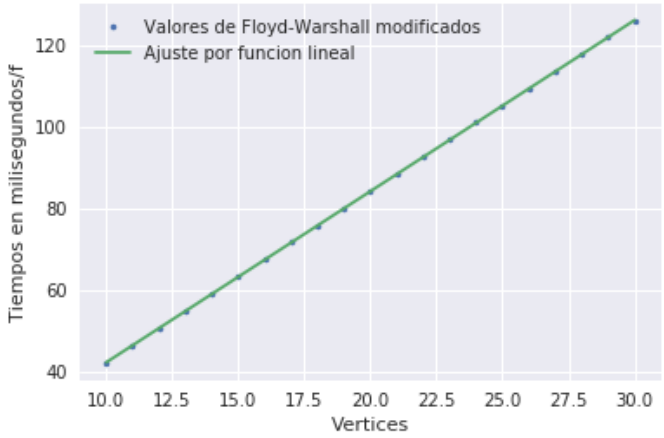

Figura 13: Tiempo de ejecución del algoritmo de Floyd-Warshall en función de la cantidad de vértices, escalados por n^2 y ajustados por una recta

Como se puede ver en el grafico de la figura 13, los resultados obtenidos confirman que nuestro cálculo de la complejidad teórica fue correcto. Los valores obtenidos se ajustan a la recta que representa un polinomio de grado 1, dado que los mismos fueron divididos por nuestra función f.

3.2.6. Complejidad Dantzig

Al igual que en la complejidad de Floyd-Warshall, utilizamos la misma función f para llevar los tiempos de ejecución medidos a un orden lineal.

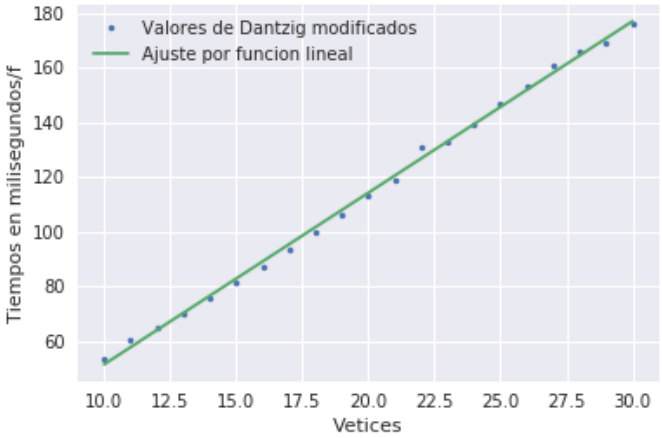

Figura 14: Tiempo de ejecución del algoritmo de Dantzig en función de la cantidad de vértices, escalados por n^2 y ajustados por una recta

Al igual que para los algoritmos anteriores, el gráfico muestra que la recta es un buen ajuste para los valores modificados, con lo cual podemos concluir que nuestro cálculo de la complejidad del algoritmo fue correcto.

Esto concluye con las verificaciones de las complejidades teóricas de cada algoritmo. A continuación, procederemos a comparar los tiempos de ejecución de los distintos algoritmos implementados para resolver el problema. Una aclaración importante al respecto de esto es que no es necesariamente cierto que la comparación entre cualquier par de los algoritmos desarrollados tenga sentido. Por ejemplo, los algoritmos de Dijkstra (en sus dos versiones) y Bellman-Ford son idóneos para resolver la versión del problema de camino mínimo con un único origen y múltiples destinos; A* resuelve la versión del problema para origen y destino únicos, y que Floyd-Warshall y Dantzig resuelven específicamente la versión de múltiples orígenes y múltiples destinos. Por lo tanto, es esperable que, al adaptar los algoritmos con único origen para resolver la versión del problema con múltiples orígenes, su performance sea peor que la de los algoritmos que fueron específicamente pensados para resolver esa versión del problema, por lo cual no haremos comparaciones de performance entre algoritmos que resuelvan versiones distintas del problema de camino mínimo.

Otra aclaración importante es que en las comparaciones que vendrán a continuación, los grafos que usaremos como entrada serán análogos a los utilizados para las verificaciones de complejidad: salvo que específicamente aclaremos lo contrario, definiremos a la cantidad de aristas del grafo como $m = \frac{(n-1)(n-2)}{2} + 1$, para garantizar que las ciudades generadas sean conexas; el costo de la nafta en cada ciudad lo tomaremos aleatorio entre 20 y 100 y el costo en litros de las aristas aleatorio entre 1 y 60.

3.2.7. A* versus Dijkstra

En este experimento compararemos los tiempos de ejecición del algoritmo A* con los de Dijkstra. Para esto, utilizaremos mapas de ciudades que cumplan con la desigualdad triangular y donde el costo del litro de nafta sea el mismo en todas las ciudades, para poder utilizar el primer algoritmo. Para generar de una forma sencilla valores de entrada que cumplan simultáneamente ambas condiciones, nuestra definición consiste en asignarle el valor 60 a todas las aristas generadas (es decir, el viaje directo entre dos ciudades costará 60 litros de nafta si están conectadas), y fijar en 1 el valor del litro de nafta en todas las ciudades. El resto de los valores de entrada del problema se dejarán igual que la definición que hemos dado antes. Con esta definición para los parámetros de entrada, los resultados obtenidos son los siguientes:

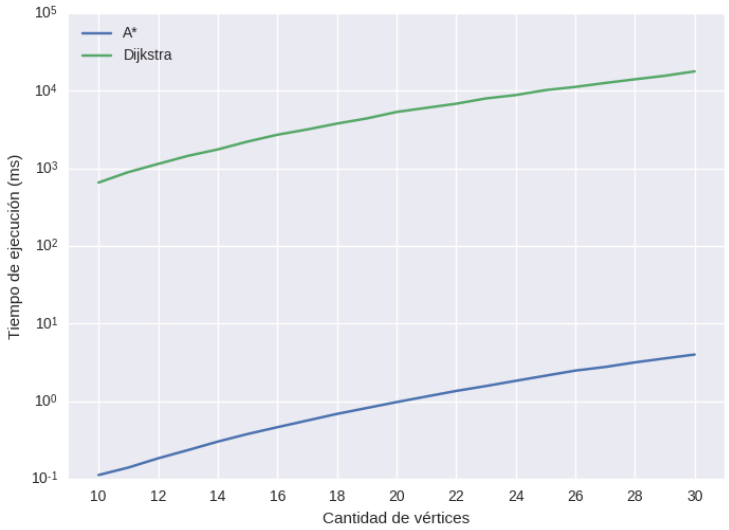

Figura 15: Tiempo de ejecución en función de la cantidad de vértices (graficado en escala logarítmica) de los algoritmos de Dijkstra y A*.

Si bien el algoritmo de Dijkstra tiene complejidad $O(n^3)$ y A* tiene complejidad $O(n^4 + m * n^3)$, lo cual hace que para nuestra configuración de los datos de entrada resulte $O(n^5)$ (pues el valor de m es cuadrático en función de n), se ve claramente que A* resuelve el problema mucho mas rápido que Dijkstra. Este comportamiento era de esperarse, ya que para este experimento consideramos grafos que cumplen con las condiciones necesarias para que A* funcione correctamente. Es decir, que para este tipo de grafos, utilizar A* resulta ser más idóneo que utilizar el algoritmo de Dijkstra.

Antes de continuar con el resto de las comparaciones, haremos una aclaración: siendo que A* representa una generalización del algoritmo de Dijkstra, que además tiene una serie de precondiciones extra sobre el grafo de entrada, hemos decidido que este sea el único experimento donde se utilice A*, ya que las condiciones necesarias para que A* funcione son excesivamente restrictivas y no representarían el funcionamiento general de los otros algoritmos a considerar.

3.2.8. Dijkstra vs DijkstraPQ vs Bellman-Ford

A continuación, realizaremos una comparación entre la performance de los algoritmos de Dijkstra (en sus dos versiones) y el de Bellman-Ford. Para esto, consideramos el caso promedio como lo hemos definido previamente, y los resultados obtenidos fueron los siguientes:

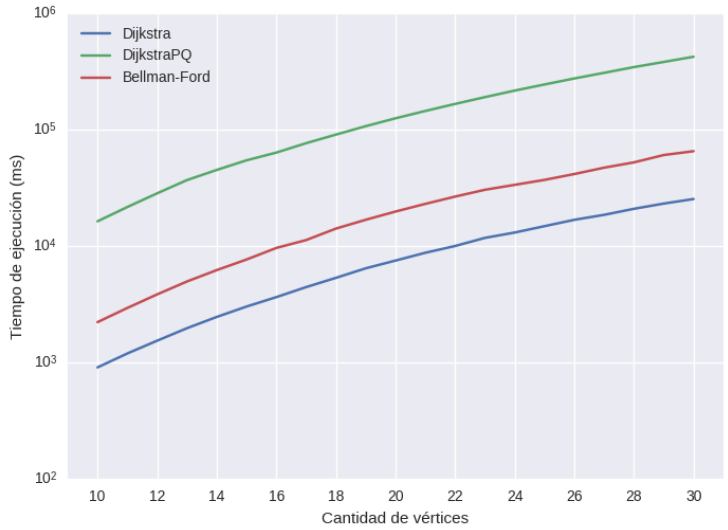

Figura 16: Tiempo de ejecución en función de la cantidad de vértices (graficados en escala logarítmica) de los algoritmos de Dijkstra, Dijkstra con cola de prioridad y Bellman-Ford.

En la figura 16 puede verse que el tiempo de ejecución del algoritmo de Dijkstra es el menor de los tres comparados, seguido por el de Bellman-Ford y, por último, del de Dijkstra utilizando una cola de prioridad.

La diferencia entre Dijkstra y Bellman-Ford puede explicarse por el hecho de que el segundo recorre, en cada iteración, todos los ejes del grafo para actualizar el vector en donde guarda la longitud del camino mínimo hasta cada vértice, mientras que el primero solo recorre los ejes que inciden en el vértice que agrega.

A priori, era de esperarse que, dada la mejora que produce (teóricamente) el uso de una cola de prioridad, el algoritmo de Dijkstra ejecutado en estas condiciones resultó peor que los otros dos. Esto se debe a que la estructura utilizada no admite la modificación de la prioridad de un elemento arbitrario. Debido a esto, no es suficiente con recorrer los vértices adyacentes al que agrego, sino que además es necesario restaurar la propiedad de minHeap de la cola.

3.2.9. Floyd-Warshall vs Dantzig

Al igual que en experimento anterior, compararemos los tiempos de ejecución de los algoritmos Floyd-Warshall y Dantzig. Los resultados obtenidos son los siguientes:

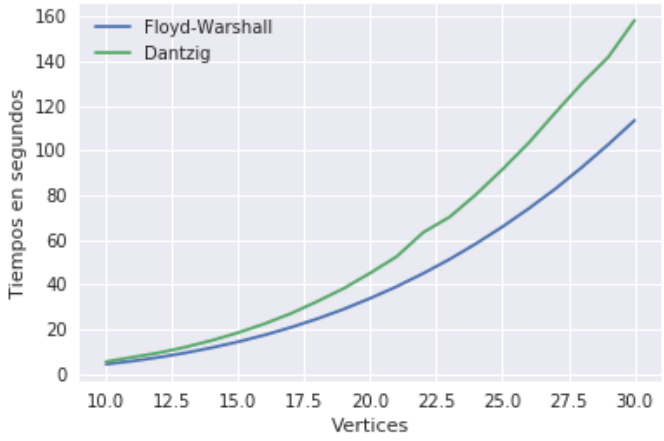

Figura 17: Compraración de tiempos de ejecución entre los algoritmos Floyd-Warshall y Dantzig

Como se comprobó anteriormente, las complejidades de estos dos algorimos son del orden $O(n^3)$ pero en la práctica

puede apreciarse una leve diferencia en tiempos de ejecución. La misma se produce debido a que, en el algoritmo de Dantzig, existen otros ciclos del mismo orden, que al sumarse en el calculo de la complejidad terminan eliminandose (dado que $O(3n) = O(n)$). Estos ciclos hacen que en la comparación de tiempos se note la diferencia y más cuando se incrementa el n.

Gracias a esta experimentación podemos ver que en complejidades iguales, los tiempos de ejecución pueden cambiar. Es aquí cuando se puede ver la diferencia entre una complejidad teorica y los tiempos de ejecución prácticos: cuando se calculan las complejidades se eliminan constantes, las cuales pueden marcar una gran diferencia cuando se trata de iterar 2 o 3 veces un ciclo en lugar de solamente una.

4. Conclusiones

A continuación, daremos un cierre para este trabajo exhibiendo, para cada problema, las conclusiones sacadas de los resultados obtenidos durante la experimentación.

4.1. Hiperconectados

Las comparaciones entre la solución implementada con el algoritmo de Prim y las dos versiones de Kruskal presentan similitudes: sobre grafos muy densos, es decir, con una cantidad de aristas en el orden de $O(n^2)$, el algoritmo de Prim resulta ser más conveniente en la práctica. Esta conclusión tiene cierta lógica: dado que la complejidad del algoritmo de Prim queda expresada únicamente en función de su cantidad de vértices, mientras que las complejidades de las dos versiones de los algoritmos de Kruskal dependen tanto de la cantidad de vértices como de la cantidad de aristas, que el grafo contenga o no una cantidad grande de aristas no debería afectar a la solución con Prim, y sí a las soluciones utilizando alguna de las variantes de Kruskal. Siguiendo esta línea de razonamiento, también pudimos notar una similitud análoga, pero inversa: al llevar la cantidad de aristas a una cantidad lineal en el número de vértices, las dos versiones de Kruskal presentaron mejores tiempos de ejecución que Prim. Esto nos permite concluir que, si se cuenta con más información del contexto del problema a resolver, que permita saber de antemano si el grafo es denso o no, se puede optar por uno u otro método. Además de lo dicho, nuestros experimentos nos permitieron sacar otra conclusión que parecía razonable suponer de antemano: la versión de Kruskal con las técnicas de *Union by Rank* y *Path Compression* presentó mejores tiempos de ejecución que la versión original. Esto no resulta demasiado sorprendente, puesto que la incorporación de las dos heurísticas hace que la complejidad del algoritmo sea estrictamente mejor, con lo cual en cualquier caso donde se opte por Kruskal en vez de Prim (por ejemplo, para un grafo no denso) siempre es más conveniente utilizar la versión con *Union by Rank* y *Path Compression*.

4.2. Hiperauditados

En la comparación entre los algoritmos de Dijkstra y Bellman-Ford, se encontró que, como se esperaba, el primero se comportaba mejor que el segundo para los casos estudiados. Sin embargo, esto no implica que el algoritmo de Dijkstra sea siempre la mejor opción. Por las características particulares del problema que se propuso resolver, los grafos utilizados para modelarlo no poseían aristas de peso negativo. Si las hubiera habido, entonces la única opción de entre estas dos hubiera sido Bellman-Ford, independientemente de que asintóticamente fuera peor.

En cuanto a la modificación del algoritmo Dijkstra utilizando una cola de prioridad, puede verse que no basta con una estructura genérica, sino que es preciso contar con una que provea las funciones adecuadas (en este caso, el poder modificar la prioridad de un elemento arbitrario sin recorrer todos) para obtener una mejora en la complejidad con respecto al algoritmo original.

El algoritmo A* superó en performance al de Dijkstra al ser ejecutados con grafos que satisfacían la desigualdad triangular. Esto tiene sentido, pues se trata de una optimización de Dijkstra pensada específicamente para funcionar mejor en grafos que satisfacen hipótesis más fuertes sobre sus elementos. Si bien no se realizó este experimento, dados los tiempos observados, puede especularse que, para grafos con las características de los utilizados en esta comparación, el algoritmo A* será más eficiente que el de Dijkstra con cola de prioridad y que el de Bellman-Ford.

El tiempo de ejecución del algoritmo de Dantzig resultó mayor al de Floyd, correspondiéndose esto con la diferencia en la cantidad de operaciones, a pesar de tener ambos la misma complejidad. Algo que hubiera podido estudiarse es el tiempo que tarda cada algoritmo en recalcular la matriz de caminos mínimos al agregar un nuevo vértice al grafo. Sería de esperar que, en este caso el algoritmo de Dantzig se comporte mejor que el de Floyd.

Referencias

[1] Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009). *Introduction to Algorithms* (3ra ed.). The MIT Press. McGraw-Hill.