Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

28 de Mayo de 2017

Trabajo Práctico Número 2

Integrante	LU	Correo electrónico
Delger, Agustín	274/14	agusdel_94@hotmail.com
Delmagro, Nicolás Agustín	596/14	agustin.delmagro@gmail.com
Lopez Valiente, Patricio	457/15	patricio454@gmail.com
Zar Abad, Ciro Román	129/15	ciromanzar@gmail.com

Índice

1. Introducción

En este informe presentaremos distintas técnicas comúnmente utilizadas para resolver problemas de optimización en grafos. Estos problemas son muy usuales y ampliamente utilizados en rubros tan diversos como mapas, transporte, robótica y hasta en algo tan abstracto como resolver un cubo Rubik. Esto es posible ya que los grafos como herramienta para representar diversos escenarios son altamente flexibles y amplios, por lo que se convierte en una forma de modelar problemas muy potente.

En las siguientes secciones de este informe, mostraremos cómo es que tratamos los problemas presentados por la cátedra, y cómo los transformamos en problemas equivalentes de grafos, para poder facilitar su solución utilizando las técnicas algorítmicas presentadas en la materia. Las técnicas algorítmicas utilizas fueron las siguientes:

Algoritmos de Camino Mínimo: Estos consisten en encontrar un camino entre dos nodos de tal manera que la suma de los pesos de las aristas que lo constituyen sea mínima. En particular nosotros utilizaremos:

- **Algoritmo de Dijkstra:** Comúnmente llamado algoritmo de caminos mínimos, es un algoritmo eficiente para la determinación del camino más corto dado un nodo origen al resto de los nodos en un grafo con **pesos positivos** en cada arista. Éste se encarga de formar un camino a partir de otro ya existente.

  El algoritmo de Dijkstra trabaja por etapas bajo el principio de optimalidad, tomando en cada etapa la mejor solución sin considerar consecuencias futuras, ya que puede modificarse posteriormente si surge una solución mejor.

- **Algoritmo de Bellman-Ford:** Este algoritmo determina la ruta más corta desde un nodo origen hacia los demás nodos en una grafo con pesos en las aristas. La diferencia de este algoritmo con el de Dijkstra es que los pesos pueden tener valores negativos mientras no formen ciclos negativos. Además, Bellman-Ford permite detectar la existencia de estos ciclos.

Algoritmos de Árbol Generador Mínimo: Estos consisten en encontrar un Árbol Generador Mínimo lo cual es un Árbol Generador (AG) del grafo, es decir, un subgrafo que es árbol y que contiene a todos los nodos del grafo, pero con la característica de ser el AG de menor peso del grafo al cual genera.

Utilizaremos principalmente el **Algoritmo de Kruskal:** Este algoritmo se basa en una propiedad clave de los árboles que permite estar seguros de si una arista debe pertenecer al árbol o no, y usa esta propiedad para seleccionar cada arista. Siempre que se añade una arista (u, v), ésta será la conexión más corta (menor costo) alcanzable desde el nodo u al resto del grafo. Así que, por definición, ésta deberá ser parte del árbol. Este algoritmo es de tipo goloso ya que a cada paso selecciona la arista más barata y la añade al subgrafo.

1.1. Detalles Técnicos

Los experimentos realizados en este informe comprenden mayoritariamente costos computacionales temporales, los cuales son susceptibles a eventos no controlables por nuestro ambiente de ejecución, como por ejemplo el Scheduler del sistema. Para evitar la presencia de estos errores se utilizo el siguiente protocolo:
A la hora de realizar la experimentación de cada problema, se efectuaron numerosas ejecuciones de las mismas instancias y se tomó el promedio de los tiempos de ejecución de cada conjunto de instancias de igual tamaño para evitar la presencia de posibles outliers. Ademas todas estas mediciones se efectuaron en un ambiente controlado de ejecución, con el cual se busco minimizar el impacto de los procesos externos sobre nuestra medición.

2. Problema 1: Delivery Óptimo

2.1. El problema

La empresa de logística Transportex debe transportar mercaderías entre ciudades de una provincia de Optilandia, un país donde todo es óptimo. Para cumplir con la optimalidad, la empresa debe hallar la forma más rápida de trasladarse de una ciudad a otra, sin incumplir con la regulación provincial que le permite utilizar un máximo de k rutas *premium*[1] para realizarlo.

En este problema, se conoce la ciudad de origen, la ciudad de destino y se cuenta con un mapa de la provincia que incluye todas las rutas que conectan ciudades entre sí, junto con sus distancias y si son *premium*. Se desea encontrar la distancia mínima que debe recorrer un camión de Transportex para conectar ambas ciudades, dado que se pueden tomar hasta k rutas *premium*.

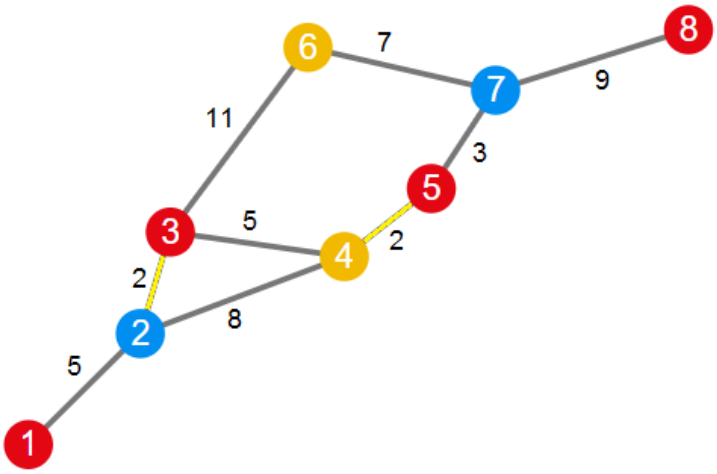

Figura 1: Ejemplo de la provincia de Optilandia donde los caminos amarillos representan rutas *premium*.

Por ejemplo, en la provincia de la figura, si se quiere encontrar la forma más rápida de llegar desde la ciudad 2 a la 7, dependerá de la cantidad de rutas *premium* que se puedan utilizar.

- En caso de que no se permita usar ninguna *premium* ($k = 0$), hay un único camino para llegar de una ciudad a la otra, y ese determina la distancia. Pasando por las ciudades 4, 3 y 6, la distancia de la ciudad 2 a la 7 es 31.
- Si, en cambio, se permite utilizar hasta una ruta *premium* ($k = 1$), los posibles recorridos aumentan. En este caso, el camino más corto es a través de las ciudades 4 y 5, teniendo una distancia de 13.
- Para cualquier valor de k mayor a 1, la distancia es 12, la cual se obtiene al hacer una escala en la ciudad 3 antes de ir a la 4 en el recorrido del caso anterior.

[1] Rutas que se encuentran en mejor estado y utilizan caminos más directos que suelen ser mas cortos que sus alternativas.

2.2. La solución

Debido a que se trata de un problema de camino mínimo, tiene sentido modelarlo mediante grafos, ya que el problema de camino mínimo ha sido estudiado muy profundamente en esta rama de la matemática y ciencias de la computación. Dado el mapa de la provincia, es muy sencillo convertirlo a un grafo, ya que cada ciudad se traduce a un nodo y cada ruta a una arista, siendo la distancia de ésta su peso.

De todos modos, el problema no finaliza acá, ya que si se aplica algún algoritmo de camino mínimo sobre este grafo, éste efectivamente devolverá el camino más corto entre ambas ciudades pero no tendrá en cuenta la cantidad de caminos *premium* utilizados. Esto se debe a que esta restricción no se encuentra modelada en el grafo obtenido.

Para incluir esta restricción, es necesario remodelar el grafo de modo que cada nodo indique la cantidad de caminos *premium* que se utilizaron para llegar a él. El problema es que puede haber muchas formas de llegar a una misma ciudad, y por lo tanto la cantidad de caminos *premium* utilizados para llegar a ella varía según el camino que se tome. Para solucionar esto, se 'clona' cada nodo de modo que haya uno para cada cantidad de caminos *premium* utilizados, quedando como un par (c, v) donde u es el número de la ciudad y v es la cantidad de *premiums* utilizados. Por ejemplo, el par $(1, 0)$ representa la ciudad 1 llegando a ella a través de todas rutas normales. Solo hace falta clonar cada nodo $k+1$ veces ya que, en este problema, se puede utilizar un máximo de k *premiums*.

Una vez realizada la modificación de los nodos, es necesario adaptar las rutas a este nuevo modelo. Para esto, la traducción de rutas en aristas ya no es directa, sino que ahora se tiene en cuenta si ésta es *premium* o no.

- Si la ruta que conecta las ciudades c_i y c_j **no** es *premium*, entonces los nodos (c_i, v) y (c_j, v) estarán conectados entre sí, ya que si se puede llegar hasta la ciudad c_i utilizando v caminos *premium*, de ahí se puede llegar a la ciudad c_j sin aumentar esa cantidad. Sucede lo mismo a la inversa. Esto vale para todo v entre 0 y k.
- Si la ruta que conecta las ciudades c_i y c_j es *premium*, entonces si se puede llegar a la ciudad c_i utilizando v caminos *premium*, a través de ella se puede llegar a la ciudad c_j utilizando $v+1$ *premiums*. De esta forma, el nodo (c_i, v) se conecta con el nodo $(c_j, v+1)$ y ocurre lo mismo a la inversa: (c_j, v) se conecta con $(c_i, v+1)$. Por el mismo razonamiento, los nodos (c_i, v) y (c_j, v) no se encuentran conectados. Esto ocurre para todo v entre 0 y $k-1$, sin incluir k, ya que una vez llegado a un nodo (c, k), por la regulación provincial ya no se pueden tomar más caminos *premium*. Lo única opción en este punto es moverse hacia ciudades conectadas mediante caminos normales.

Ahora sí, el mapa de la provincia se traduce en un grafo que contempla si una ruta es *premium* o no y cada nodo guarda la información de cuántos de éstos fueron utilizados para llegar hasta él independientemente del recorrido que se haga. En este nuevo grafo, dos ciudades están conectadas entre sí únicamente si hay un camino entre ellas que emplea menos de k caminos *premium*.

Para resolver el problema, es decir, encontrar el camino mínimo de una ciudad c_i a otra ciudad c_j cumpliendo la regulación, ahora sí se puede aplicar algún algoritmo de camino mínimo sobre este nuevo grafo. Pero, recordando que hay $k+1$ nodos por cada ciudad, ya no

es tan simple como buscar el camino entre c_i y c_j. La ciudad de origen es ahora el nodo $(c_i, 0)$, ya que se parte de la ciudad c_i sin haber utilizado ningún camino *premium* inicialmente. El nodo de destino, en cambio, es cualquier nodo (c_j, v) con v entre 0 y k. Esto se debe a que, independientemente de la cantidad de *premiums* que se recorran, al llegar a cualquiera de esos nodos se habrá llegado a la ciudad c_j. Para determinar cuál es el camino mínimo entre ambas ciudades, es necesario calcular el mínimo de los caminos mínimos entre el nodo de origen y los $k + 1$ nodos destino. Por como fue construido el grafo, el mínimo de ellos será precisamente el camino mínimo entre ambas ciudades, es decir, la solución al problema.

2.3. La implementación

Para implementar la solución, primero es necesario determinar el algoritmo de camino mínimo a utilizar, teniendo en cuenta lo siguiente:

- Las aristas tienen pesos diferentes y son todas positivas.
- Debe calcular el camino de uno a todos: Por lo mencionado anteriormente, el camino mínimo entre dos ciudades es el mínimo de los caminos mínimos entre el nodo de origen y los $k + 1$ nodos destino.
- Por restricción del enunciado, la complejidad temporal en peor caso para la resolución del problema debe ser $O(n^2k^2)$, y por lo tanto la del algoritmo también, siendo n la cantidad de ciudades de la provincia.

El primer punto descarta algunos algoritmos como BFS[2] que no contemplan los pesos de las aristas. Asimismo, el segundo punto descarta otros que solo calculan la distancia de uno a uno.

Un algoritmo que cumple los primeros dos puntos es el algoritmo de Dijkstra. Además, la complejidad de este algoritmo implementado mediante matriz de adyacencias es $O(v^2)$, donde v es la cantidad de nodos. En nuestro problema, la cantidad de nodos es $n * (k + 1)$ ya que cada ciudad se clona $k + 1$ veces, por lo que aplicando Dijkstra, se sabe de antemano que la complejidad temporal será no mayor a $O((n*(k+1))^2) = O(n^2k^2)$, cumpliendo el tercer punto.

Luego, utilizando el algoritmo mencionado, la solución al problema consiste en 3 pasos:

- Construir la matriz de adyacencias.
- Aplicar el algoritmo de Dijkstra.
- Buscar el mínimo entre los caminos mínimos desde la ciudad origen a todos los 'clones' de la ciudad destino.

2.3.1. Matriz de adyacencias

Usualmente una matriz de adyacencias es una matriz cuadrada de dos dimensiones, donde la primera dimensión indica el nodo de salida y la segunda el nodo de llegada. Así, en la posición (i, j) se guarda el peso de la arista que va desde el nodo i-ésimo al j-ésimo.

[2] Para más información sobre el algoritmo ver https://www.cs.us.es/cursos/cc-2009/material/bfs.pdf

En este problema, como cada nodo no está representado por un número sino por un par (c, v), se decidió utilizar, en cambio, una matriz de 4 dimensiones. Las primeras dos dimensiones indican el nodo de salida y las últimas dos el nodo de llegada. En esta matriz, el peso de la arista que conecta una ciudad c_i dado que se llegó a ella a través de v caminos *premium* (nodo (c_i, v)) hacia otra ciudad c_j a la que se llega a través de la misma cantidad de *premiums* (nodo (c_j, v)) se guarda en la posición (c_i, v, c_j, v). Para almacenar esta información, el tamaño de la matriz es n x $(k+1)$ x n x $(k+1)$.

Para rellenar la matriz de adyacencias con la información provista por el mapa de la provincia, primero se inicializa la matriz con infinito en todas las posiciones. El paso subsecuente se ve en el siguiente pseudocódigo:

```
function RellenarMatrizDeAdyacencias

    para cada ruta r del mapa hacer
        si r es premium entonces
            para j entre 0 y k − 1 hacer
                adyacencias[c1][j][c2][j + 1] ← distancia;
                adyacencias[c2][j][c1][j + 1] ← distancia;
            fin

        en otro caso
            para j entre 0 y k hacer
                adyacencias[c1][j][c2][j] ← distancia;
                adyacencias[c2][j][c1][j] ← distancia;
            fin
        fin
    fin
```

Algoritmo 1: Se rellena la matriz de adyacencias. Cada arista r contiene las ciudades que vincula c_1 y c_2, su distancia y si es premium.

Lo que esta fracción de código realiza es precisamente lo descrito en la solución del problema. Dependiendo de si la ruta es *premium* o no, la adyacencia se establece entre las dos ciudades para el mismo valor de j o de j a $j+1$.

2.3.2. Dijkstra

Debido a que el algoritmo de Dijkstra[3] toma una matriz de adyacencia de la forma usual (dos dimensiones), para este problema fue necesario adaptar el algoritmo a nuestra matriz de 4 dimensiones y a los nodos como un par (c, v).

Para ello, en vez de tenerse un arreglo de tamaño $n * (k+1)$ para ir guardando la distancia desde el nodo de origen hasta el resto, se tiene una matriz de n x $(k+1)$ que cumple el mismo rol, donde la distancia hasta el nodo (i, j) se guarda justamente en la posición (i, j). Lo mismo se hace con el arreglo de nodos ya visitados, utilizándose también una matriz de n x $(k+1)$.

Debido a esta modificación, buscar los vecinos de un nodo ya no implica recorrer una

[3] Para más información sobre el algoritmo de Dijkstra visitar http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

fila de la matriz de adyacencias usual, sino que se requiere recorrer las últimas dos dimensiones de la matriz. De todas formas, por la construcción de la matriz, no es necesario recorrer toda la última dimensión (la que indica la cantidad de rutas *premium* que se utilizaron para llegar hasta la ciudad), ya que si, por ejemplo, se buscan los vecinos de un nodo (c_i, v), éstos solo podrán tener v o $v+1$ en su segunda coordenada. Esto se debe a que al ir de una ciudad a otra, la cantidad de rutas *premium* utilizadas puede aumentar en a lo sumo uno.

function RecorrerVecinos
para i *entre* 0 *y* $n-1$ **hacer**
  para j *entre* v *y* $v+1$ **hacer**
    El par (i, j) es vecino de (u, v) si adyacencias$[u][v][i][j] \neq \infty$;
  fin
fin

Algoritmo 2: Dado un nodo (u, v), se recorren los vecinos.

Es importante aclarar que, independientemente de los cambios, el funcionamiento del algoritmo es el mismo que el del algoritmo de Dijkstra original.

2.3.3. Camino mínimo

Este último paso generalmente no se realiza en problemas de camino mínimo, ya que la distancia de un nodo a otro es única. En el caso de nuestro problema, lo que se busca es la distancia entre ciudades, no entre nodos. Como cada ciudad está representada por más de un nodo, para hallar la distancia entre las ciudades, es necesario encontrar el mínimo entre la ciudad de origen con 0 rutas *premium* utilizadas y todos los nodos que representan a la ciudad de destino. Este último paso se explica más detalladamente en la sección "La solución"del problema.

Realizar este es muy sencillo, ya que una vez aplicado al paso anterior, Dijkstra, solo hay que recorrer de la matriz resultante la fila que contiene la distancia hasta los $k+1$ 'clones' de la ciudad de destino buscando la distancia mínima. Este valor es el resultado del problema.

2.4. La complejidad

Dado que la solución se divide en 3 partes, la complejidad temporal de la solución es la suma de las complejidades de sus partes.

En el primer paso, el de construir la matriz de adyacencias, inicialmente se debe completar la matriz de n x $(k+1)$ x n x $(k+1)$ con infinito. Esto da una complejidad inicial de $O(n^2k^2)$ ya que hay que recorrer el total de las posiciones. Luego, para cada una de las m rutas de la provincia, se generan hasta $2*(k+1)$ aristas, las cuales no implican más de una asignación cada una. Esto agrega una complejidad de $O(m*(k+1)) = O(m*k)$. Esto deja el costo del primer paso en $O(n^2k^2) + O(m*k) = O(n^2k^2 + m*k)$.

En el segundo paso, Dijkstra, el algoritmo hace una iteración por cada nodo para recorrerlos todos $(n*(k+1)$ iteraciones), y en cada una de ellas busca todos sus vecinos para relajar la arista que los conecta[4]. Por lo mencionado en la sección de implementación sobre el

[4] Para más información sobre el algoritmo de Dijkstra visitar http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

algoritmo, no es necesario recorrer todos los nodos, sino únicamente los que tienen igual o una ruta *premium* utilizada más. El costo de esta búsqueda es muy claro al ver el pseudocódigo presentado anteriormente, donde se muestran dos ciclos anidados, donde el primero recorre toda la tercera dimensión de la matriz (n iteraciones) y el ciclo interno únicamente los dos valores posibles para la última dimensión. El costo de la búsqueda de vecinos es $O(n*2) = O(n)$. Como esta búsqueda se realiza una vez por cada nodo, el costo total del algoritmo para este problema es $O(n * k * n) = O(n^2 * k)$.

Finalmente, en el último paso, donde se busca el camino mínimo entre todos los calculados, se comparan las distancias desde el nodo de origen hasta los $k + 1$ 'clones' de la ciudad de destino y se guarda la mínima de ellas. Como este paso implica solamente recorrer una fila de la matriz y en cada paso hacer a lo sumo una comparación y una asignación, el costo es $O(k + 1) = O(k)$.

El costo temporal total de resolver el problema es la suma de todos los costos calculados. Por lo tanto, el costo total es $O(n^2k^2 + m * k) + O(n^2 * k) + O(k) = O(n^2k^2 + m * k + n^2 * k + k) = O(n^2k^2)$.

2.5. Los experimentos

2.5.1. Tiempos de ejecución

En primer lugar quisimos ver si la complejidad $O(n^2k^2)$ calculada para este algoritmo se asemeja a los tiempos que obtuvimos al ejecutarlo. Para ello, corrimos nuestro algoritmo sobre grafos completos de distintos tamaños y medimos el tiempo de ejecución que tuvieron. Dejamos fija la cantidad de caminos *premium* k que se pueden utilizar como 1 ya que su influencia se verá en otro experimento. Como el valor de k se mantiene fijo en 1, la complejidad del algoritmo es $O(n^2)$.

Los resultados obtenidos se muestran en el siguiente gráfico junto a la función x^2 multiplicada por cierta constante que acerca la curva a los resultados, con el fin de apreciar la similitud entre ambos.

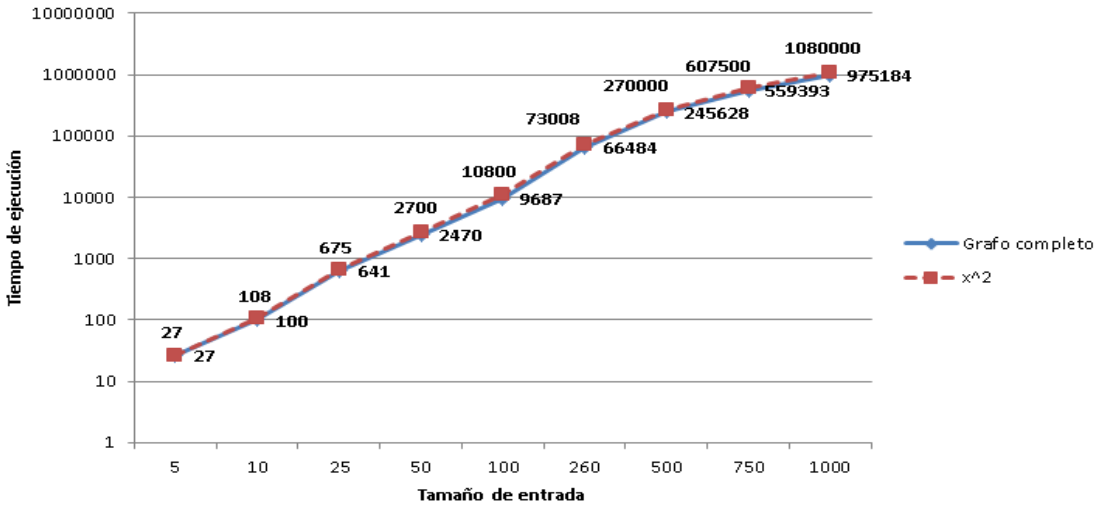

Figura 2: Comparación entre los tiempos del algoritmo y la función x^2

Se puede ver que los gráficos son similares por lo que podemos decir que el algoritmo tiene realmente la complejidad calculada.

Recordando que el costo dominante de nuestra solución se encuentra en crear la matriz de adyacencias, los cual es cuadrático sobre $n * (k + 1)$ debido al tamaño de la matriz, esto nos hizo considerar a ese producto como un único valor $\gamma = n * (k + 1)$. Luego, la complejidad del algoritmo es $O(\gamma^2)$.

Visto de esta forma, se puede esperar que el tiempo de ejecución de la solución dependa de γ, independientemente de los valores de n y $k + 1$ (siempre y cuando su producto sea igual a γ). Esta hipótesis motivó el siguiente experimento.

2.5.2. Variación de caminos *premium*

Dado que la complejidad de nuestro algoritmo es $O(n^2 * (k + 1)^2)$ quisimos ver qué tanto se asemejan los tiempos de ejecución cuando la operación $n^2 \times (k + 1)^2$ da el mismo resultado para distintos valores de n y k.

Para ello, se tomaron distintos valores de n y k de forma tal que la operación $n \times (k+1)$ de siempre como resultado 1024. Se utiliza $k + 1$ dado que el algoritmo en realidad genera matrices de $n \times (k + 1) \times n \times (k + 1)$. Se crearon grafos completos con respecto a estos valores y se midió el tiempo que tomaba nuestro algoritmo en procesar cada entrada.

Suponemos que los tiempos deberían de ser similares ya que la complejidad del algoritmo es $O(n^2k^2)$ y estamos haciendo que la operación $n^2 \times (k + 1)^2$ de siempre como resultado 1024. Estos fueron los resultados obtenidos:

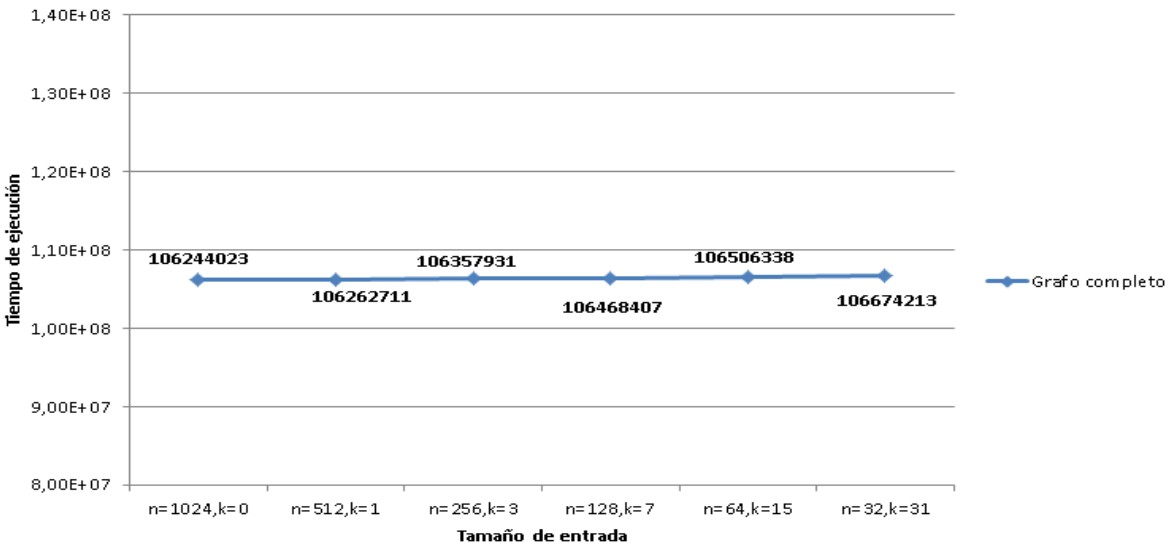

Figura 3: Comparación entre valores de n y k tales que $n^2 \times (k + 1)^2 = 1024$

Se puede ver que, tal como esperábamos, los tiempos de ejecución son muy similares manteniendo el resultado del producto $n * (k + 1)$. De todas formas, mirando más detalladamente, se puede observar que a medida que disminuimos n y aumentamos k, los tiempos crecen levemente. Atribuimos esto a que aumentar el valor de k no solo aumenta el tamaño de 2 dimensiones de nuestra matriz de adyacencias sino que también aumenta el costo de algunas operaciones como la cantidad de posiciones en la que debemos de buscar el camino mínimo al

final o la cantidad de aristas que se generan en el grafo ($O(m * k)$ con m la cantidad de rutas) lo cual depende de k pero no de n.

3. Problema 2: Subsidiando el transporte

3.1. El problema

Nos encontramos nuevamente en Optilandia, el país donde todo es óptimo. En este caso estamos en una provincia donde las rutas son de una sola mano por lo que los vehículos viajan en una sola dirección. No necesariamente es posible viajar de una ciudad a otra de forma directa pero sí es posible llegar desde cualquier ciudad hacia otra ciudad.

Se agregaron cabinas de peaje en cada ruta para financiar el costo de mantener las rutas. A estos peajes se les asignó un costo asociado al largo de la ruta y su tráfico de forma tal que se pueda amortizar el costo de mantener las rutas.

Dado que se logró pagar el costo de todas las obras rápidamente, el gobierno quiere reducir el costo de todos los peajes por un costo fijo c. Esto podría resultar en un problema ya que podrían existir peajes que deban pagarle a los vehículos en lugar de cobrarles. El gobierno no tiene inconvenientes con que esto sea así mientras nadie pueda abusar de la situación. Será considerado un abuso el caso en el que una persona pueda comenzar en una ciudad a y realizar un recorrido que termine nuevamente en a de forma tal que haya ganado plata, pudiendo ser severamente perjudicial para la economía de este país.

El gobernador provincial quiere saber cual es valor máximo por el cual puede reducir el costo de los peajes[5] sin permitir que la gente abuse de su bondad.

3.2. La solución

Como primer paso vamos a modelar este problema mediante un grafo. Siendo cada nodo la representación de una ciudad, las aristas las rutas entre ciudades y considerando sus pesos como los costos de los peajes, tenemos un grafo que se corresponde con el mapa de nuestra provincia de Optilandia. Dado que las rutas son de una sola mano el grafo resultante será un digrafo o grafo dirigido.

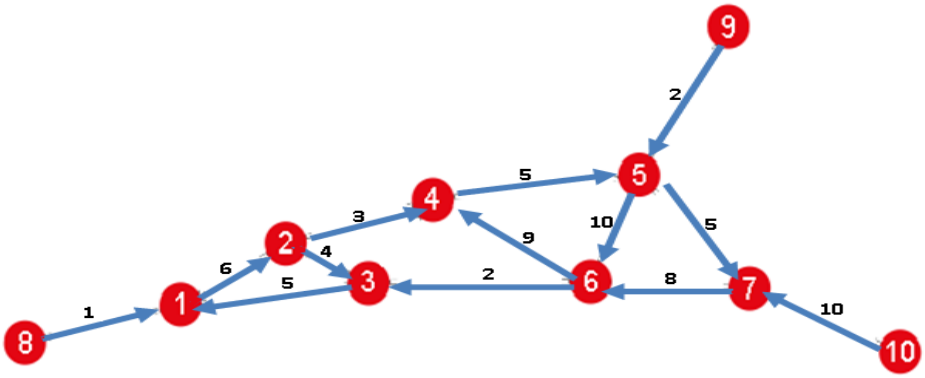

Figura 4: Ejemplo de la provincia de Optilandia donde las rutas son de una sola mano.

[5] Desde nuestro grupo apoyamos esta medida y deseamos que sea ejemplar para otros países. Ver http://www.occovi.gob.ar/www/492/19114/tarifas-vigentes.html

Sabiendo que el enunciado indica que desde cualquier nodo se puede ir a otro nodo nos aseguramos la existencia de por lo menos un ciclo ya que éste es un grafo dirigido y todo nodo se conecta por lo menos con otro nodo.

Vamos a considerar que los pesos de las aristas son positivos cuando se está cobrando un peaje y que son negativos si el peaje le esta pagando a los vehículos. Por lo tanto todas las aristas del grafo original tienen peso positivo.

Debemos traducir qué es **abusar** en término de grafos. El problema lo plantea como partir de una ciudad, hacer un recorrido y volver a la misma ciudad ganando plata, por lo tanto, esto se puede representar como comenzar en un nodo y recorrer aristas de forma tal que se regrese a ese mismo nodo, o sea, formar un ciclo. Se considerará que alguien puede ganar plata si al sumar los pesos de las aristas de un ciclo del grafo el resultado es negativo (se le pagó mas dinero al vehículo que lo que el mismo pagó de peaje en su recorrido), en otras palabras, que se forme un ciclo negativo en el grafo.

Por lo tanto estamos buscando una constante c de forma tal que si le restamos a todas las aristas ese valor, no haya ningún ciclo negativo en el grafo. Nuestra solución debe encontrar el mayor valor que puede tener c de forma que se cumpla que el grafo resultante no posea ciclos negativos, o dicho de otra manera que no se pueda abusar de ese recorrido.

Es importante aclarar que el grafo original lo cumple dado que todos los pesos son positivos.

A lo sumo, c puede tomar el valor de la arista de máximo peso en el grafo ya que si restamos a todas las aristas un valor mayor a este, todas las aristas tendrán peso negativo y en particular las que son parte de un ciclo formarán un ciclo negativo (y sabemos que por lo menos existe un ciclo).

Con este breve análisis, concluimos que el c buscado es un valor que se encuentra entre 0 y el mayor peso de una arista en el grafo c_m.

Básicamente lo que vamos a hacer es una búsqueda binaria entre 0 y c_m dado que sabemos que la solución está entre estos valores y veremos si restando el valor de un c a todas las aristas se forma algún ciclo negativo. Nuestra solución será el máximo valor c que podemos restar a todas las aristas del grafo de forma tal que no se genere ningún ciclo negativo, y por lo tanto no haya ninguna ruta de la cual los usuarios se puedan abusar.

Nos queda ver cómo hacemos para ver si en un grafo hay un ciclo negativo.

No resultaría intuitivo *a priori* aplicar un algoritmo de camino mínimo a nuestro grafo para resolver el problema planteado dado que no estamos buscando ningún camino, sin embargo, vamos a valernos de una propiedad que tienen algunos de estos algoritmos, la cual es que detectan ciclos negativos. Esto lo hacen dado que si hay un ciclo negativo en un posible camino entre nodos, se puede llegar a recorrer ese ciclo infinitas veces para hacer que el costo del camino tienda a $-\infty$, quedando mal definido el problema de encontrar un camino mínimo.

En nuestro caso, utilizaremos el algoritmo de Bellman-Ford[6] que calcula el camino mínimo de un nodo hacia todos y detecta si existe un ciclo negativo alcanzable por el nodo inicial.

El primer problema que se nos plantea es que no está garantizado que desde algún

[6] Para más información sobre este algoritmo ver *https : //es.wikipedia.org/wiki/Algoritmo_de_Bellman−Ford*

nodo se pueda llegar a todos los demás, por lo que dependiendo del nodo elegido como inicial podríamos no estar analizando todos los ciclos del grafo. Para solventar esto, agregaremos un nodo para elegir como inicial que se dirija al resto de los nodos y el peso de sus aristas sea 0 así garantizamos que alcanzamos todos los ciclos del grafo. Vale aclarar que agregar este nodo no modifica la solución dado que ningún nodo se dirige a éste porque no existía previamente, por lo que no formará nuevos ciclos.

Finalmente, nuestro algoritmo consistirá en:

1. Agregar un nodo que se dirija al resto de los nodos.
2. Encontrar cual es el mayor peso de una arista en el grafo que llamaremos c_m.
3. Iniciar la búsqueda binaria entre 0 y c_m.
4. En cada iteración restar a todas las aristas del grafo el valor del c actual y verificar mediante Bellman-Ford si el grafo resultante tiene ciclos negativos.
5. Si no forma ciclos negativos continuamos con la búsqueda binaria hacia valores mayores de c, caso contrario vamos por los menores.
6. Al finalizar la búsqueda binaria tendremos el máximo valor que puede tener c.

3.3. La implementación

Como ya fue determinado anteriormente, este problema puede ser resuelto realizando una búsqueda binaria tomando como criterio de comparación la presencia de ciclos negativos, la implementación y por lo tanto el algoritmo usado para resolver este problema se vuelve casi explícito. Éste se puede dividir en los siguientes pasos:

- Carga de los datos en una estructura acorde a la de un grafo.
- Búsqueda de la arista máxima, con el fin de conocer nuestro valor c inicial.
- Agregar nuestro nodo de partida al grafo de forma que este sea neutral a la solución, pero pueda usarse como punto de partida.
- Búsqueda binaria entre 0 y c, usando como criterio de comparación la presencia de ciclos negativos los cuales son descubiertos aplicando Bellman-Ford.

Si bien el codigo implementado no es exactamente igual al siguiente, cumple con el mismo algoritmo que este. Las sutiles diferencias que presentan, las cuales son mínimas como por ejemplo realizar un paso simple de un ciclo y después realizar el ciclo desde la siguiente iteración, tienen que ver con comodidades a la hora de implementarlo sobre el lenguaje elegido, en este caso *C++*. Por lo tanto, aquí se presentará la forma que tiene nuestra implementación abusando de las bondades y libertades que provee el pseudocódigo, por ejemplo, no se analizarán casos borde ya que estos no hacen a la comprensión del algoritmo.

```
Aristas ← Vacio[m]
para cada arista de la Entrada hacer
|     Aristas[i] ← arista
    fin
```

Algoritmo 3: Procesamiento de datos.

En este paso se recorren las m entradas del texto de entrada y se guardan en el Arreglo de Aristas.

$c \leftarrow PesoMaximo(Aristas)$

Algoritmo 4: Busqueda del maximo(c).

Se busca el máximo de las aristas de manera lineal.

$AristaNueva \leftarrow Vacio[n]$
para i *entre* 0 *y* n **hacer**
  $AristaNueva[i] \leftarrow$ aristaConPeso0Alnodo(i)
fin
$Aristas \leftarrow Union(AristasNueva, Aristas)$

Algoritmo 5: Agregar el nodo(v) que se conecta a todos los nodos originales.

Se agrega el nodo nuevo que posee una arista hacia los n nodos originales.

$loEncontre \leftarrow false$
$cUp \leftarrow c$
$cDown \leftarrow 0$

mientras $cDown \neq cUp \wedge \neg\ loEncontre$ **hacer**
  $AristaAux \leftarrow RestarCaLasAristas(Aristas, c)$
  $SirveC \leftarrow \neg HayCicloNegativo(Aristas, m, n, v)$
  $AristaAux \leftarrow RestarCALasAristas(Aristas, c+1)$
  $SirveCproximo \leftarrow \neg HayCicloNegativo(Aristas, m, n, v)$
  $ActualizarVariablesBusqBinaria(cUp, cDown, SirveC, SirveCproximo)$
fin
$Aristas \leftarrow Union(AristasNueva, Aristas)$

Algoritmo 6: Búsqueda binaria entre 0 y c usando Bellman-Ford como criterio de comparación.

Se realiza la búsqueda binaria sobre c la cual usa como criterio de comparación la presencia de ciclos negativos, es decir, actualiza los índices de la búsqueda binaria en función de la presencia o no de los mismos. La presencia de los mismos se detecta fácilmente corriendo Bellman-Ford normalmente, y por último corriendo una pasada mas: si el peso de las rutas cambia de alguna forma, entonces hay ciclos negativos.

Finalmente, se devuelve el valor de c buscado.

3.4. La complejidad

La complejidad del algoritmo se puede ver calculando la complejidad de sus distintas partes:

- La carga de datos al arreglo es un ciclo que recorre m lineas de la entrada y las asigna al arreglo en $O(1)$ por lo que es $O(m) * O(1) = O(m)$.

- La búsqueda del máximo se realiza de manera lineal en $O(m)$.
- La creación de las n Aristas nuevas es $O(1)$ para cada una de ellas por lo que es $O(n)$ para el total y la unión de los Arreglos de aristas se realiza en el tamaño de la unión, es decir $O(m) + O(n) = O(m)$
- Realizar la búsqueda binaria tiene complejidad $O(\log c)$ y dentro de cada iteración de la misma se ejecuta Bellman-Ford, el cual tiene complejidad $O(m * n)$, ademas la actualización de los indices y demás variables se realiza en $O(1)$. Por lo tanto la complejidad total es: $O(\log c) * (O(m * n) + O(1)) = O(\log c * m * n)$

La complejidad total es la suma de sus partes independientes: $O(m) + O(m) + O(m) + O(\log c * m * n) = O(\log c * m * n)$

3.5. Los experimentos

Basándonos en que nuestra solución tiene una complejidad de $O(n * m * log(c))$, contamos con 3 parámetros que afectan el rendimiento de nuestro algoritmo: la cantidad de nodos n, la cantidad de aristas m y el máximo peso que tiene una arista en nuestro grafo c. Nuestros experimentos consistirán en variar estos parámetros y analizar la forma en que se ve afectado el rendimiento del algoritmo. Además, se pondrán a prueba instancias de grafos particulares que intuimos pueden representar los mejores y peores casos de nuestra solución.

3.5.1. Distintos tipos de digrafos

Dado que los valores de n y m no son independientes ya que la cantidad de nodos limita la cantidad de aristas (un digrafo de n nodos donde cada vértice se dirige por lo menos a otro vértice como lo indica el enunciado tiene como mínimo n aristas y a lo sumo $n * (n - 1)$ aristas). Por lo tanto vamos a analizar 3 tipos de grafos:

- Digrafos completos (fuertemente conexos) donde se utilizan todas las posibles aristas por lo que $m = n * (n - 1)$
- Digrafos que son un único ciclo simple por lo que $m = n$
- Digrafos con una cantidad aleatoria de aristas de forma que $n \leq m \leq n * (n - 1)$

Para este experimento se fijarán los valor de n y c, y se modificará m de forma tal que se corresponda con los 3 tipos de grafos mencionados anteriormente. Se generarán 5 instancias del mismo tamaño para cada grafo, se pasarán a nuestro algoritmo como parámetro midiendo el tiempo que demora en procesar la solución y se tomará el promedio de la soluciones. Luego, se repetirá este proceso mientras se aumenta el tamaño de n, siempre dejando fijo al mismo c. Finalmente, se comparará el rendimiento de nuestro algoritmo para los 3 tipos de grafos generados.

Es intuitivo pensar que el digrafo completo debería presentar siempre mayores tiempos de ejecución que el digrafo con un único ciclo simple dado que presentan los 2 casos bordes de

cantidad de aristas en nuestro digrafo: el digrafo completo tiene la mayor cantidad posible de aristas y el ciclo simple la menor. También sería esperado que los tiempos del digrafo generado aleatoriamente se muevan entre los de los 2 digrafos anteriores.

Estos fueron los resultados obtenidos:

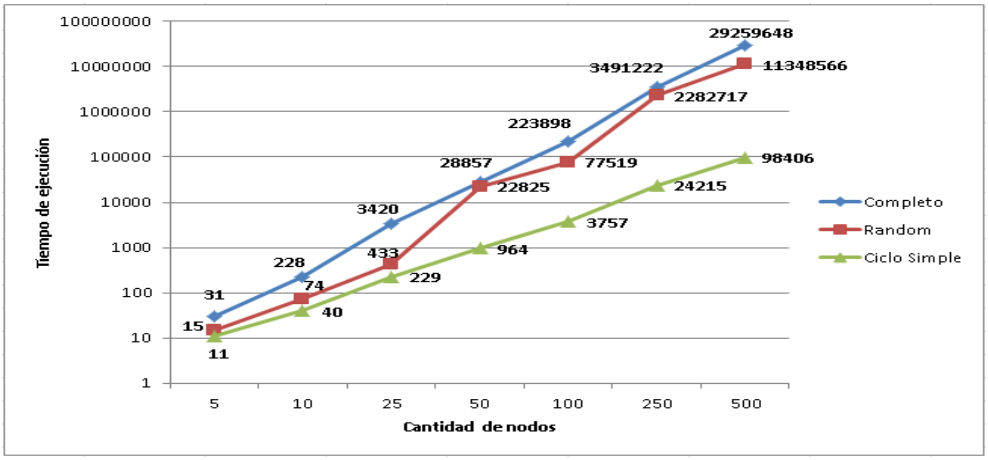

Figura 5: Comparación entre distintos tipos de grafos con $c = 10$

Tal como suponíamos, el tiempo que demora nuestro algoritmo está acotado superiormente por el tiempo de ejecución de un digrafo completo e inferiormente por un ciclo simple, ambos con la misma cantidad de nodos que el digrafo original. Esto se debe a que $O(n^3 * log(c)) \leq O(n * m * log(c)) \leq O(n^2 * log(c))$.

Luego, se corrió el mismo experimento para 2 valores más de c para ver cómo variaba el resultado:

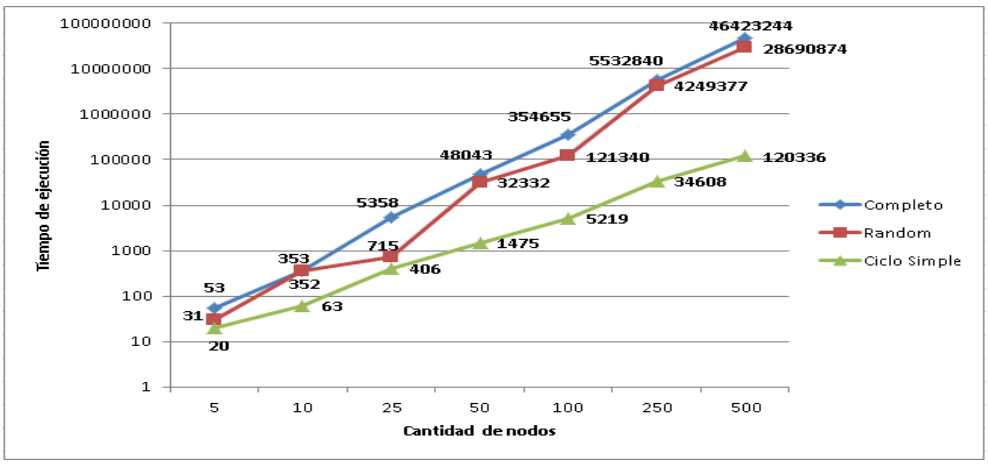

Figura 6: Comparación entre distintos tipos de grafos con $c = 100$

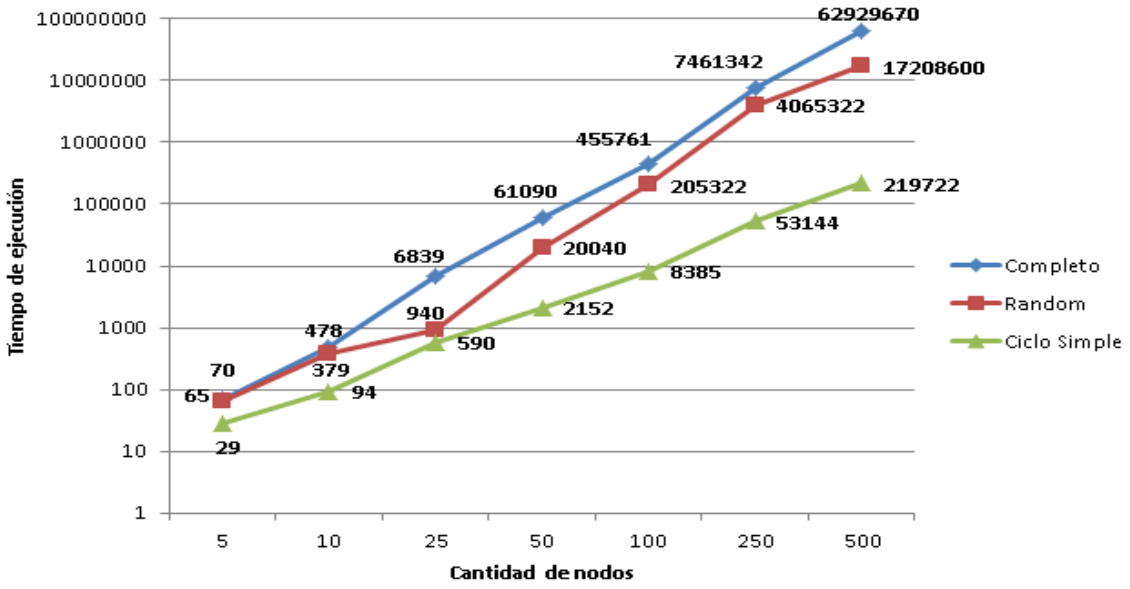

Figura 7: Comparación entre distintos tipos de grafos con $c = 1000$

Se puede ver que la diferencia de tiempos entre los tipos de digrafos se sigue manteniendo, por más que se varíe c. Sin embargo, a medida que crece c aumenta el tiempo de ejecución de nuestro algoritmo. Esto se debe a que aumenta la cantidad de iteraciones que realiza la búsqueda binaria.

3.5.2. Distribución de pesos de las aristas

Ya se analizó como afecta el rendimiento la variación de n y m, ahora se dejarán estos valores fijos y se verá para qué casos nuestro c afecta considerablemente el rendimiento de nuestro programa.

Dado que c es el mayor peso de una arista en nuestro digrafo, éste determinará el valor inicial con que ingresaremos a la búsqueda binaria. Sin embargo, la cantidad de iteraciones de la búsqueda está determinada por el momento en que encuentra la solución, en principio es un valor entre 0 y $log(c)$.

Se realizan 0 iteraciones, es decir que no se ingresa al ciclo, cuando la solución es el mayor peso de una arista en el grafo que es c. Esto sucede únicamente cuando todas las aristas que pertenecen a un ciclo tienen peso c.

Se realizan $log(c)$ iteraciones, por ejemplo, cuando hay por lo menos un ciclo que contiene a una o más aristas de peso c y hay más ciclos donde todas sus aristas tienen peso 1.

En primer término se pusieron a prueba estos mejores y peores casos para grafos completos con un tamaño fijo de 50 nodos y se fue aumentando el valor de c obteniendo los siguientes resultados:

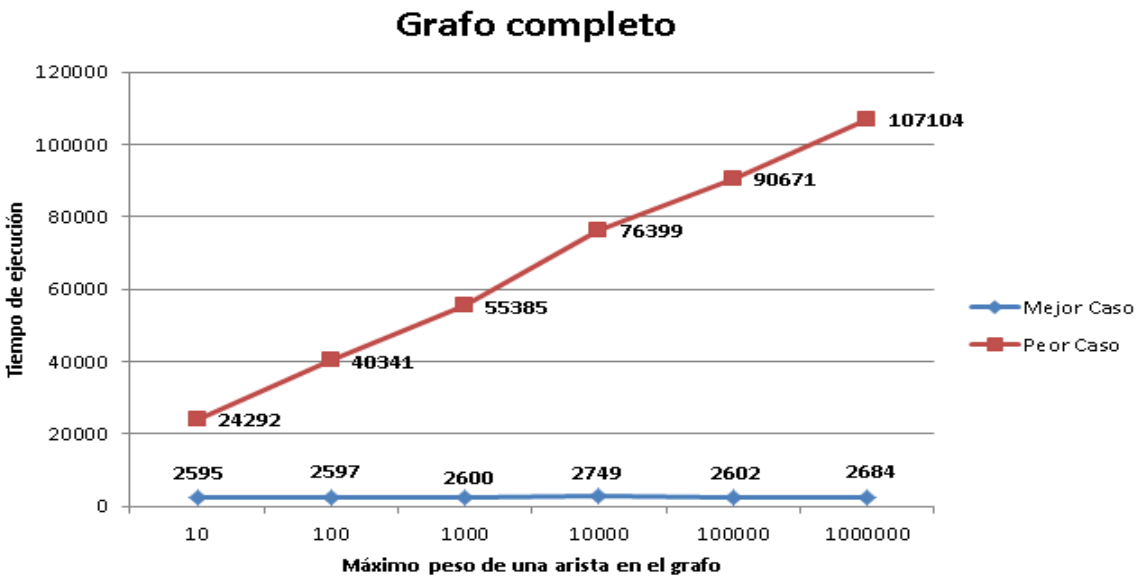

Figura 8: Comparación de mejor y peor caso en digrafo completo

Luego, se realizó la misma prueba para un ciclo simple con todas las aristas con peso c en el mejor caso y con un grafo con 2 ciclos para el peor caso: un ciclo tiene todas las aristas con peso c y el otro ciclo de peso 1. Se obtuvo lo siguiente:

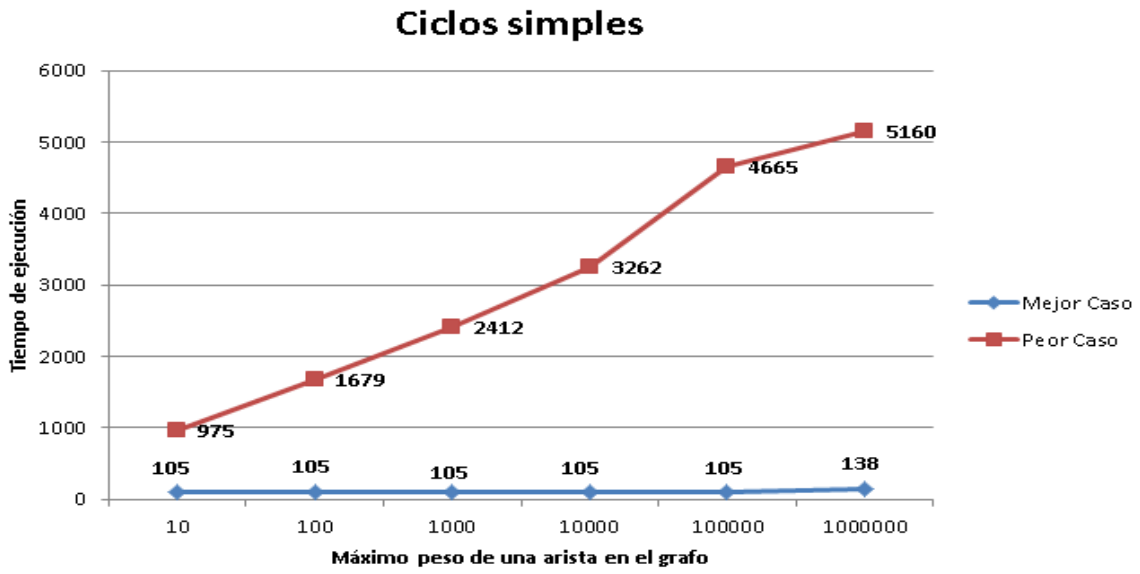

Figura 9: Comparación de mejor y peor caso en digrafo con 2 ciclos simples

En ambos casos se puede ver que para el mejor caso no tiene influencia la variación de c dado que se realizan 0 iteraciones sin importar el valor de c ya que se chequea primero si el valor de c no genera ciclos negativos y al ser todas las aristas iguales esto será cierto. Por esto, es que los tiempos son similares dado que la complejidad quedaría $O(n * m)$. En cambio, para el peor caso se realizan todas las iteraciones posibles dado que se debe llegar hasta 1 que es la solución. Aquí la complejidad es realmente $O(n * m * log(c))$ por lo que al aumentar el valor de c, crecen los tiempos de ejecución.

3.5.3. Complejidad vs tiempo

En este experimento quisimos ver si realmente la complejidad calculada para nuestro algoritmo que es $O(n * m * log(c))$ se correspondía con los tiempos que obteníamos de ejecutar el algoritmo para distintas instancias.

Dado que la complejidad depende de 3 parámetros quisimos simplificar esto para poder comparar el gráfico de los tiempos obtenidos con el gráfico de una función que representara la complejidad $O(n * m * log(c))$. En primer lugar dejamos el valor de c fijo en 10 dado que el $log(c)$ solo afecta para valores más grandes de c. En segundo lugar utilizamos un grafo completo dado que esto nos permite reemplazar m por $n * (n - 1)$. Entonces, la complejidad del algoritmo para esta entrada particular debería de ser $O(n^3 * log(10))$. Fuimos generando grafos completos con aristas de peso random entre 0 y 10 y midiendo el tiempo que demoraba en ejecutarse para distintos tamaños del grafo. Finalmente lo comparamos con el gráfico de la función x^3 multiplicada por una constante que consideramos representante de la complejidad calculada del algoritmo.

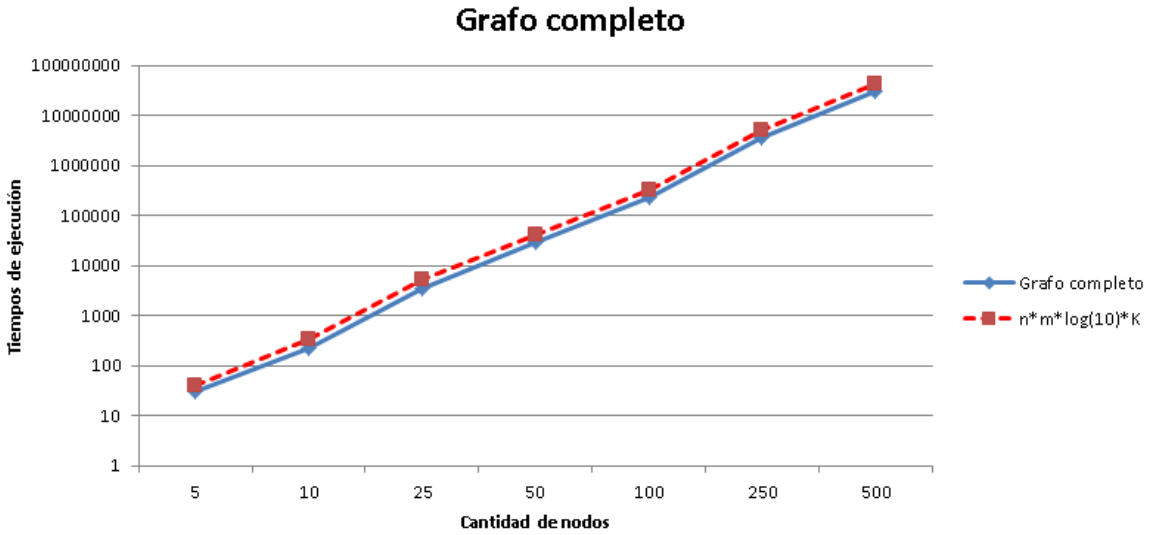

Figura 10: Comparación entre los tiempos del algoritmo y la función x^3

Se puede ver que los gráficos son similares por lo que concluimos que la complejidad del algoritmo calculada previamente se condice con los tiempos de ejecución obtenidos.

4. Problema 3: Reconfiguración de rutas

4.1. El problema

En otra provincia de Optilandia, el gobierno está planificando una reconfiguración de las rutas que conectan las ciudades ya que desde hace mucho tiempo recibe quejas por parte de ciudadanos disconformes: Hay ciudades a las que se puede viajar de una a la otra de varias formas y otras entre las cuales no hay ningún camino.

Para finalmente hacer frente a este problema, el gobernador decidió tomar medidas socialistas: habrá una, y sólo una, forma de llegar desde cualquier ciudad a cualquier otra. Para ello, será necesario construir nuevas rutas y/o destruir rutas existentes. Por supuesto que tanto la construcción como la destrucción de rutas tiene un costo asociado, bastante alto porque todas las rutas, nuevas o existentes, son de doblemano.

Con un mapa de las rutas y los costos de construcción y destrucción, se necesita realizar un plan de trabajo que logre el objetivo del gobernador gastando la menor cantidad de dinero posible: Entre cada par de ciudades debe haber una única ruta (existente o por construir, según el caso) que las conecta directamente.

4.2. La solución

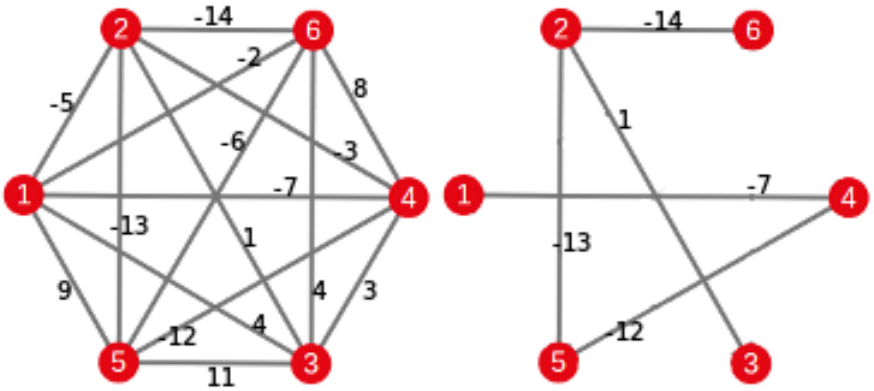

Figura 11: Ejemplo de grafo G a la izquierda, y su AGM a la derecha

El problema pide que todas las ciudades estén conectadas entre si, y que entre cada par de ciudades haya un solo camino. Si modelamos el problema como un grafo G, en el que cada ciudad es un vértice y cada ruta una arista, la solución S debería ser un subgrafo de G con un único camino entre cada par de vértices, es decir un árbol generador.

Como disponemos de los costos de construcción y de destrucción de todas las rutas entre las ciudades, podemos modelar el problema como un grafo completo de n nodos. Dado que queremos minimizar el gasto y las operaciones que conllevan un costo son las de construir una ruta no construida, y destruir una ya construida, queremos que el algoritmo de AGM priorice la elección de rutas ya construidas con costo alto de destrucción y luego las rutas no construidas con costo más bajo. Esto se consigue asignando el peso de cada arista $a = (c1, c2)$ definido por la función $\mathrm{l}(a) : \mathbb{X} \to \mathbb{Z}$:

$$l(a) = \begin{cases} p & : e = true \\ -p & : e = false \end{cases}$$

Donde e es un booleano que indica si la ruta que une las ciudades c1 y c2 ya esta construida, y p es el costo de construcción o destrucción de la misma.

Con el grafo modelado de esta forma, utilizando cualquier algoritmo que devuelva el Árbol Generador Mínimo (AGM) de G habremos encontrado el árbol generador que representa la configuración de las rutas de Optilandia de la solución con el mínimo gasto.

Finalmente, el gasto total g será la suma de las aristas positivas en S, menos la suma de las aristas negativas en $G - S$.

Veamos porque resolver un problema de AGM sobre nuestro grafo, resuelve el problema originalmente planteado, para esto es vital comprender dos puntos:

- α Todas las ciudades alcanzables por rutas construidas originalmente en el mapa resultante que minimice el costo estarán conectadas por rutas Pre Existentes, esto quiere decir que el sub-árbol de las componentes conexas formadas por rutas Pre Existentes no poseerán rutas nuevas.
- β Cualquier ciudad no alcanzable por rutas Pre Existentes, es decir cualquier nodo no perteneciente a una componente conexa, no tiene forma de conectarse en el mapa mediante rutas construibles de forma tal que no cumpliendo el punto anterior se genere un mapa de coste menor, es decir que los nodos no pertenecientes a la componente conexa no se pueden incluir en el AG Solución del problema de forma que haya otro AG de coste menor que se construya a partir de cambiar alguna ruta de las componentes conexas por construibles.

Probados estos dos puntos, se puede probar que el AGM generado por kruskal es solución, el utilizar kruskal no es una decisión al azar, ya que se buscará aprovechar la forma en la que este selecciona las aristas, es decir la menor que no genere un ciclo con las aristas ya seleccionadas previamente. Esto combinado con que las aristas ya construidas presentan peso negativo, hace que éstas sean usadas prioritariamente para conectar la mayor cantidad posible de nodos, y posteriormente se conectaran los nodos no alcanzables con rutas existentes, eligiendo las de menor coste, e iterando nuevamente para así conseguir el AGM buscado.

Probado que las ciudades con rutas existentes conservaran alguna de estas que minimizan el costo y que las no alcanzables usaran caminos simples que minimizan el costo de construir, es decir las mínimas aristas de peso positivo. Se prueba que el AGM es solución, ya que ésta es la forma en que kruskal seleccionara las aristas para construir el mismo seleccionando primero las aristas de rutas construidas ya que son de peso negativo, por lo tanto las menores y luego elegirá las construibles que minimicen el volver conexo el subgrafo.

Veamos α, supongamos que esta no es cierta, entonces quiere decir que hay una forma de construir un mapa de coste menor que el que utiliza solo construidas, es decir las de peso negativo, entre las ciudades de la componente conexa de forma tal que utilizando alguna ruta no construida, es decir de peso positivo, entre alguno de estos nodos. Esto es trivialmente absurdo, ya que basta con tomar cualquier camino de rutas ya construidas entre estos dos nodos que no forme un ciclo, que existe por hipótesis y reemplazarlo por el construido, de esta forma se

obtiene un mapa de coste menor, ya que se destruyen menos rutas y se reduce la construcción de por lo menos una. Y Esto es absurdo ya que el mapa se suponía optimo. Entonces α se cumple.

Veamos ahora β, supongamos también que esto no es cierto, aplicando un razonamiento similar a α podríamos reemplazar la ruta construible por cualquier ruta de construibles entre estos dos nodos que no forme un ciclo, que existe por hipótesis, y obtendríamos un mapa de coste menor con los nodos de la componente conexa de construibles conectados entre ellos por rutas de este tipo. Absurdo. Entonces debe ser β

4.3. La implementación

La implementación se realizó en c++, y el algoritmo elegido para resolver el problema de AGM es Kruskal, que trabaja con lista de aristas con peso como representación de los grafos, por lo que el primer paso es guardar las aristas del grafo completo en una lista de aristas de la forma (v_1, v_2, p).

Luego se aplica el algoritmo de Kruskal, que consiste en ordenar la lista de menor a mayor segun el peso de las aristas e ir agregando aristas a la solución siempre que esta no genere un ciclo.

```
function PROBLEMA3(aristas, n, solucion)

    gasto ← 0, i ← 0 , j ← 0
    init(n)
    ordenar aristas
    mientras j < n − 1 hacer
        si find(aristas_{i_1}) ≠ find(aristas_{i_2}) entonces
            Agregar arista a la solucion
            union(aristas_{i_1}, aristas_{i_2})
            si aristas_{i_3} > 0 entonces
                gasto ← gasto + aristas_{i_3}
            fin
            j ← j + 1
        fin
        si no, si aristas_{i_3} < 0 entonces
            gasto ← gasto - aristas_{i_3}
        fin
        i ← i + 1
    fin
    devolver gasto
```

En este pseudocódigo se pueden observar tres funciones *init*, *find* y *union*. La primera inicializa dos arreglos altura y padre de tamaño n, que representa n conjuntos cada uno con un vértice no conectados entre si.

La segunda devuelve el representante de un conjunto al que pertenece determinada arista, de forma que se pueda determinar si la arista conecta dos vértices de una misma componente conexa o de dos componentes conexas distintas.

La tercera, une dos conjuntos para representar que todos sus elementos pertenecen a

una misma componente conexa.

```
function INIT(n)
para i entre 1 y n hacer
    altura_i ← 1;
    padre_i ← i;
fin

function FIND(x)
si padre_x ≠ x entonces
    padre_x ← find(padre_x);
fin
devolver padre_x

function UNION(x, y) x ← find(x) , y ← find(y);
si altura_x <altura_y entonces
    padre_x ← y;
fin
en otro caso
    padre_y ← x;
fin
si altura_x = altura_y entonces
    altura_x ← altura_x + 1;
fin
```

4.4. La complejidad

El Algoritmo consta de dos pasos:

- Crear la representación del grafo, el cual en nuestro caso es completo, por lo cual la cantidad de aristas m es $\frac{n*(n-1)}{2}$. Además por propiedad sabemos que $m < n^2$.
  Para representarlo usamos un vector de aristas, por lo que su complejidad es del orden del tamaño del mismo, por lo cual es $O(m)$, aplicando nuestra cota: $O(n^2)$.
- Crear un AGM sobre nuestro grafo usando el Algoritmo de Kruskal, el cual tiene como complejidad $O(m * \log n)$, aplicando nuestra cota: $O(n^2 * \log n)$.

Entonces la complejidad final queda determinada por la suma de estos dos pasos: $O(n^2) + O(n^2 * \log n) = O(n^2 * \log n)$.

4.5. Los experimentos

Dado que este algoritmo trabaja con grafos completos, la cantidad de aristas m es siempre igual a $n * (n-1)/2$, por lo que la única variable para experimentar es n.

4.5.1. Experimento de Tiempo

La primera experimentación consiste en generar un caso de n ciudades, y todas las rutas con costo de construcción y destrucción pseudo aleatorio entre 0 y 99, usando la función $rand()$ en c++. Con la misma función determinamos con 50 % de probabilidades si la ruta esta construida o no. Luego resolvemos el problema midiendo 5 veces el tiempo de cómputo con la librería chrono de c++, y calculando la media. Todo esto variando el n entre 6 y 1000.

En la figura 12 se expone el resultado de las mediciones de tiempo, comparadas con una función $n^2 * log(n)$ multiplicada por alguna constante C. Se puede observar que efectivamente los gráficos se parecen.

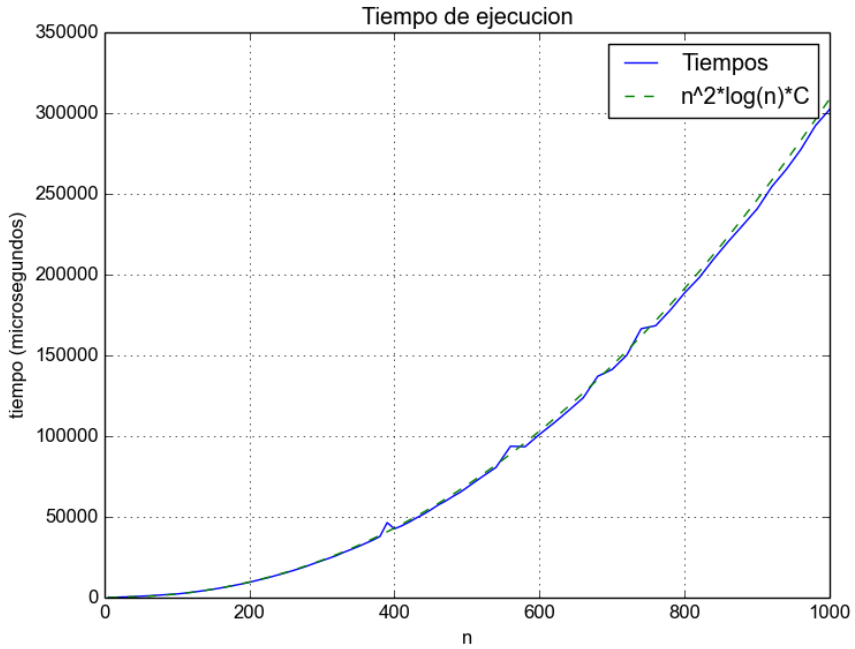

Figura 12: Tiempos de cómputo del algoritmo, comparado con una función de orden $n^2 * log(n)$

4.5.2. Mejor caso vs peor caso

Para este experimento vamos a comparar los tiempos de cómputo para un tipo de entrada que por sus características debería hacer que el algoritmo realice menos operaciones, contra uno que representaría el peor caso. Dado que la complejidad del algoritmo esta determinada desde un principio por ordenar las aristas por su peso, y el grafo es siempre completo, para cualquier tipo de entrada de un mismo tamaño n esta primera operación llevaría el mismo tiempo, la única variación que se puede dar entre casos distintos es a la hora de recorrer las aristas.

Mejor caso: Se creó una instancia de entrada en la que las ciudades ya se encuentran conectadas entre si una a una de una única forma. Particularmente habiendo una ciudad c_0 que está conectada a todas las demás $n-1$ ciudades, y ninguna ruta construida que no pase por c_0. De esta forma al ordenar las aristas, estas quedarán primeras, y como no forman ningún ciclo serán elegidas por el algoritmo para la solución. De esta forma el ciclo recorrerá siempre $n-1$ iteraciones.

Peor caso: Se creó una instancia de entrada en la que todas las rutas y sus costos se generan aleatoriamente, a excepción de las $n-1$ rutas que conectan a una determinada ciudad c_0. Éstas están todas sin construir, y tienen un costo de construcción mayor a todas las demás rutas. De esta forma, al ordenar las aristas las que son incidentes al vértice v_0 que representa la ciudad c_0 quedarán últimas, pero hasta no agregar alguna a la solución, v_0 quedará aislado y el algoritmo no terminará. Por lo tanto el ciclo realizará siempre $n*(n-1)-(n-2)$ iteraciones.

Al igual que en el experimento anterior los tiempos fueron medidos 5 veces con con la librería chrono de c++, y calculando la media. Todo esto variando el n entre 6 y 1000.

Como resaltamos antes, es de esperar que los tiempos de cómputo no varíen significativamente entre un caso y otro y eso se ve reflejado en la figura 13, donde se observa que los tiempos para los peores casos se encuentran siempre levemente por encima que los mejores casos, pero manteniendo una curva similar.

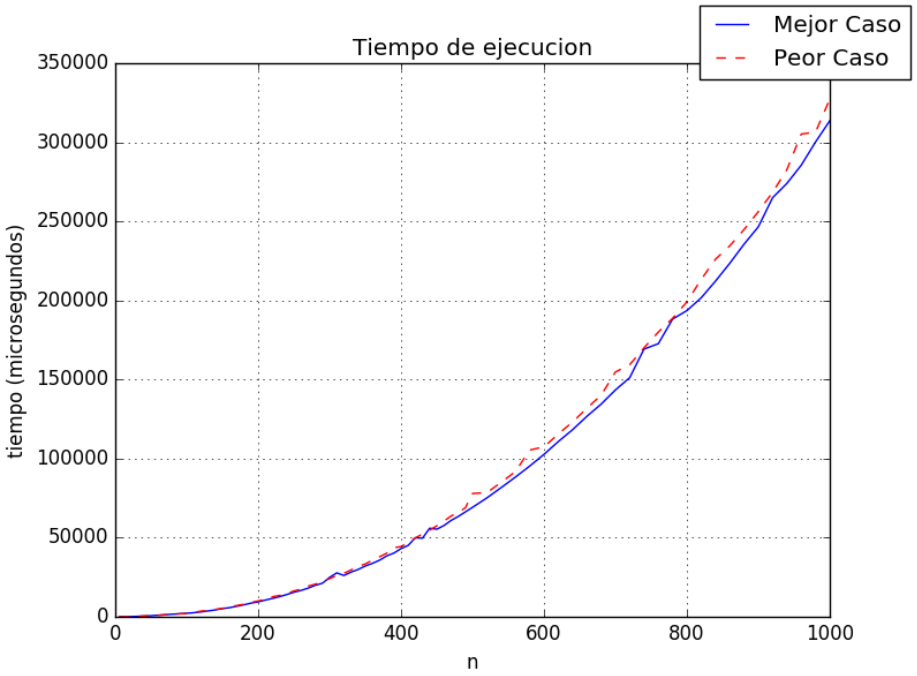

Figura 13: Tiempos de cómputo del algoritmo según n, para mejor caso y peor caso.

5. Conclusión

Este trabajo nos permitió descubrir que la aplicación de grafos para la resolución de problemas no está limitada únicamente a la resolución de los algoritmos típicos como la optimización en el recorrido de caminos, sino que también es posible aplicarlo a soluciones menos intuitivas como puede ser la administración de proyectos utilizando técnicas como la revisión y evaluación de programas (PERT), la construcción conceptual de redes sociales para su estudio teórico así como problemas de genética relacionados con cadenas de ADN.

El mayor desafío se encuentra en transformar el problema planteado de forma tal que podamos representarlo como un grafo, luego podemos utilizar los algoritmos que conocemos aplicables sobre grafos para poder llegar a una solución.

También se deben de buscar los algoritmos que se adapten mejor a nuestro problema para poder obtener una solución con una mejor complejidad. Por ejemplo, en el Problema 1 era igualmente válido aplicar Bellman-Ford para hallar los caminos mínimos pero hubiéramos obtenido una complejidad superior.

Es importante conocer distintos algoritmos aplicables sobre grafos que resuelvan los mismos problemas o similares de forma que tengamos una amplia gama de elección a la hora de seleccionar el algoritmo que mejor se adapte a nuestro problema para poder obtener una solución óptima.