Algoritmos y Estructuras de Datos III

Trabajo Práctico N°2

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

Integrante	LU	Correo electrónico
Izcovich, Sabrina	550/11	sizcovich@gmail.com
Garcia Marset, Matias	356/11	matiasgarciamarset@gmail.com
Orellana, Ignacio	229/11	nacho@foxdev.com.ar
Vita, Sebastián	149/11	sebastian_vita@yahoo.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

Este trabajo práctico consiste en la resolución de ciertos problemas algorítmicos que cumplen con restricciones impuestas por la cátedra, como por ejemplo el orden de complejidad máximo de los mismos, entre otras. Para justificar las implementaciones de los problemas en cuestión, fue necesaria la utilización de herramientas lógico-matemáticas que serán mencionadas a lo largo del desarrollo de cada ejercicio.
Para comprobar que nuestras soluciones resolvieran correctamente los problemas propuestos, debimos dividir el análisis de los mismos en secciones a fin de estudiar minuciosamente las características de éstos. Estas secciones se dividen de la siguiente forma:

- **Problema a resolver:** En esta sección, nos encargamos de describir detalladamente el problema a resolver dando ejemplos del mismo y sus soluciones.
- **Resolución coloquial:** En esta parte, nos dedicamos a explicar de forma clara, sencilla, estructurada y concisa las ideas desarrolladas para la resolución del problema en cuestión. Para ello, decidimos utilizar pseudocódigo y lenguaje coloquial combinando ambas herramientas de manera adecuada.
- **Demostración de correctitud:** Utilizamos este apartado para justificar que el punto anterior resuelve efectivamente el problema y demostramos formalmente la correctitud del mismo.
- **Complejidad del algoritmo:** En esta sección, nos ocupamos de deducir una cota de complejidad temporal del algoritmo propuesto en función de los parámetros considerados como correctos. Por otro lado, justificamos por qué el algoritmo desarrollado para la resolución del problema cumple con la cota dada.
- **Instancias posibles:** Este sector presenta un conjunto de instancias que permiten verificar la correctitud del programa implementado cubriendo todos los casos posibles y justificando la elección de los mismos. Dichas instancias fueron evaluadas por el algoritmo realizado y los resultados obtenidos fueron comprobados.
- **Experimentación:** Por último, los tests consistieron en experimentaciones computacionales utilizadas para medir la performance del programa implementado. Para ello, debimos preparar un conjunto de casos de test que permitieran observar los tiempos de ejecución en función de los parámetros de entrada que fueran relevantes. Para ello, nos encargamos de generar instancias aleatorias como también particulares. Para que los resultados fueran visibles y claros, utilizamos una comparación gráfica entre los tiempos medidos y la complejidad teórica calculada.
- **Código fuente:** En este apéndice, presentamos las funciones relevantes del código fuente que implementa la solución propuesta. Para ello, decidimos utilizar el lenguaje `C++` dado que éste cuenta con la librería *stl* que proporciona las estructuras necesarias para la realización de dicha tarea.

2. Ejercicio 1

2.1. Problema a resolver

El siguiente problema consiste en hallar un algoritmo capaz de dividir un conjunto de tareas ordenadas de forma tal que la realización de las mismas tenga costo total mínimo. El problema se ubica en una imprenta que cuenta con dos máquinas capaces de realizar distintas tareas de impresión. Estas máquinas deben ser preparadas de acuerdo al trabajo que vayan a realizar en ese momento. Dicha preparación requiere de un costo específico que depende del trabajo a realizar y del último ejecutado en esa máquina.

Es importante considerar que los trabajos, t_1,...,t_n, pueden ser asignados a cualquiera de las dos máquinas, pero siempre respetando el orden en el que ingresan.

Los datos otorgados consisten en los costos para preparar una máquina para el trabajo t_i si antes se elaboró t_j, para cada par t_i y t_j, con $j < i$. Por otro lado, se conoce el costo de preparar una máquina para cada trabajo al iniciar el día. Debe tenerse en cuenta que los costos son enteros positivos.

El objetivo del problema consiste, entonces, en tomar la información de los trabajos a realizar y que el algoritmo generado sea capaz de indicar en qué máquina debe realizarse cada uno de ellos para que la suma de los costos de preparación de los mismos sea mínima.

Formatos de entrada y salida:

La entrada contiene varias instancias del problema. La primera línea de cada instancia contiene un entero positivo n que indica la cantidad de trabajos $(t_1, ..., t_n)$ a organizar. A esta línea le siguen n líneas donde, en cada una de ellas, se indican los costos de preparación de cada trabajo. La i-ésima línea consta de los siguientes valores:

$$c_{0i}\ c_{1i}\ \ldots\ c_{(i-1)i}$$

donde c_{0i} es el costo de preparación de una máquina para t_i si t_i es el primer trabajo a realizar en esa máquina y c_{ji} es el costo de preparación de una máquina para t_i si t_j es el último trabajo realizado en esa máquina. La entrada concluye con una línea comenzada por 0.

La salida debe contener una línea por cada instancia de entrada, con el siguiente formato:

$$C\ k\ i_1\ \ldots\ i_k$$

donde C es el costo total de la solución, k es la cantidad de trabajos asociados a una de las máquinas y los valores $i_1, ..., i_k$ son los índices de los trabajos asignados a esa máquina.

En lo que sigue, presentaremos dos ejemplos del problema a resolver:

- **Ejemplo 1:**

  Para este ejemplo, decidimos elegir un caso típico del problema. En éste, puede observarse que, a pesar de que existan costos menores a los elegidos como parte de la solución, el costo final va a ser menor que si éstos hubiesen sido seleccionado. Esto nos permite afirmar que recorrer de forma ordenada los trabajos a realizar no garantiza una solución correcta del problema. A partir de esto, descartamos diseños algorítmicos, como *algoritmos golosos*, para idear la resolución del problema.

Formato de entrada:

5

1

3 2

2 4 6

6 7 4 2

6 9 3 5 8

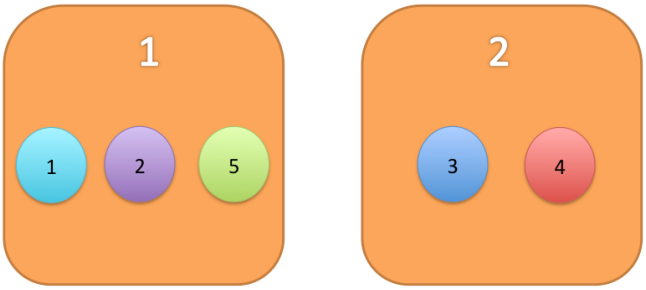


Figura 1: Ejemplo 1.

Formato de salida:

10 3 1 2 5

- ## Ejemplo 2:

Al igual que en el caso anterior, decidimos mostrar un caso en el que resultara extremadamente importante verificar todas las opciones posibles antes de tomar algún tipo de decisión sobre el armado de la solución al problema. En este caso, hasta no llegar al costo del último trabajo, cualquier decisión previa resulta inútil.

Formato de entrada:

5

3

3 2

2 4 5

6 7 4 3

1 15 35 23 4

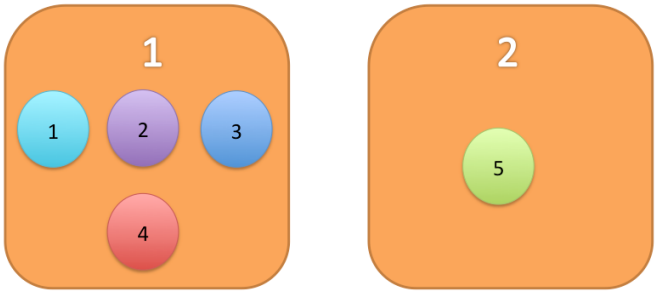


Figura 2: Ejemplo 2.

Formato de salida:

14 4 1 2 3 4

2.2. Resolución coloquial

Para resolver el problema descrito, consideramos armar todas las permutaciones posibles de los trabajos en las dos máquinas disponibles para realizarlos. Dado que contamos con dos máquinas y n trabajos, las posibles combinaciones son 2^n, siendo esta opción altamente costosa. Es entonces que decidimos resolver el problema utilizando *programación dinámica*.

La *programación dinámica*, al igual que el método de Divide & Conquer, resuelve problemas combinando las soluciones de sus subproblemas. A diferencia de este último, la *programación dinámica* resuelve cada subproblema una única vez y almacena su solución en una tabla, evitando el trabajo de recomputar la respuesta cuando éste haya sido evaluado previamente.

Luego, decidimos que nuestro programa almacenara las subsoluciones en una matriz del siguiente modo:

		Máquina 1				
Máquina 2	**i/j**	**0**	**1**	...	**(n-1)**	**n**
	0	x_{00}	x_{01}	...	$x_{0(n-1)}$	x_{0n}
	1	x_{10}	/	...	$x_{1(n-1)}$	x_{1n}
	...	...	...	...	...	...
	n	x_{n0}	x_{n1}	...	$x_{n(n-1)}$	/

Figura 3: Matriz de almacenamiento de subproblemas calculados.

donde x_{nm} representa el menor costo de realizar el trabajo $max(n, m)+1$ si la Máquina 1 realizó el trabajo n y la 2 el m previamente. Por otro lado, '/' representa que dicha casilla no puede ser completada dado que $i = j$, y un mismo trabajo no puede haberse realizado en ambas máquinas en una ejecución. El valor 0 representa el estado inicial de cada máquina.

Por lo tanto, los casos que contempla nuestro algoritmo son de la siguiente forma:

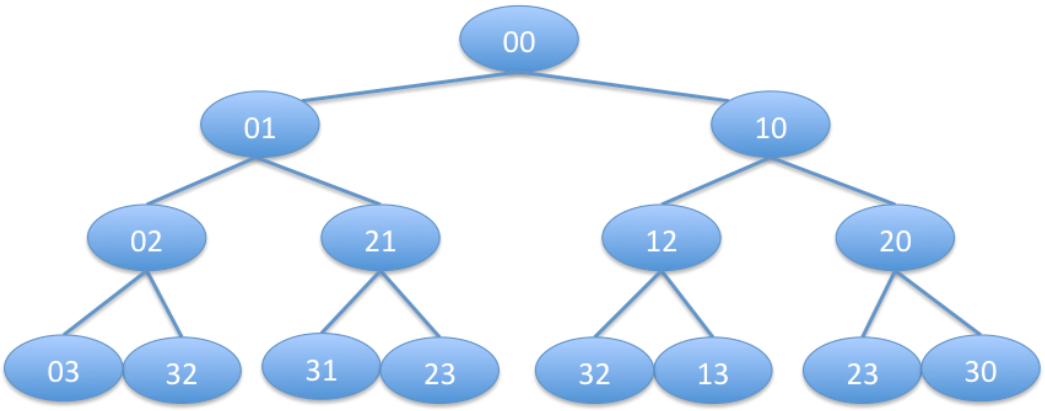


Figura 4: Árbol de decisiones del modelado del problema.

Prueba con cantidad de trabajos = 3.

Donde en cada 

, m representa al último trabajo realizado en la máquina 1 y n al realizado en la máquina 2. El estado 

representa el estado inicial del problema, en el que no hay trabajos. Como puede observarse, se repiten una gran cantidad de casos dado que las máquinas son indistinguibles entre sí, luego 

$=$ 

para cada $i \neq j$. Por lo tanto, recalcular los costos resulta, además de altamente costoso, innecesario, siendo la *programación dinámica* un método adecuado para la realización del algoritmo.

El pseudocódigo que resuelve el problema es el siguiente:

Algorithm 1: MinimizacióndeCostosdeTareas

Input: Entero $m1$, Entero $m2$, Entero $cantidadDeTrabajos$, Costos cs, Matriz $valores$
Output: Costo c

```
1  Entero siguienteTrabajo = max(m1, m2) + 1
2  if siguienteTrabajo > cantidadDeTrabajos then
3  |  c = 0
4  end
5  if valores[m1][m2] > −1 then
6  |  c = valores[m1][m2]
7  else
8  |  Entero costo1 = cs[siguienteTrabajo-1][m1] +
9  |  MinimizacióndeCostosdeTareas(siguienteTrabajo,m2)
10 |  Entero costo2 = cs[siguienteTrabajo-1][m2] +
11 |  MinimizacióndeCostosdeTareas(m1, siguienteTrabajo)
12 |  if costo1 < costo2 then
13 |  |  valores[m1][m2] = costo1
14 |  |  valores[m2][m1] = costo1
15 |  |  c = costo1
16 |  else
17 |  |  valores[m1][m2] = costo2
18 |  |  valores[m2][m1] = costo2
19 |  |  c = costo2
20 |  end
21 end
22 devolver c
```

Algorithm 2: inicializarMinimizacionDeCostos

Input: Entero $cantidadDeTrabajos$
Output: Costo c, Entero $cantidadAsociada$, $listaIndices$

```
1  if cs = ∅ then devolver 0, 0, ListaVacia
2  Entero costoMinimo = MinimizacióndeCostosdeTareas(0,0)
3  Lista maquina1
4  for Entero i=1 a cantidadDeTrabajos do
5  |  if valores[i][m2] < valores[m1][i] then
6  |  |  m1 = i
7  |  else
8  |  |  m2 = i
9  |  |  maquina1 ← i
10 |  end
11 end
12 devolver costoMinimo, |maquina1|, maquina1
```

Donde *Costos* es una matriz que contiene los costos de poner un trabajo i en una máquina si previamente se realizó el trabajo j en ella misma, con $j < i$. Por otro lado, $maquina1$ consiste en una lista de los trabajos que realiza la máquina a la que se le inserta el primer trabajo.

2.3. Demostración de correctitud

Veamos que el algoritmo realizado es correcto. Empecemos por demostrar que cada subproblema es óptimo. Para ello, consideremos que cada uno de éstos consiste en determinar en qué máquina debe ubicarse el trabajo x si en la 1 se encuentra el i y en la 2 el j, con $x = max(i, j) + 1$, tal que el costo sea mínimo. Esto significa que, dadas dos opciones posibles, nuestro algoritmo calcula ambas y selecciona la de menor costo en cada subproblema. De este modo, nuestro algoritmo recorre todas las secuencias de decisiones almacenando las subsoluciones óptimas.

Luego, demostremos que nuestro algoritmo cumple con el Principio de optimalidad de Bellman para corroborar su correctitud.[1] Para ello, supongamos que éste devuelve la solución óptima S. Dicha solución se encuentra conformada por todas subsoluciones óptimas salvo una, x_i. Luego, reemplacemos x_i por la solución óptima de su subproblema correspondiente[2], llamemos S^* a esta nueva solución. Pero entonces, S^* tiene un costo final menor que S. Esto resulta absurdo pues supusimos que S era solución óptima. Luego, todas las subsoluciones de S son óptimas implicando que la solución final también lo es. Entonces, queda demostrado que nuestro algoritmo cumple con el Principio de optimalidad de Bellman confirmando su optimalidad.

2.4. Complejidad del algoritmo

Veamos que nuestro agoritmo cumple con la complejidad requerida. Para ello, llamemos n a la cantidad total de trabajos a realizar:

- En primer lugar, la función **inicializarMinimizacionDeCostos** inicializa la matriz *valores* con una dimensión de n^*n y aplica una única vez la función **resize** $(\mathcal{O}(n))^3$. Dicha matriz se encuentra representada por un vector de vectores cuyo espacio de memoria es reservado para que las asignaciones se realicen en $\mathcal{O}(1)$. Dado que la función la recorre una única vez para inicializarla, el costo temporal resulta $\mathcal{O}(n^2)$. Por otro lado, se utilizan las funciones $size()(\mathcal{O}(1))^4$ y $push\_back()(\mathcal{O}(1))^5$ pero, dado que sus complejidades no resultan significativas frente a la complejidad final de la función, no son consideradas para el cálculo de la misma. Luego, la complejidad de **inicializarMinimizacionDeCostos** resulta $\mathcal{O}(n^2)$.
- Por otro lado, se realiza un llamado a **MinimizacióndeCostosdeTareas**. Veamos que una vez completada la matriz *valores*, la complejidad de la función recursiva se torna constante y puede verse acotada por el tamaño de la matriz (n^2).
  - En cada llamada recursiva a **MinimizacióndeCostosdeTareas**, se determina en qué máquina se ubica el $sigTrabajo$. Para tomar dicha decisión, se realizan operaciones de tiempo lineal y se ejecuta recursivamente la función dos veces. En la primera, agregando el trabajo en la máquina 1 y, en la segunda, agregando el trabajo en la máquina 2. De esta manera, se calculan las soluciones a dichos subproblemas a menos que los resultados de éstos ya se encuentren en la matriz *valores* (dado que ésta abarca todas las subsoluciones posibles), acotando dicha ejecución por $\mathcal{O}(n^2)$. A partir de esto, podemos determinar que el peor caso consiste en llenar la matriz entera $(\mathcal{O}(n^2))$ y que las soluciones a las llamadas recursivas siguientes se encuentren en ella, tornándolas constantes. Por lo tanto, el costo temporal de la llamada recursiva se encuentra acotado por la matriz, luego, por $\mathcal{O}(n^2)$.
  - Una vez almacenadas todas las subsoluciones en *valores*, se procede armando la solución final. Para ello, se reproducen las decisiones tomadas en cada paso a partir de la matriz. Dicho procedimiento se ejecuta en **inicializarMinimizacionDeCostos**, con un ciclo de 1 hasta n que compara, en cada paso, dos valores enteros de la matriz. Luego, este extracto tiene una complejidad de $\mathcal{O}(n)$ ya que las comparaciones se realizan en $\mathcal{O}(1)$.

[1] Dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima.
[2] Sabemos que existe una subsolución óptima pues estamos ante un conjunto no vacío de soluciones y todo conjunto acotado no vacío alcanza un mínimo local.
[3] http://en.cppreference.com/w/cpp/container/vector/resize
[4] http://en.cppreference.com/w/cpp/container/vector/size
[5] http://en.cppreference.com/w/cpp/container/vector/push_back

- Por lo tanto, el costo temporal de **inicializarMinimizacionDeCostos** es $\mathcal{O}(n^2)$ y el de **MinimizaciondeCostosdeTareas** es $\mathcal{O}(n^2)$ y, dado que dichas funciones se realizan en forma paralela, la complejidad temporal final es $\mathcal{O}(2n^2)$, que se encuentra acotado por $\mathcal{O}(n^2)$.

2.5. Instancias posibles

Para verificar la correctitud de nuestro programa, dispusimos variar estratégicamente las instancias de entrada al ejecutarlo.

- En primer lugar, ejecutamos el programa ingresando trabajos de igual costo independientemente del trabajo realizado anteriormente. De este modo, verificamos que nuestro algoritmo organizara los trabajos tal como esperado cuando más de una combinación posible fuera solución.

  Parámetro de entrada:

  5

  5

  5 5

  5 5 5

  5 5 5 5

  5 5 5 5 5

  Parámetro de salida:

  25 4 1 2 3 4

- Por otra parte, probamos no ingresar ningún trabajo para corroborar que la salida estuviera comprendida por un costo 0, una cantidad de trabajos asociados a una máquina igual a 0 y una lista vacía.

  Parámetro de entrada:

  0

  Parámetro de salida:

  0 0 0

- Luego, probamos ingresar costos que implicaran que todos los trabajos debían realizarse en una misma máquina con el fin de verificar que esto, efectivamente, ocurriera.

  Parámetro de entrada:

  5

  4

  7 4

  8 9 5

  6 7 5 2

  5 6 8 9 1

  Parámetro de salida:

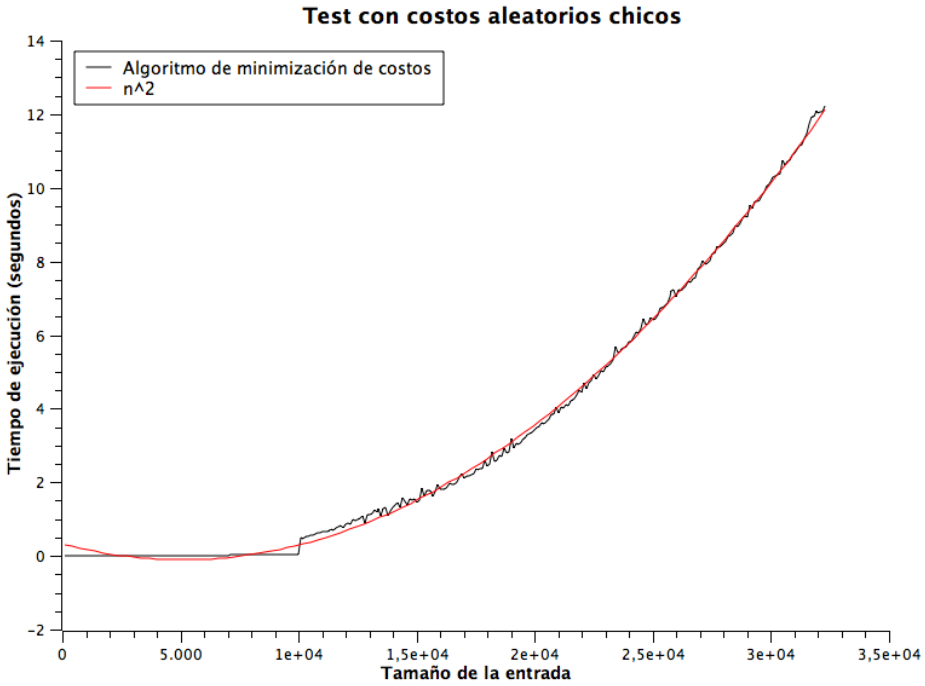
  16 5 1 2 3 4 5

De este modo, logramos abarcar los casos límite en los que la implementación pudiera haber encontrado algún problema. Dado que los resultados obtenidos fueron los esperados, determinamos que para todas las instancias válidas posibles de entrada nuestra implementación resulta correcta.

2.6. Experimentación

Para las pruebas de complejidad empírica, generamos instancias aleatorias de costos de los trabajos alterando la cantidad de los mismos. Estas instancias fueron generadas en $C++$ con la función *rand*(). La cantidad de trabajos generados se comprendió entre 100 y 32300, agregando de a 100 en cada iteración. Las mediciones de tiempo en nanosegundos se realizaron con la función *high resolution clock*[6] de la librería *Chrono* de $C++$. Debido a que éstas fueron realizadas en nanosegundos, las pruebas cuyo tamaño de entrada era menor a 1000 se realizaban en mayor tiempo que las instancias más grandes pues el procesador les otorgaba menos atención al realizar cambio de contexto. De este modo, logramos medir las pruebas de nuestro algoritmo para comprobar que la complejidad correspondiera con la mencionada anteriormente.

Las funciones de complejidad con las que se compararon nuestros gráficos de tiempo fueron ajustadas por algoritmos matemáticos (proporcionados por **sci davis**). Dichos algoritmos se encargaron de multiplicarle y sumarle constantes a las funciones con el fin de que éstas se ajustaran a nuestros resultados sin modificar el comportamiento de las funciones utilizadas para comparar.



En este caso, utilizamos instancias de números aleatorios entre 1 y 150. En este gráfico, se puede ver de forma clara que la función de la complejidad se adapta perfectamente a los datos obtenidos. De este modo, se deja en evidencia la complejidad esperada.

La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue $a_0 + a_1x + a_2x^2$. En este caso, los valores que aproximan la función a nuestros datos son:

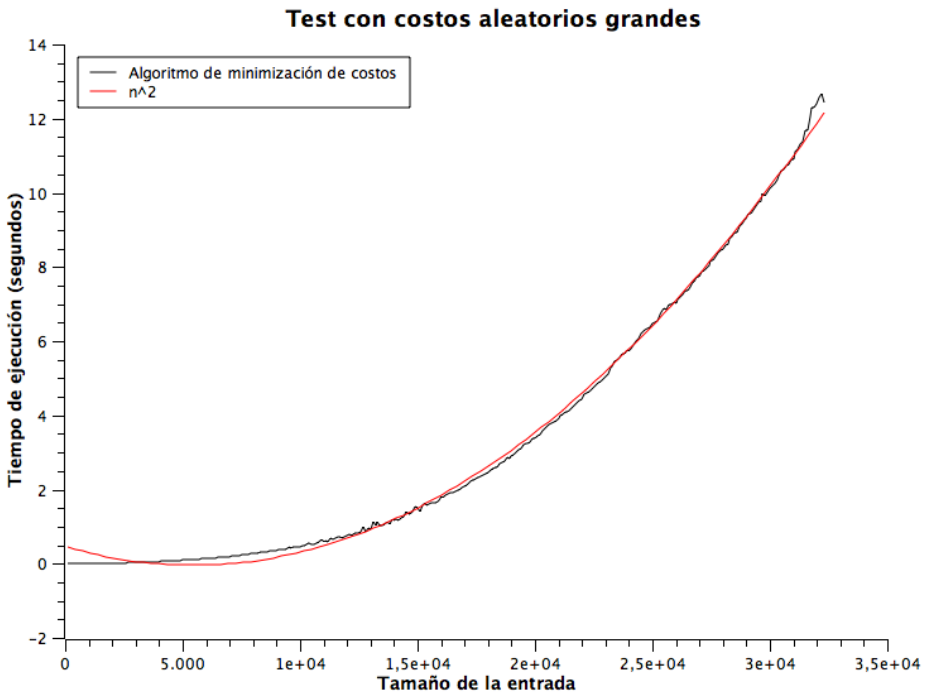
$$desde\ x = 100\ a\ x = 32{,}300$$

$$a_0 = 0,32423419158503$$

$$a_1 = -0,00016812837387441$$

$$a_2 = 1,65040451628269e^{-8}$$

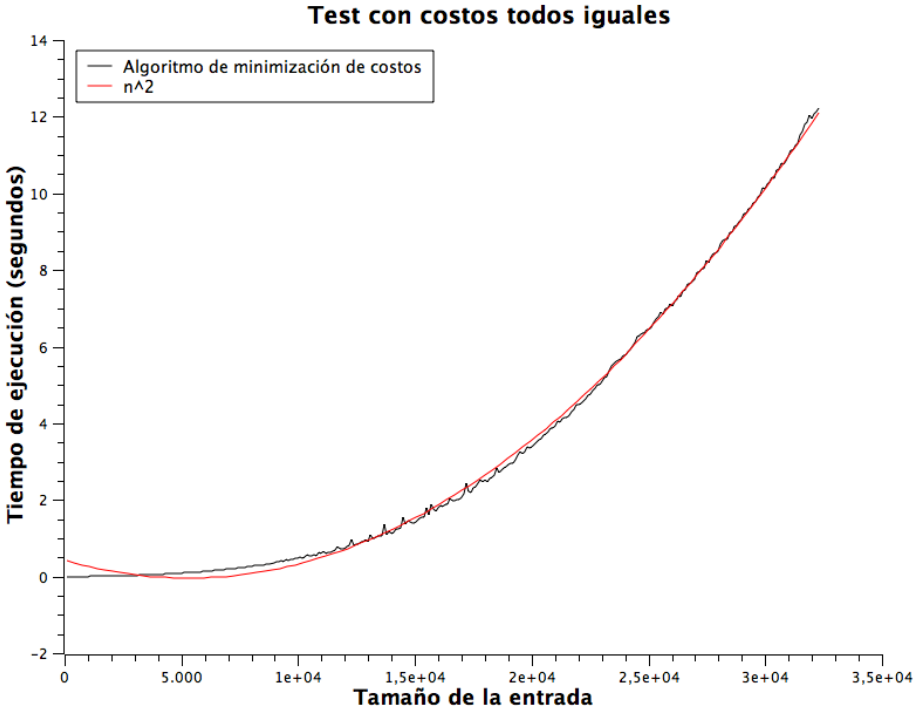
[6] http://en.cppreference.com/w/cpp/chrono/high_resolution_clock



En este caso, utilizamos instancias de números aleatorios entre 500 y 1500. Al igual que en la situación anterior, se puede observar que la función de complejidad se acerca perfectamente a los valores devueltos por las distintas ejecuciones del programa. En esta situación, encontramos que la magnitud de los costos de los trabajos no afectan la complejidad, y esto tiene sentido ya que la complejidad se encuentra determinada por la cantidad de trabajos ingresados.

La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue $a_0 + a_1x + a_2x^2$. En este caso, los valores que aproximan la función a nuestros datos son:

$$desde\ x = 100\ a\ x = 32{,}300$$
$$a_0 = 0,463420767939423$$
$$a_1 = -0,000185064576979962$$
$$a_2 = 1,69442722903951e^{-8}$$



En este caso, utilizamos instancias cuyos valores son todos iguales. Esto implica que existen múltiples soluciones al problema, y que el programa debe contemplar cada una de ellas. Al observar los resultados, llegamos a la conclusión de que la cantidad de soluciones posibles no es un factor significante a la hora de medir la complejidad de las ejecuciones. Esto se debe a que, a pesar de encontrar una solución al problema, el algoritmo debe probar con todas las demás posibilidades para comprobar que no exista una mejor.

La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue $a_0 + a_1 x + a_2 x^2$. En este caso, los valores que aproximan la función a nuestros datos son:

$$desde\ x = 100\ a\ x = 32{,}300$$

$$a_0 = 0,424466923410826$$

$$a_1 = -0,000176224064649403$$

$$a_2 = 1,66520770037108e^{-8}$$

3. Ejercicio 2

3.1. Problema a resolver

Este problema radica en una organización encargada de replicar contenido para su red de entrega de información en Internet. Para esto, dispone de n servidores interconectados mediante m enlaces *backbone* de alta velocidad. El servidor *máster* es el primero en recibir los nuevos datos a replicar y es el encargado de transmitir dicha información a otros servidores. Éstos guardarán su propia copia y la retransmitirán hasta que todos los servidores contengan la nueva información. Es importante considerar que todos los enlaces transmiten la misma información, que los n servidores se encuentran interconectados por dichas conexiones y que los enlaces *backbone* tienen costo en función del tráfico que transmiten.

La organización seleccionó dos consultoras con el fin de resolver los siguientes problemas:

- El primero se basa en hallar un algoritmo capaz de encontrar el camino mínimo entre un conjunto de servidores. En el contexto del problema, esto consiste en elegir los enlaces que permitan distribuir la información a todos los servidores con un costo mínimo.
- El segundo consiste en encontrar el servidor *máster* de forma tal que la replicación de la información, que empieza una vez que el *máster* recibe la información y termina cuando todos los servidores tienen su copia, se realice en el mínimo tiempo posible. Para ello, se considera que la información tarda el mismo tiempo en atravesar cualquier enlace y que cada servidor transmite simultáneamente a todos los vecinos elegidos.

Formatos de entrada y salida:

La entrada contiene varias instancias del problema. La primera línea de cada instancia contiene un entero positivo n que indica la cantidad de servidores (numerados de 1 a n), un espacio, y un entero no negativo m que corresponde a la cantidad de enlaces. A esta línea le siguen m líneas con la descripción de cada enlace (números separados por un espacio):

$$a_j \ b_j \ ... \ c_j$$

donde a_j y b_j son los servidores que el enlace j conecta y c_j es el costo de usar el enlace j (entero no negativo). La entrada concluye con una línea comenzada por 0.

La salida debe contener una línea por cada instancia de entrada, con el siguiente formato:

$$C \ U \ k \ e_1^1 \ e_2^1 \ e_1^2 \ e_2^2 ... \ \ e_1^k \ e_2^k$$

donde C es el costo total de la solución, U es el servidor elegido como *máster*, k es la cantidad de enlaces y e_1^i y e_2^i son los extremos del i-ésimo enlace de la solución obtenida, para $i \in [1,k]$.

En lo que sigue, presentaremos dos ejemplos del problema a resolver:

- **Ejemplo 1:**

  El siguiente ejemplo consiste en un caso simple de la situación a resolver con la particularidad de que existe más de una solución posible al problema. Como puede observarse, tanto el enlace (1,5) como el (5,4) forman parte de la solución óptima, pero de éstos se elige únicamente uno.

  Formato de entrada:

5 6

1 2 2
1 3 1
1 5 2
2 3 1
3 4 1
4 5 2

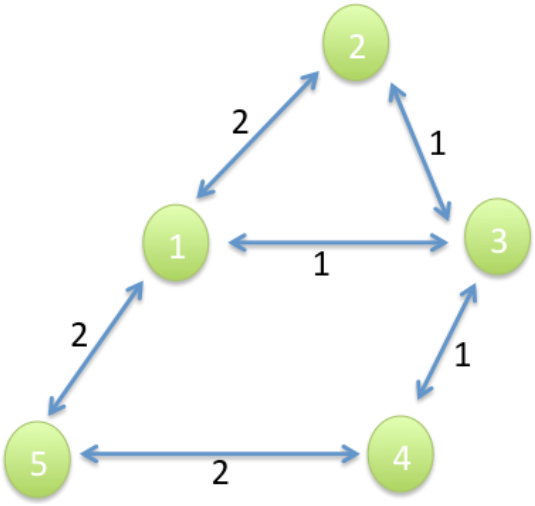


Figura 5: Ejemplo 1 - entrada.

Formato de salida:

5 2 4 1 2 1 5 2 3 3 4

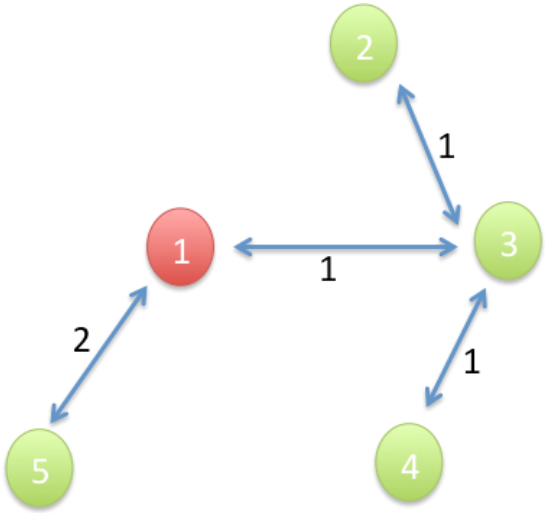


Figura 6: Ejemplo 1 - salida.

- **Ejemplo 2:**

  Este ejemplo consiste en un caso usual de la situación planteada en el que puede verse de forma simple y clara la elección que debe realizar el algoritmo para hallar la solución óptima.

  Formato de entrada:

  5 6
  1 2 2
  1 3 3
  2 3 1
  3 4 2
  3 5 2
  4 5 1

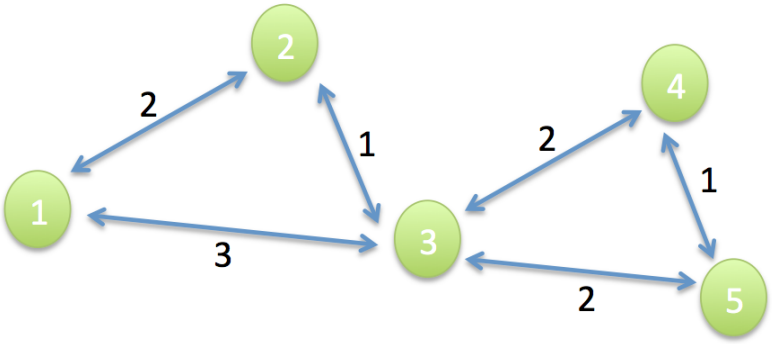


Figura 7: Ejemplo 2 - entrada.

Formato de salida:

8 3 4 1 3 2 3 3 4 3 5

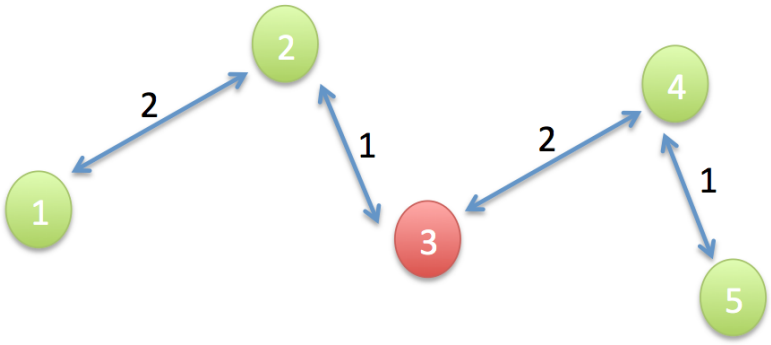


Figura 8: Ejemplo 2 - salida.

3.2. Minimización de enlaces

3.2.a. Resolución coloquial

El problema planteado consiste en conectar todos los servidores de la red de forma tal que el costo de transmisión de datos sea el mínimo. Para poder resolver esto, decidimos utilizar un algoritmo que encontrara un Árbol Generador Mínimo[7] que contuviera los enlaces de menor peso necesarios para unir a todos los servidores. Para ello, utilizamos el algoritmo de *Kruskal*. Éste genera un conjunto de árboles que representan los nodos del grafo original. El algoritmo toma la menor arista (según su peso) y evalúa si sus extremos pertenecen a árboles distintos. En caso de así serlo, los une y agrega la arista al árbol generador mínimo. Si, por el contrario, los extremos forman parte del mismo árbol, el algoritmo descarta dicha arista.

El pseudocódigo del algoritmo mencionado es el siguiente:

Algorithm 3: kruskal

```
   Input: Entero CantidadServidores, Grafo G
   Output: Grafo Gr
 1 //G = (V, E)
 2 ordenarDeMenorAMayorPorPeso E
 3 Grafo Gr ← <>
 4 forall the E do
 5 |   Entero nodoA ← PrimerVertice E
 6 |   Entero nodoB ← SegundoVertice E
 7 |   if find(nodoA) != find(nodoB) then
 8 |   |   union(find(nodoA), find(nodoB))
 9 |   |   Marcar(E)
10 |   end
11 end
12 forall the E do
13 |   if EstaMarcado?(E) then
14 |   |   Insertar(Gr,E)
15 |   end
16 end
17 devolver Gr
```

Para lograr que el algoritmo de Kruskal fuera más eficiente, decidimos aplicarle una mejora tanto a la función union como a la función find.

En el caso de la función *find*, decidimos aplicar el método de *compresión de caminos* que, al momento de retornar la raíz en la recursión, actualiza el padre de cada vértice visitado. Luego, cada nodo ya visitado tiene como padre a la raíz.

Para la función *union*, implementamos el método *unión por rango*. Éste une el árbol de menor rango con la raíz del de mayor rango. Luego, la altura resultante es igual a la mayor de los dos. Si no se aplicara dicha función, la altura resultante sería la del árbol más grande +1 dado que éste se colocaría como hijo de la raíz del de menor tamaño.

Los pseudocódigos de dichos algoritmos son los que se presentan a continuación:

[7] El peso de un árbol generador es la suma de los pesos de las aristas del árbol. Un árbol generador mínimo es uno con peso mínimo.

Algorithm 4: find

```
Input: Entero nodo, vector<Entero> padre
Output: Entero res
if nodo = Raiz(nodo) then
    devolver nodo
end
devolver Raiz(nodo) ← find(Raiz(nodo))
```

Algorithm 5: union

```
Input: Entero nodoA, Entero nodoB, vector<Entero> padre
Output: Entero res
if AlturaDeLaComponente nodoA > AlturaDeLaComponente nodoB then
    Raiz(nodoB) = nodoA
else
    Raiz(nodoA) = nodoB
    AlturaDeLaComponente nodoB++
end
```

Donde la función $Raiz(x)$ devuelve la raíz del árbol conteniendo el nodo x y la función $AlturaDeLaComponente(x$ devuelve la altura del árbol que comprende a x.

3.2.b. Demostración de correctitud

Para demostrar la correctitud de nuestro algoritmo, debemos probar que se cumplen las siguientes propiedades:

- Se unen todos los servidores mediante enlaces *backbone*.
- La solución es óptima, es decir, no existe otra combinación de enlaces que una a todos los servidores y cuyo costo sea menor.

Se unen todos los servidores mediante enlaces backbone:

La entrada de nuestro algoritmo es un grafo conexo. Esto significa que, para cualquier par de vértices (v_1, v_2) del grafo, existe al menos un camino que los une.

Sea G el grafo de entrada de n nodos y m aristas, y G' uno conformado por n componentes conexas, donde cada una de ellas es un vértice de G. En primer lugar, nuestro algoritmo comprueba si la inserción de cada arista de G en G' genera algún ciclo. En caso de producirlo, no la agrega, caso contrario sí. En otras palabras, se insertan en G' las aristas que únicamente unen dos componentes conexas distintas. Al repetir dicho procedimiento para las m aristas de G, obtenemos un grafo sin ciclos que une a los n nodos. Esto es posible dado que G es conexo.

La solución es óptima:

Sea G un grafo conexo, E un arreglo de aristas de G ordenadas por peso y G' un bosque con n árboles, que representan a los nodos de G. Probemos que si G es conexo y G'_k tiene i árboles mínimos $\Rightarrow$ G'_{k+1} tiene i o $i-1$ árboles mínimos.

Sea e_k la k-ésima arista perteneciente a E y sean v_1 y v_2 sus extremos. Si v_1 y v_2 pertenecen al mismo árbol, no agregamos e_k para no generar un ciclo y quitarle la propiedad de árbol. Entonces, G'_{k+1} queda conformado por i árboles mínimos.

Por otro lado, si v_1 y v_2 no pertenecen al mismo árbol, e_k es la arista de menor peso que los une. Esto se debe a que las $k-1$ aristas anteriores ya conforman G'_k o no, de acuerdo a si la inserción de las mismas generan ciclos. Al agregar e_k en G'_k, se forma un nuevo árbol de peso mínimo por estar conformado por dos árboles mínimos y la arista de menor peso entre todas las que los conectan. Por lo tanto, G'_{k+1} queda conformado por $i-1$ árboles mínimos.

Al ejecutar este proceso m veces, en G' se obtiene una única componente conexa con $n-1$ aristas (no pueden ser más dado que sino G' no sería un árbol).

3.2.c. Complejidad del algoritmo

Sea n la cantidad de servidores que posee una red de entrega de contenidos (nodos) y m la cantidad de enlaces que los unen (aristas). Supongamos que tenemos un grafo completo, entonces $m = \frac{n(n-1)}{2}$. Luego, $m \leq \frac{n(n-1)}{2} \leq n^2$.

Para analizar la complejidad de nuestro algoritmo, veámoslo por partes:

- Algoritmo de $Kruskal$
  La función $ordenarDeMenorAMayorPorPeso$ se encuentra implementada con la función $sort$[8]. La misma se aplica sobre un vector que contiene todas las aristas ingresadas por parámetro. Como dicho vector tiene tamaño m, la complejidad de aplicar esta función resulta $\mathcal{O}(m\ log\ m)$. Al acotar m por n^2, obtenemos, como complejidad final, $\mathcal{O}(n^2\ log\ n^2)$.

  Luego, se ingresa en el ciclo principal que recorre todas las aristas del grafo en $\mathcal{O}(m) = \mathcal{O}(n^2)$. En él, se calculan las funciones $PrimerVertice$ y $SegundoVertice$ con una complejidad de $\mathcal{O}(1)$ ya que dicha información se encuentra alojada en la estructura. Posteriormente, se calcula $find$ a cada uno de los nodos cuya complejidad es $\mathrm{O}(ACK(n))$ amortizado, donde ACK corresponde a la función de Ackermann acotada por $\mathcal{O}(1)$ amortizado[9].

  Luego, se controla si los nodos pertenecen o no a la misma componente conexa. En caso de que no pertenezcan, se aplican las funciones $union$ y $Marcar$. La función $union$, al igual que $find$, está acotada por la función de Ackermann, o sea que por $\mathcal{O}(1)$ amortizado. Por otro lado, $Marcar$ indica que la arista que se le pasa por párametro debe ser insertada posteriormente en un grafo. Esta acción se realiza en $\mathcal{O}(1)$ ya que se utiliza un arreglo al cual se accede en forma directa.

  Por lo tanto, podemos concluir que las ejecuciones dentro del ciclo tienen una complejidad temporal de $\mathcal{O}(1)$. Luego, la complejidad del ciclo es de $\mathcal{O}(n^2)$.

  Por último, se recorren todas las aristas para ver si fueron marcadas o no. Si lo fueron, se las inserta en un grafo. Este grafo esta implementado sobre un vector al cual se le redefine su tamaño mediante la funcion $resize$[10] con una complejidad de $\mathcal{O}(n-1)$. Dicha inserción se realiza mediante la función $Insertar$ la cual agrega la arista en $\mathcal{O}(1)$. Por lo tanto, el costo de este ciclo es $\mathcal{O}(n-1)$

  Finalmente, al sumar las complejidades de $ordenarDeMenorAMayorPorPeso$ con la del ciclo principal, obtenemos que la complejidad temporal del Algoritmo de $Kruskal$ está dada por $\mathcal{O}(n^2\ log\ n^2) + \mathcal{O}(n^2)$, siendo estrictamente menor que $\mathcal{O}(n^3)$.
- Cálculo del costo de la red
  Para poder calcular el costo temporal de la red, debe considerarse que se recorren todas las aristas devueltas por el algoritmo de $Kruskal$ y se suman sus respectivos pesos. Dado que dicho algoritmo devuelve un árbol, la cantidad de aristas equivale a $n-1$, por lo tanto la complejidad del Cálculo del costo de la red es $\mathcal{O}(n)$.

- Generación de la lista de adyacencia
  Para poder generar la lista de adyacencia, creamos un vector de vectores. El vector principal es de n elementos y en cada posición se guarda un vector conteniendo los nodos adyacentes. Para poder rellenarlo, se recorre el vector de aristas resultante de aplicar $Kruskal$ ($\mathcal{O}(n)$) y

[8] http://en.cppreference.com/w/cpp/algorithm/sort
[9] TARJAN, ROBERT ENDRE (1975). .Efficiency of a Good But Not Linear Set Union Algorithm". Journal of the ACM 22
[10] http://www.cplusplus.com/reference/vector/vector/resize/

se obtienen los extremos de cada arista en $\mathcal{O}(1)$. Luego, se inserta el primer nodo en la lista del segundo y el segundo en la lista del primero para cada par de aristas. Dicha inserción se realiza con *push_back*() ($\mathcal{O}(1)$)[11]. Por lo tanto, generar la lista de adyacencia tiene una complejidad de $\mathcal{O}(m)$*$\mathcal{O}(1)$ y, al acotar m por n^2, ésta resulta $\mathcal{O}(n^2)$.

Finalmente, la complejidad final del algoritmo resulta de sumar las fracciones mencionadas anteriormente. Luego, ésta es $\mathcal{O}(n^2\ log\ n^2) + \mathcal{O}(n) + \mathcal{O}(n^2)$ que es estrictamente menor a $\mathcal{O}(n^3)$.

3.3. Selección de servidor *máster*

3.3.a. Resolución coloquial

El problema a resolver consiste en encontrar el nodo del grafo generado en el ítem *a*) desde el que la replicación de contenido se realice en la menor cantidad de pasos posibles. Para esto, decidimos utilizar un algoritmo que nos devolviera el camino más largo del árbol a procesar. A partir de éste, se devuelve el nodo del centro que es el que menos pasos debe realizar para llegar a los más alejados del árbol.

El algoritmo que utilizamos para encontrar el camino de longitud máxima es BFS (*Búsqueda en anchura*). Éste toma un nodo inicial y devuelve el más distante a él. Para ello, procede recorriendo desde el nodo inicial y explorando todos sus nodos adyacentes de manera sucesiva hasta recorrer todo el árbol. La particularidad del algoritmo es que recorre el árbol por niveles, luego, el último nodo que visita pertenece al último nivel del árbol, resultando éste el más alejado del inicial.

Como BFS retorna el camino más largo desde un nodo inicial, decidimos aplicarlo dos veces sobre nuestro grafo. La primera vez lo hicimos sobre el *nodo*1, cuya existencia está asegurada para cualquier grafo no nulo. De este modo, obtenemos el nodo más distante a éste, que es a su vez utilizado para ejecutar el algoritmo por segunda vez. De esta forma, obtenemos el camino de longitud máxima conformado por el nodo resultante de la primera pasada de BFS y el de la segunda.

El pseudocódigo del algoritmo BFS es el siguiente:

Algorithm 6: BFS

```
Input: Entero nodoInicial, Entero CantidadNodos, Grafo G
Output: Par<Entero, Entero> res
encolar(Q, nodoInicial)
marco nodoInicial
Distancia(nodoInicial) ← 0
while !vacia(Q) do
    nodo ← extraer(Q)
    forall the v ∈ Adyacentes(nodo) do
        if v no esta marcado then
            marco v
            Distancia(v) ← Distancia(nodo) + 1
            encolar(Q, v)
        end
    end
end
primero(res) ← nodo
segundo(res) ← nodoDelMedio
devolver res
```

Donde *marco* evalúa si ese nodo ya fue visitado, *Distancia* indica la distancia de ese nodo al *nodoInicial*, *Adyacentes* devuelve una lista con todos sus nodos adyacentes y *nodoDelMedio*

[11] http://www.cplusplus.com/reference/vector/vector/push_back/

retorna el nodo que se encuentra en la mitad del camino generado entre el nodo inicial y el que devuelve el BFS.

3.3.b. Demostración de correctitud

Demostremos que nuestro algoritmo encuentra, efectivamente, el nodo central del camino más largo de un árbol. Para lograr esto, se realiza BFS una vez para obtener uno de los extremos de este camino. Posteriormente, se repite la ejecución con ese extremo para obtener el camino más largo.
Para demostrar esto, necesitamos probar dos cosas:

- La primera ejecución de BFS nos devuelve uno de los extremos del camino más largo del árbol.
- La segunda ejecución nos devuelve el camino más largo del árbol.

La primera ejecución de BFS nos devuelve un extremo del camino más largo del árbol:

Sea G un árbol y v_1, v_2 dos de sus vértices. Sabemos que existe un camino entre v_1 y v_2 que los une.

Por otro lado, el algoritmo BFS[12] toma el nodo inicial de un árbol y devuelve el más distante a él, junto con el camino máximo entre estos.

Demostremos por el absurdo que, dado cualquier nodo del árbol, BFS retorna uno de los extremos del camino máximo. Para ello, miremos todos los casos en los que esto no ocurre y probemos que ninguno de éstos es posible.

- El nodo final no es una hoja de G.
  Sea v_1 el nodo inicial y v_2 el nodo final del algoritmo BFS. Si v_2 no es una hoja, existe un nodo v' que sí lo es (pues es un árbol y no posee ciclos) y un camino c que une a los nodos v' y v_2. Entonces, v_2 no es el nodo más distante a v_1 ya que el camino que une v_1 con v' es más largo que el que une v_1 con v_2. Pero esto contradice la hipótesis de la que partimos. Luego, es absurdo.

- El nodo inicial pertenece al camino máximo del árbol y el final no.

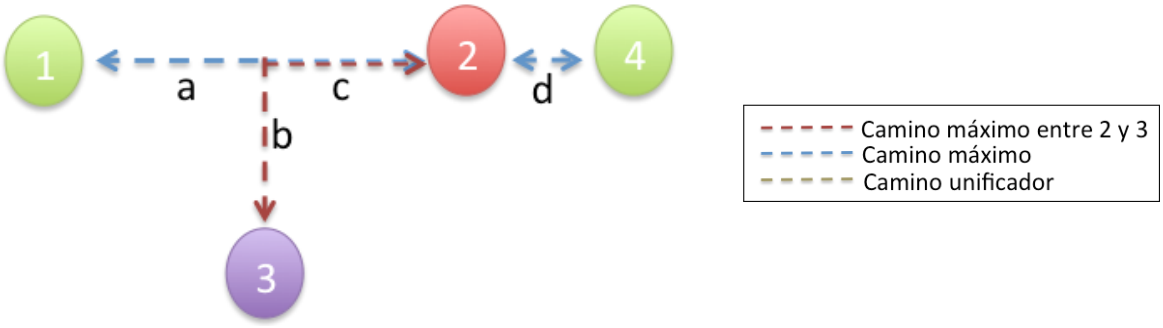


Figura 9: Segundo caso posible.

A partir del gráfico anterior, supongamos que el camino rojo representa al máximo que resulta del algoritmo BFS luego de haberlo aplicado sobre el nodo 2.
Por otro lado, supongamos que dicho algoritmo retorna al nodo 3 como el más lejano del 2. Como el nodo 3 se encuentra más lejos del 2 que el 1, $b \geq a$.

[12] Cormen, Thomás H., Charles E. Leiserson, and Ronald L. Rivest. Chapter 22.2

- $a = b$:
  En este caso, ambos caminos hubieran sido correctos ya que dist(4,1) sería igual a dist(4,3).
- $a < b$:
  Si este caso sucediera, b debería formar parte del camino máximo del árbol. Esto se debe a que el camino más largo del árbol debe estar conformado por los caminos c-d y el máximo entre a y b. Por lo tanto, si $b > a$, el camino azul no sería el máximo. Luego, es absurdo suponer que el extremo del camino máximo retornado por BFS no pertencía al camino máximo del árbol.

- Tanto el nodo inicial como el final no pertenecen al camino máximo del árbol.

  Sea el camino *azul* el máximo del árbol cuyos extremos son los nodos 1 y 4. Supongamos que el camino *rojo* representa al camino devuelto por BFS al aplicarlo sobre el nodo 2. En este caso, tenemos dos opciones posibles: El camino *azul* comparte un tramo con el camino *rojo* o no comparte ninguno. Veamos cada una de ellas por separado.

  - El camino azul comparte un tramo con el camino rojo.

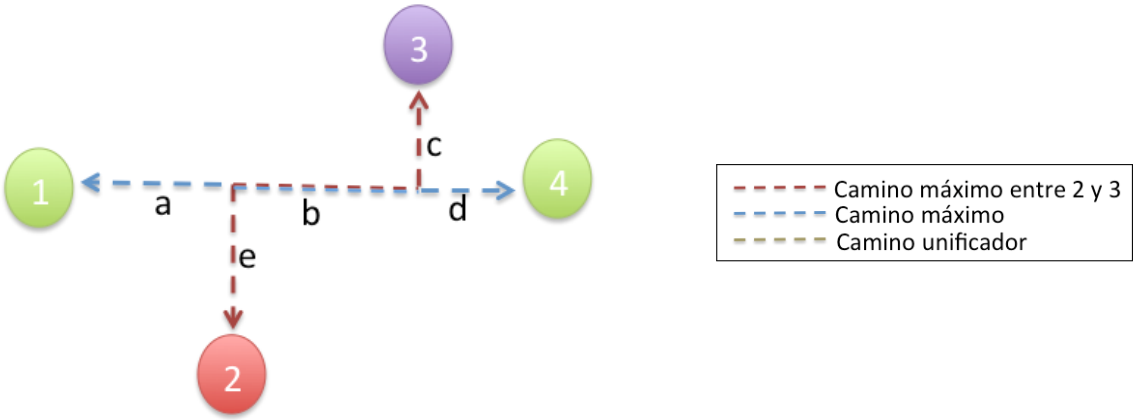


Figura 10: Tercer caso posible.

Supongamos que el algoritmo BFS devuelve el nodo 3 como el más alejado al 2. Entonces, podemos deducir que:

- $c = d$:
  Ambas soluciones son correctas.
- $c > d$:
  Si este caso ocurriera, c debería formar parte del camino máximo del árbol. Luego, el camino azul no sería el máximo. Por lo tanto, resulta absurdo suponer que el extremo del camino máximo otorgado por BFS no pertencía al camino máximo del árbol.

  - El camino azul no comparte nigún tramo con el rojo.

Figura 11: Cuarto caso posible.

Supongamos que el algoritmo BFS nos devuelve el nodo 3 como el más alejado del 2. Entonces, podemos deducir que:

- a U $c = e$:
  Ambas soluciones son correctas.
- b U $c = e$:
  Ambas soluciones son correctas.
- a U $c < e$:
  Si este caso sucediera, e U c debería formar parte del camino máximo del árbol. Pero entonces el camino azul no sería el máximo, lo que resulta absurdo.
- b U $c < e$:
  En este caso, e U c debería formar parte del camino máximo del árbol. Luego, resulta absurdo.

Luego, logramos demostrar que la primera ejecución de BFS retorna como resultado una hoja del árbol al que se le aplica. Además, probamos que dicha hoja pertenece obligatoriamente al camino máximo. Por lo tanto, podemos concluir que la primera ejecución devuelve uno de los dos extremos del camino más largo del árbol.

La segunda ejecución nos devuelve el camino más largo del árbol.

A la segunda ejecución del algoritmo le ingresa por párametro uno de los extremos del camino más largo del árbol. Luego de dicha ejecución, se obtiene entonces el otro extremo. Una vez encontrado el camino más largo, éste debe ser recorrido hasta hallar el nodo intermedio.

3.3.c. Complejidad del algoritmo

Sea n la cantidad de servidores que posee una red de entrega de contenidos (nodos) y m la cantidad de enlaces que los unen (aristas). Dado que procesamos un árbol, sabemos que $m = n - 1$, en particular, $m \leq n$.

Para analizar la complejidad de nuestro algoritmo, vamos a calcular la complejidad del algoritmo BFS.

Lo primero que hace BFS es crear dos arreglos de tamaño n, uno para almacenar las distancias y el otro para ir marcando los nodos por los cuales ya ocurrió. Crear ambos arreglos tiene una complejidad de $\mathcal{O}(n)$. Además se encola, mediante la función *push*[13], el primer elemento. La complejidad de esta función es $\mathcal{O}(1)$ ya que la estructura interna de la cola está implementada sobre $double - ended\ queue$ y su función *push_back* tiene complejidad $\mathcal{O}(1)$.

Posteriormente, se ingresa a un ciclo que se ejecuta mientras la cola no se encuentre vacía. Esto se controla utilizando la función *empty*[14] cuya complejidad es de $\mathcal{O}(1)$.

Una vez dentro del ciclo, se aplica la función $front$[15] sobre la cola para poder obtener el valor del primer nodo. Luego se usa la función *pop*[16] para quitar este valor de la cola. Ambas funciones tienen una complejidad constante.

[13] http://www.cplusplus.com/reference/queue/queue/push/
[14] http://www.cplusplus.com/reference/queue/queue/empty/
[15] http://www.cplusplus.com/reference/queue/queue/front/
[16] http://www.cplusplus.com/reference/queue/queue/pop/

Por ultimo, se ejecuta un ciclo el cual recorre un vector de nodos adyacentes. Para cada uno de estos nodos, el algoritmo se fija si fue marcado o no. En caso afirmativo, aplica inserciones sobre arreglos con un *push*; Por lo tanto, dentro del ciclo la complejidad es constante.

Como podemos observar, el ciclo principal se ejecuta n veces ya que todos los nodos son colocados una sola vez en la cola. Además, dentro de este se encuentra otro bucle el cual recorre todos los nodos adyacentes a cada vértice. Es por esto que en el ciclo principal se terminan recorriendo m aristas en las n iteraciones . Por lo tanto la complejidad del ciclo principal es de $\mathcal{O}(n) + \mathcal{O}(m)$ que es igual a $\mathcal{O}(n)$.

En resumen, el ciclo principal tiene complejidad $\mathcal{O}(n)$ y las funciones que se encuentran antes que este, tienen complejidad $\mathcal{O}(n)$ por lo tanto, la complejidad del algoritmo es $\mathcal{O}(n)$

3.3.d. Instancias posibles

Para verificar la correctitud de nuestro programa, dispusimos variar estratégicamente las instancias de entrada al ejecutarlo.

- En primer lugar, ejecutamos el programa ingresando un caso con una única arista. De este modo, logramos corroborar el correcto funcionamiento de nuestro algoritmo para este tipo de casos borde.
  Parámetro de entrada:

  2 1

  1 2 13

  Parámetro de salida:

  13 2 1 1 2

- Posteriormente, ejecutamos el programa ingresando un solo nodo. De esta forma, pudimos controlar que ocurría si no se ingresaban aristas.
  Parámetro de entrada:

  1 0

  Parámetro de salida:

  0 1 0

- Por otra parte, quisimos ver un caso que minimizara los costos en el primer ítem pero que ralentizara, luego, el trabajo del *máster*. Este caso nos permite ver que no siempre es mejor elegir el camino mínimo antes que el *máster*, sino que depende lo que se priorice (en este caso el costo final).
  Parámetro de entrada:

  5 5

  1 3 10

  3 2 10

  3 4 10

  3 5 10

  4 5 5

  Parámetro de salida:

  35 3 4 4 5 1 3 3 2 3 4

- También, buscamos una situación inversa al caso anterior, es decir, en el que la elección de conexiones con menor costo permitiera, luego, encontrar un *máster* eficiente.
  Parámetro de entrada:

  5 6

  1 2 12

1 3 14

1 4 10

2 4 9

3 4 10

4 5 11

Parámetro de salida:

40 4 4 2 4 1 4 3 4 4 5

- Para finalizar, buscamos un caso que tuviera varias soluciones óptimas en el primer ítem pero que, cada una de estas, implique un costo de replicación de datos distintos (segundo ítem).
  Parámetro de entrada:

5 8

1 2 1

2 3 1

3 4 1

4 1 1

5 1 1

5 2 1

5 3 1

5 4 1

Parámetro de salida:

4 2 4 1 2 2 3 3 4 5 1

De este modo, logramos abarcar los casos límite en los que la implementación pudiera haber encontrado algún problema. Dado que los resultados obtenidos fueron los esperados, determinamos que para todas las instancias válidas posibles de entrada nuestra implementación resulta correcta.

3.3.e. Experimentación

Para las pruebas de complejidad empírica, generamos instancias aleatorias de costos de los trabajos alterando la cantidad de los mismás. Estas instancias fueron generadas en $C++$ con la función *rand*(). La cantidad de trabajos generados se comprendió entre 1000 y 100000, agregando de a 1000 en cada iteración. Las mediciones de tiempo en nanosegundos se realizaron con la función *high resolution clock*[17] de la librería *Chrono* de $C++$. Debido a que éstas fueron realizadas en nanosegundos, las pruebas cuyo tamaño de la entrada era menor a 1000 se realizaba en mayor tiempo que las instancias más grandes pues el procesador le otorga más atención al no realizar cambio de contexto. De este modo, logramos medir las pruebas de nuestro algoritmo para comprobar que la complejidad correspondiera con la mencionada anteriormente.

Las funciones de complejidad con las que se compararon nuestros gráficos de tiempo fueron ajustadas por algoritmos matemáticos (proporcionados por **sci davis**). Dichos algoritmos se encargaron de multiplicarle y sumarle constantes a las funciones con el fin de que éstas se ajustaran a nuestros resultados sin modificar el comportamiento de las funciones utilizadas para comparar.

[17] http://en.cppreference.com/w/cpp/chrono/high_resolution_clock

3.4. Preguntas adicionales

3.4.a. Pregunta 1

En primer lugar, mostremos un contraejemplo en el que se pueda ver que es posible resolver las dos partes por separado de manera óptima pero que aún así haya una solución en la que la replicación termine en menos tiempo.

Supongamos que tenemos el siguiente grafo:

Formato de entrada:

5 5
4 3 1
5 3 1
3 2 1
1 5 1
3 1 1

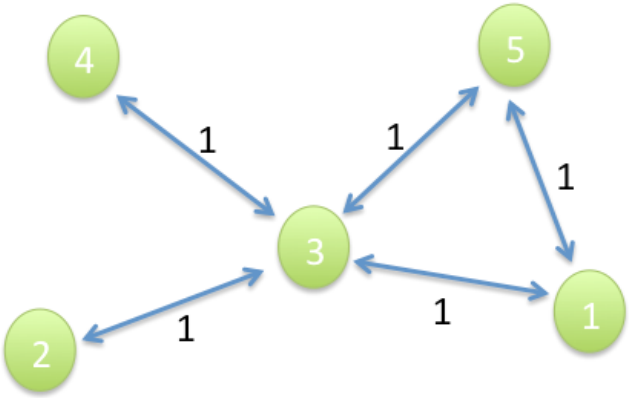


Figura 12: Pregunta 1 - entrada.

Ahora ejecutemos nuestro programa y veamos qué es lo que nos devuelve.

Formato de salida:

4 5 4 4 3 5 3 3 2 1 5

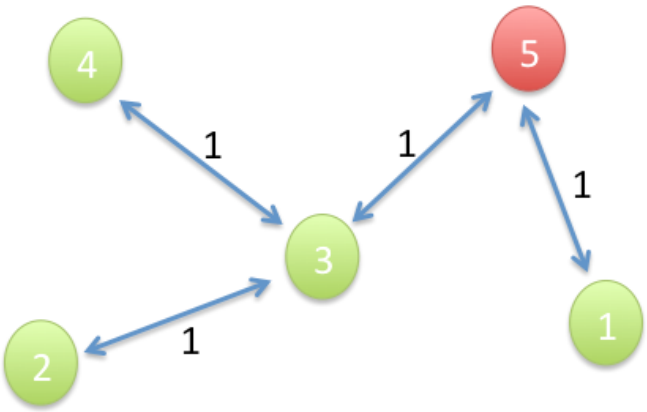


Figura 13: Pregunta 1 - salida.

Como podemos observar en el gráfico anterior, el nodo *máster* es el 5 y el costo de replicación es igual a 2. Otra solución posible hubiera sido la que tiene al 3 como nodo *máster* ya que ésta también tiene costo de replicación 2.

Analicemos por qué nuestro algoritmo devolvió dicha solución. En primer lugar, el árbol generador mínimo (AGM) retornado, se encuentra conformado por las primeras cuatro aristas que ingresamos como parámetro de entrada. Ésto se debe a que nuestro algoritmo ordena primero las aristas por su peso y luego toma las menores para generar el AGM. En este caso, como todas las aristas tienen el mismo peso, el algoritmo no cambia su orden y genera el AGM con las primeras cuatro.

Ahora veamos que ocurre si cambiamos el orden de las aristas:

Formato de entrada:

5 5

4 3 1

5 3 1

3 2 1

3 1 1

1 5 1

Para ellos, cambiemos de lugar las cuarta y quinta aristas:

Formato de salida:

4 3 4 4 3 5 3 3 2 3 1

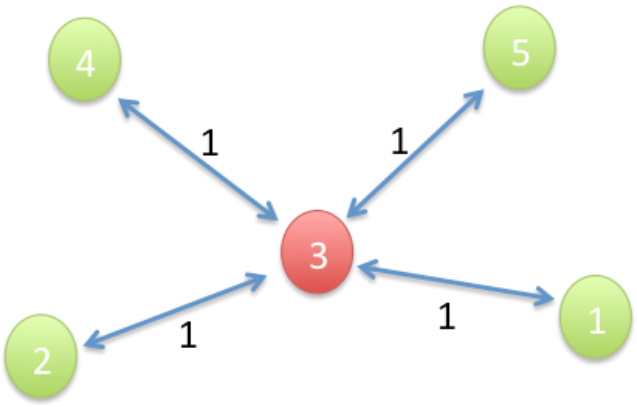


Figura 14: Pregunta 1 - salida2.

Al realizar dicho cambio, el *AGM* resultante varía. Esto se debe a que, nuevamente, nuestro algoritmo lo genera con las primeras 4 aristas. Sin embargo, esta distribución sigue siendo solución de nuestro problema ya que el costo de sus aristas sigue siendo 4.
Ahora, observemos qué sucede con nuestro nodo *máster*. El nodo máster que nos da esta solución es 3, que también era una posible solución del *AGM* anterior. La única diferencia con la solución anterior es que, ahora, nuestro costo de replicación es igual a 1, lo que significa que éste es menor.

Este es un claro ejemplo de un grafo que se resolvió dos veces de manera óptima pero cuya replicación de uno fue menor a la del otro. Esta diferencia se debió únicamente al orden en el que se ingresaron los parámetros.

más generalmente, esto puede ocurrir para cualquier grafo en el cual la solución del *ítem a* (aplicar el algoritmo de Kruskal) no sea única. Esto se debe a que todas las soluciones van a ser optimás para el *ítem a* y para cada una de estas, se va a encontrar la solución optima para el *ítem b*. Al hacer esto, vamos a encontrar un subconjunto de las soluciones posibles del *ítem a* que van a tener un menor costo de replicación que las demás.

Dos posibles soluciones a esto serian:

- Para todas las soluciones posibles del *ítem a*, calcular el *ítem b* y me quedo con la que me de menor costo de replicación.
- Modificar el *ítem a* para que ordene primero por peso de la arista y luego por la cantidad (de mayor a menor) de nodos adyacentes que tienen ambos extremos de esta.

3.4.b. Pregunta 2

Por otro lado, veamos de qué modo puede modificarse la solución si en lugar de transmitir por *broadcast* se lo hace por *multicast*, enviando un paquete a cada destino sin hacer copias.

La principal diferencia entre uno y otro radica en que el primero (*broadcast*) con mandar una vez la informacion le alcanza, ya que cada servidor se guarda y enviá una copia, mientras que el segundo lo que hace es enviar una copia a cada nodo, por lo que realizar un envío de un nodo a a un nodo b cuesta la sumatoria del costo de cada arista del camino.

Dicho lo anterior, una posible solución priorizando el costo de envió de datos es:

Algorithm 7: Minimización de Caminos entre Fábricas y Clientes

Input: Grafo G
Output: Grafo

```
1 Grafo Gmin
2 Entero pesoMin = 0
3 Mientras(n ∈ Nodos(G))
4   Grafo G' = Dijktra(n,G)
5   Si (pesoMin ≤ pesoTotal(G'))
6     Gmin = G'
7   FinSI
8 FinMientras
9 return Gmin
```

Donde *pesoTotal* calcula la sumatoria del peso de todas las aristas y $Dijktra$[18] calcula los caminos mínimos de un nodo hasta todos los demás. De esta manera nos vamos quedando con aquella solución que la sumatoria de todas las aristas posea menor peso.

Esta solución tiene una complejidad de $\mathcal{O}(n^3)$ siendo n la cantidad de servidores (nodos), ya que por cada vértice aplica dicho algoritmo.

[18] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. p.658

4. Ejercicio 3

4.1. Problema a resolver

El siguiente problema consiste en realizar un algoritmo capaz de resolver problemas de caminos mínimos con ciertas particularidades. El mismo radica en una empresa productora de ladrillos cuyas fábricas se encuentran en distintos lugares de una provincia, al igual que sus clientes. Para transportar los ladrillos entre las fábricas y los clientes, la empresa debe encargarse de fortalecer las rutas por las que se trasladan sus camiones. Dichas rutas tienen un costo de inversión proporcional a la longitud de las mismas. Luego, la empresa desea replanificar sus recorridos a modo de minimizar los gastos que implican los fortalecimientos de las rutas a utilizar. Para ello, la empresa debe asegurarse de que exista un camino fortalecido entre cada cliente y al menos una de sus fábricas. Debe considerarse que desde todas las fábricas sale al menos una ruta, que los costos de inversión son enteros positivos, que es posible satisfacer la demanda de cada cliente y que no hay más fábricas que clientes.

Los formatos de entrada y salida del problema son los que siguen:

Formatos de entrada y salida:

La entrada contiene varias instancias del problema. La primera línea de cada instancia contiene los siguientes valores (enteros positivos, separados por un espacio):

$$F\ C\ R$$

donde F indica la cantidad de fábricas de la empresa (numeradas de 1 a F), C indica la cantidad de clientes (numerados de $F+1$ a $F+C$) y R indica la cantidad de rutas de la provincia. A esta línea le siguen R líneas y cada una de ellas describe una de las rutas de la provincia. Para cada ruta, la línea correspondiente contiene los siguientes valores (enteros positivos, separados por un espacio):

$$e_1\ e_2\ l$$

donde e_1 y e_2 son los extremos de la ruta y l es la longitud de la misma. Los extremos de la ruta pueden ser fábricas y/o clientes y son números enteros en el rango $[1,F+C]$. La entrada concluye con una línea comenzada por 0.

La salida debe contener una línea por cada instancia de entrada. Esta línea debe contener los siguientes valores (separados por un espacio):

$$L\ k\ e_1^1\ e_2^1\ e_1^2\ e_2^2 ...\ \ e_1^k\ e_2^k$$

donde L es el costo total de la solución (la suma de las longitudes de las rutas utilizadas), k es la cantidad de rutas utilizadas en la solución y e_1^i y e_2^i son los extremos de la i-ésima ruta utilizada, para $i \in [1,k]$.

En lo que sigue, presentaremos dos ejemplos del problema a resolver:

- **Ejemplo 1:**

  En el ejemplo que sigue, decidimos mostrar un clásico caso del problema. En éste, se puede observar que, si bien existe más de una ruta entre cada cliente y alguna fábrica, la solución devuelve una sola ruta para cada uno de ellos.

  Formato de entrada:

3 5 8
1 4 40
1 5 40
4 5 100
1 6 70
6 7 100
7 2 70
2 8 40
3 8 40

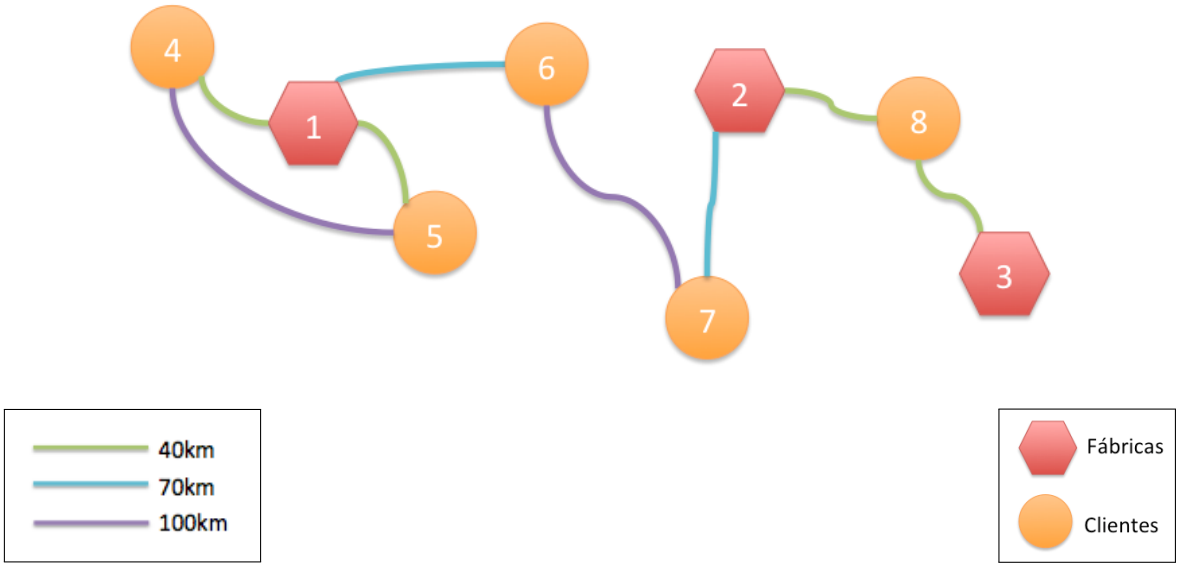


Figura 15: Ejemplo 1 - entrada.

Formato de salida:

260 5 1 4 1 5 2 8 1 6 7 2

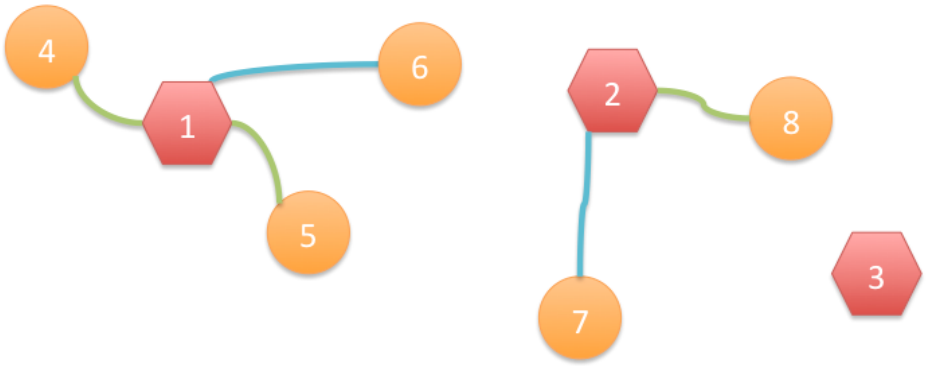


Figura 16: Ejemplo 1 - salida.

- **Ejemplo 2:**

  En este ejemplo, puede verse que si bien todos los clientes se encuentran conectados a una fábrica de forma directa, las rutas elegidas son las indirectas dado que la suma de éstas es inferior a la otra opción. Por otra parte, puede observarse que algunas fábricas resultan inutilizables luego de la minimización de rutas, como ocurre para la fábrica n°1.

  Formato de entrada:

  2 4 7
  3 4 20
  5 6 20
  4 5 20
  2 3 90
  2 4 70
  2 5 50
  2 6 20
  1 6 70

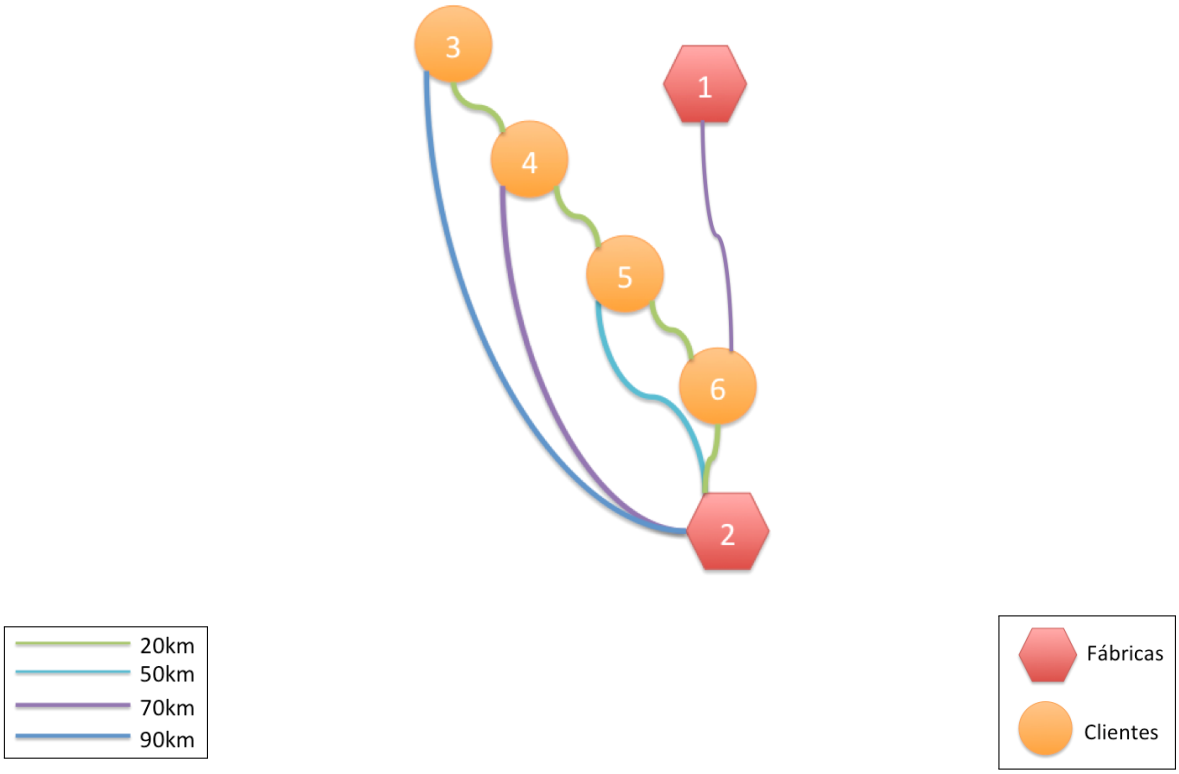


Figura 17: Ejemplo 2 - entrada.

Formato de salida:

80 4 3 4 5 6 4 5 2 6

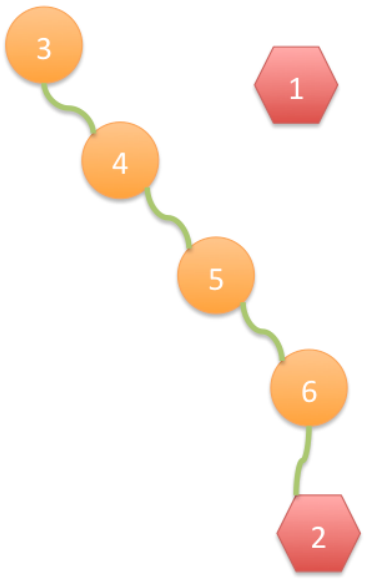


Figura 18: Ejemplo 2 - salida.

4.2. Resolución coloquial

Antes de explicar la resolución elegida para el problema descrito, formalicemos el modelo a utilizar:

- **Nodos:** Clientes y Fábricas
- **Aristas:** Rutas
- **Pesos de las aristas:** Longitud

Luego, cuando hablemos de **Grafo**, nos vamos a estar refiriendo a una provincia que contiene clientes, fábricas y rutas.

Para resolver este problema, optamos por la utilización de un algoritmo encargado de hallar un Árbol Generador Mínimo. Esto se debe a que buscamos minimizar el costo de reparación de las rutas siempre que se respete la restricción de que todos los clientes se mantengan conectados con al menos una fábrica. Luego, buscamos minimizar el peso de las aristas del grafo.

Al idealizar la resolución del problema, nos encontramos con que puede no existir un camino entre todo par de vértices (un grafo no conexo). Esto nos impidió utilizar los algoritmos vistos en clase dado que aplican sobre grafos conexos. Para solucionar dicho inconveniente, optamos por crear un nodo llamado *ghost* que se conectara con aristas de peso cero a todas las fábricas. De esta manera, nos aseguramos obtener un grafo conexo ya que todos los clientes se encuentran enlazados con al menos una fábrica y unimos todas las componentes conexas.

Una vez insertado el nodo *ghost*, se ejecuta el algoritmo de *Kruskal* a fin de hallar un árbol generador mínimo. Luego, la suma de las longitudes de todas las rutas es la menor. Posteriormente, procedemos a eliminar el nodo *ghost*, obteniendo, así, un conjunto de componentes conexas con al menos una fábrica cada una y cuyas sumas de las longitudes de rutas a reparar es la menor posible.

El pseudocódigo que resuelve el problema es el siguiente:

Algorithm 8: Minimización de Caminos entre Fábricas y Clientes

Input: Grafo G, Precios ps
Output: Grafo

```
1  NodoFabrica ghost
2  Agregar(Nodos(G),ghost)
3  forall the v ∈ Nodos(G) do
4  |  if esFabrica(v) ∧ v ≠ ghost then
5  |  |  Agregar(aristas(G),< v,ghost >)
6  |  |  ps[v,ghost] = 0
7  |  end
8  end
9  Kruskal(G,ps)
10 Filtrar(G,ghost)
11 devolver G
```

Algorithm 9: Filtrar

Input: Grafo G, Nodo $ghost$

```
1  forall the v ∈ Nodos(G) do
2  |  if v incide con ghost then
3  |  |  Eliminar arista < v,ghost >
4  |  end
5  end
```

donde $Agregar(Nodos(G), x)$ agrega el nodo x a los nodos de G. Por otro lado, $esFabrica(v)$ es una función que devuelve $true$ si v es una fábrica, caso contrario devuelve $false$.

4.3. Demostración de correctitud

Demostremos que el algoritmo planteado es correcto. Para ello, veamos que **se fortalecen las rutas necesarias al menor costo posible**. Donde **rutas necesarias** hace referencia a un conjunto de rutas tal que existe al menos un camino fortalecido entre un cliente y una fábrica. Por otro lado, **menor costo posible** hace alusión a que la sumatoria de todos los costos sea el menor posible.

Por un lado, sabemos que **es posible satisfacer la demanda de todos los clientes** implica que para todo cliente existe un camino que lo une a una fábrica [1].

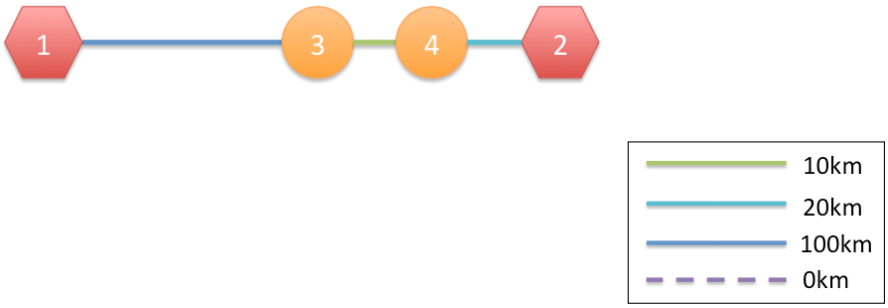


Figura 19: Ejemplo que muestra que todo cliente se une con al menos una fábrica.

Luego, queremos probar lo siguiente:

- Si agregamos aristas adyacentes entre cada fábrica y un nodo *ghost* [2] y aplicamos el algoritmo de *Kruskal*, el resultado cumple que todos los clientes están conectados con todas las fábricas (pues consiste en un árbol generador) [3]. Por otra parte, la sumatoria de todos los costos es igual a la de la solución optima (pues las aristas agregadas tenían peso cero y *Kruskal* selecciona la sumatoria de costos mínima).

  Por lo tanto, obtenemos una solución cuyo peso es mínimo, pero que mantiene las aristas de peso cero.

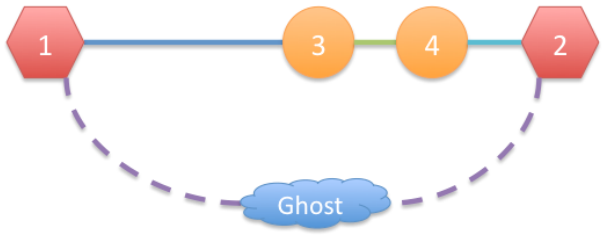


Figura 20: Ejemplo con el nodo Ghost agregado.

Luego, eliminamos las aristas que inciden en el nodo *ghost*. Dado que éstas tenían peso cero, el costo total de la solución no se ve afectado.

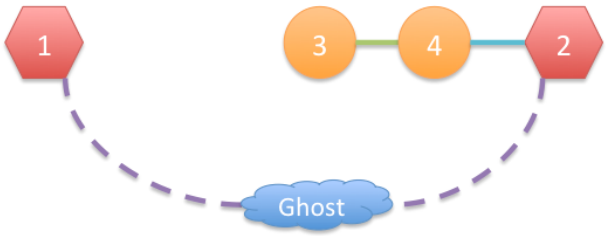


Figura 21: Ejemplo luego de aplicar el algoritmo de Kruskal.

Por último, debemos asegurarnos de que todas las componentes conexas restantes poseen al menos una fábrica. Esto se cumple ya que, por [1] todas las componentes tienen al menos una fábrica, por [3] todas las fábricas y clientes están conectados y, por [2], las aristas eliminadas inciden sólo en fábricas, por lo que eliminarlas no descarta caminos entre clientes y éstas.

Figura 22: Ejemplo luego de eliminar las aristas de costo cero.

De este modo, a partir de las rutas, fábricas y clientes ingresados como parámetro de entrada, logramos obtener el conjunto de rutas cuya suma de costos es mínima logrando que cada cliente se encuentre conectado a una fábrica, siendo ésto lo que buscábamos.

4.4. Complejidad del algoritmo

En esta sección, detallaremos la complejidad de nuestro algoritmo junto con la justificación de la elección de cada estructura.

En principio, para definir el grafo ingresado por parámetro, utilizamos un $vector < Arista >$, donde *Arista* es un struct que posee tres enteros (su peso y los nodos que conecta), y dos enteros que representan la cantidad de clientes y de fábricas. Para definir dicha estructura, reservamos ($reserve$[19]) el espacio a utilizar por el vector, siendo éste de $R + F$. Esto se debe a que, posteriormente, dicho vector almacena una suma de aristas correspondiente a la cantidad de fábricas y, de este modo, nos aseguramos de que el agregado ($push\_back$[20]) de las mismas sea $\mathcal{O}(1)$. Luego, el costo de reservar dicha memoria es $\mathcal{O}(R + F) \leq \mathcal{O}(2R) = \mathcal{O}(R)$, pues $F \leq R$ (por enunciado[21]). De este modo, el cargado de la entrada se realiza en forma lineal respecto de la cantidad de elementos, es decir, las aristas (R).

Una vez realizado esto, se comienza con la resolución del problema. Cabe aclarar que todas las funciones toman los valores por referencia, de esta manera se evitan copias innecesarias. Lo primero que realizamos es la incorporación de un nodo. Esto es constante pues consiste en sumarle uno a *cant_nodos* y retornar el valor obtenido. Luego, se incorporan las aristas incidentes al nodo *ghost*. Dicho proceso se realiza en tiempo lineal con respecto a la cantidad de fábricas ya que las recorre, crea una arista y la inserta en el vector (crear una arista es constante al igual que insertar el nodo pues el espacio se encontraba reservado previamente). Luego, la cantidad de rutas pasa a ser $R + F$.

Posteriormente, se prosigue aplicando el algoritmo de $Kruskal$ cuyo costo temporal es $\mathcal{O}(m^*log(n))$[22] donde $m = R + F$ y $n = F + C$.

Luego del algoritmo de $Kruskal$, se realiza el llamado a la función $filtrar$. Ésta comienza con la función $reserve$ ($\mathcal{O}(n)$)[23] que reserva la cantidad de rutas menos la cantidad de fábricas, luego $R+F$-$F = R$ lugares en el vector. A continuación, se recorren todas las aristas $R+F$ almacenando, en cada paso, las arista no incidentes en el nodo *ghost* en el vector creado anteriormente. Dicha comparación es constante ya que sólo se accede a un par de aristas. Por último, se le resta uno a la cantidad de nodos ($\mathcal{O}(1)$) permitiéndonos concluir que la complejidad final de esta función resulta $\mathcal{O}(R + F) = \mathcal{O}(R)$.

Finalmente, podemos concluir que la complejidad resulta $\mathcal{O}(R) + \mathcal{O}(F) + \mathcal{O}((R+F)log(F+C)) + \mathcal{O}(R) + \mathcal{O}(R) = \mathcal{O}((R+F)log(F+C)) = \mathcal{O}(2^*R^*log(2C)) = \mathcal{O}(2^*log(2)^*R^*log(C)) = \mathcal{O}(R^*log(C))$ dado que $F \leq C \leq R$.

4.5. Instancias posibles

Para verificar la correctitud de nuestro programa, dispusimos variar estratégicamente las instancias de entrada al ejecutarlo.

- En primer lugar, ingresamos una instancia que consideramos base. Ésta consiste en una única fábrica conectada a un cliente. Dado que no es posible ingresar el carácter 0 por ser el indicador del final del archivo, no pueden existir casos con valores menores a éste. De este modo, logramos evaluar un tipo de caso borde.
  Parámetro de entrada:

  1 1 1

  1 2 10

  Parámetro de salida:

  10 1 1 2

[19] http://es.cppreference.com/w/cpp/container/vector/reserve
[20] http://es.cppreference.com/w/cpp/container/vector/push_back
[21] Enunciado: "Se sabe que desde todas las fabricas sale al menos una ruta, que es posible satisfacer la demanda de cada cliente, y que no hay mas fabricas que clientes"
[22] Dicha complejidad se encuentra detallada en la sección 2,1 dado que se utilizó exactamente el mismo algoritmo.
[23] http://www.cplusplus.com/reference/vector/vector/reserve/

- Por otra parte, probamos ingresando dos clientes y una fábrica interconectados entre sí. El objetivo de esta instancia es ver que nuestro algoritmo elige, efectivamente, el camino más corto sin importar de qué modo une a los clientes con la fábrica ya que ésto resulta indistinto siempre que la componente conexa la contenga.
  Parámetro de entrada:

  1 2 3

  1 2 10

  1 3 40

  2 3 20

  Parámetro de salida:

  30 2 1 2 2 3

- También, quisimos visualizar qué ocurría cuando la entrada tenía varias soluciones posibles. De este modo, logramos ver cuál de las soluciones óptimas elegía nuestro algoritmo.
  Parámetro de entrada:

  3 2 6

  1 4 10

  1 5 10

  2 5 10

  2 4 10

  3 5 10

  3 4 10

  Parámetro de salida:

  20 2 1 4 1 5

- Para finalizar, probamos un caso compuesto por varias componentes no conexas. Para ello, creamos una instancia que posee tres pares de (clientes, fábricas) no conectadas entre ellos.
  Parámetro de entrada:

  3 3 4

  1 4 10

  1 6 10

  5 2 25

  3 6 30

  Parámetro de salida:

  45 3 1 4 1 6 5 2

De este modo, logramos abarcar los casos límite en los que la implementación pudiera haber encontrado algún problema. Dado que los resultados obtenidos fueron los esperados, determinamos que para todas las instancias válidas posibles de entrada nuestra implementación resulta correcta.

4.6. Experimentación

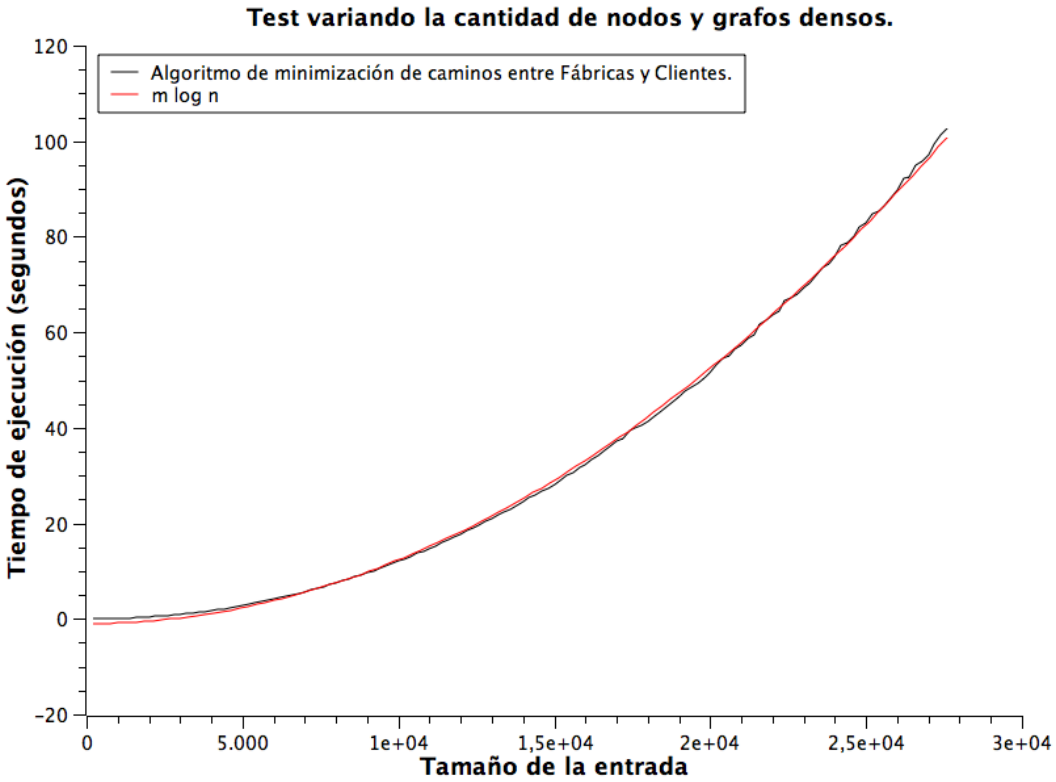
Para las pruebas de complejidad empírica, generamos instancias aleatorias de grafos alterando la cantidad de nodos y aristas. Estas instancias fueron generadas en $C++$ con la función *rand*(). La cantidad de grafos generados se comprendió entre 200 y 100000, agregando de a 200 en cada iteración. Las mediciones de tiempo en nanosegundos se realizaron con la función *high resolution clock*[24] de la librería *Chrono* de $C++$. Debido a que éstas fueron realizadas en nanosegundos, las pruebas cuyo tamaño de la entrada era menor a 200 se realizaba en mayor tiempo que las instancias más grandes pues el procesador le otorga más atención al no realizar cambio de contexto. De este modo, logramos medir las pruebas de nuestro algoritmo para comprobar que la complejidad correspondiera con la mencionada anteriormente.

Las funciones de complejidad con las que se compararon nuestros gráficos de tiempo fueron ajustadas por algoritmos matemáticos (proporcionados por **sci davis**). Dichos algoritmos se encargaron de multiplicarle y sumarle constantes a las funciones con el fin de que éstas se ajustaran a nuestros resultados sin modificar el comportamiento de las funciones utilizadas para comparar.

El algoritmo que genera las instancias garantiza que siempre va a haber al menos una fabrica conectada directa o indirectamente a un cliente, de forma tal que la instancia no se vuelva invalida.

[24] http://en.cppreference.com/w/cpp/chrono/high_resolution_clock

En una primera instancia, generamos grafos aleatorios variando la cantidad de nodos y dejando la cantidad de aristas en un valor fijo. Este valor es un porcentaje de la cantidad máxima de aristas que puede tener el grafo. El porcentaje que elegimos para determinar un grafo denso fue el 50 % de la cantidad máxima de aristas que puede tener el grafo. La cantidad de nodos que representan fabricas es del 50 %.



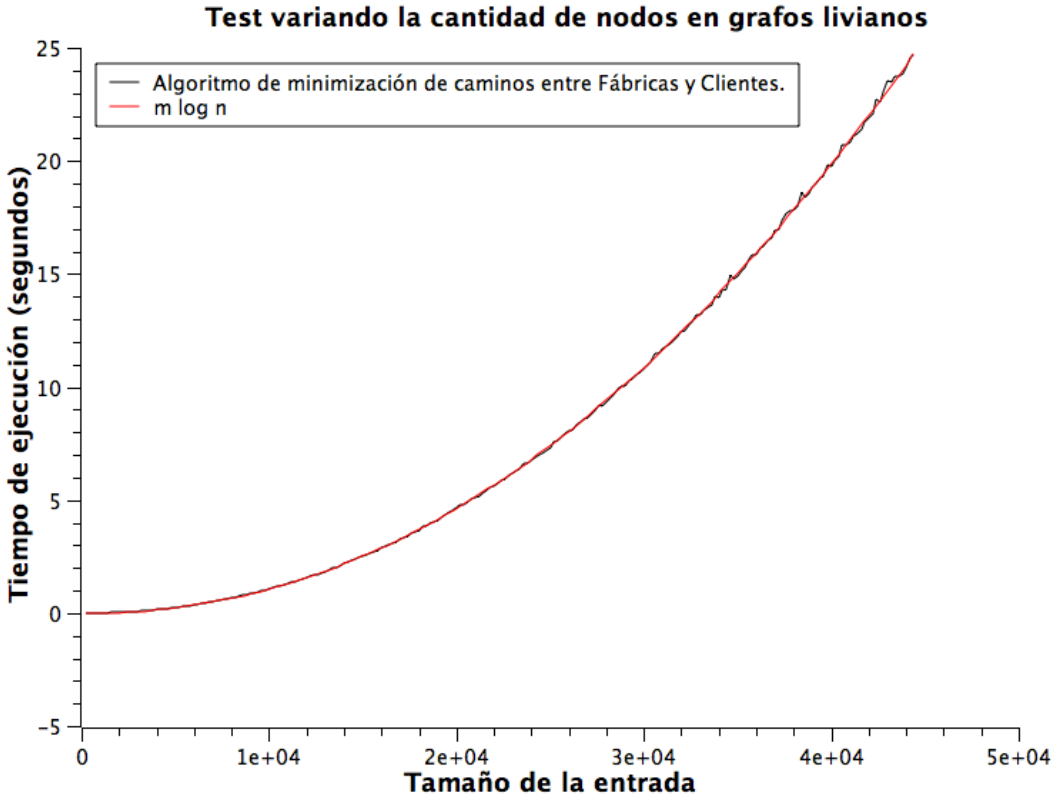
En este caso se puede apreciar como la complejidad se adapta perfectamente a los valores de entrada.
La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue

$$a * (((20 * (x * (x - 1)/2))/100) + log(x)) + b$$

. En este caso, los valores que aproximan la función a nuestros datos son:

$$desde\ x = 200\ a\ x = 27{,}600$$
$$a = 1{,}33661247156849e - 06$$
$$b = -1{,}01492247788281$$

Luego generamos un lote de pruebas pero para grafos livianos, estos a diferencia de los anteriores tienen 5 % de aristas y 5 % de nodos que representan fabricas.



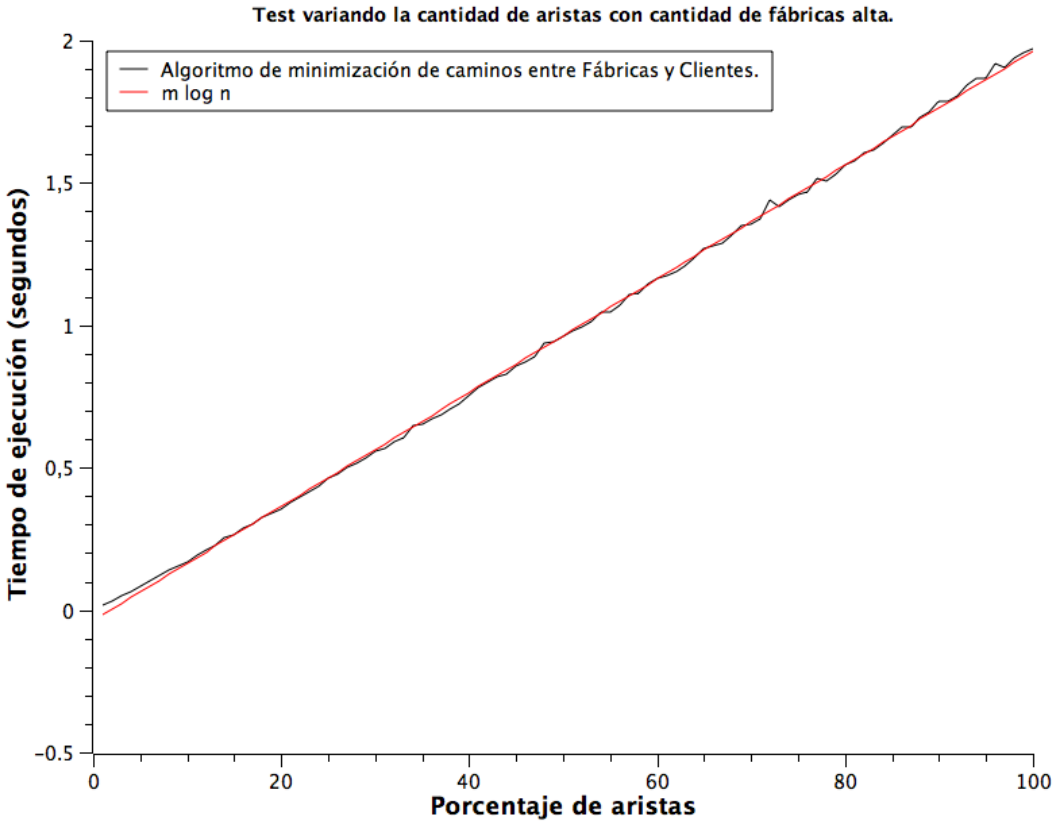
En este caso se puede apreciar como la complejidad se adapta perfectamente a los valores de entrada. Respecto al caso anterior, los tiempos de ejecución son mas rápidos con lo cual se pudieron realizar mayor cantidad de pruebas. Esto deja en evidencia que la cantidad de aristas impacta directamente en la complejidad, tal como mostramos en el análisis del mismo.
La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue

$$a * (((20 * (x * (x - 1)/2))/100) + log(x)) + b$$

. En este caso, los valores que aproximan la función a nuestros datos son:

$$desde\ x = 200\ a\ x = 44{,}400$$
$$a = 2{,}70030174291421e - 08$$
$$b = -0{,}0228951001523759$$

Luego, generamos un lote de grafos cuya cantidad de nodos esta fija, y variamos la cantidad de aristas entre 1 y 100 por ciento sobre la capacidad máxima de aristas que puede soportar el grafo. La cantidad de nodos que elegimos para este grafo es de 3000. La cantidad de fabricas en este caso definido como denso es del 20 %.



En este caso se puede apreciar como la complejidad se adapta perfectamente a los valores de entrada. Debido a que la función de complejidad fija la parte logarítmica, los valores resultantes se parecen a una función lineal, lo cual tiene sentido ya que la única variable que se modifica a lo largo de las pruebas es lineal.
La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue

$$a * ((x * (3000 * (3000 - 1)/2))/100) * log(3000) + b$$

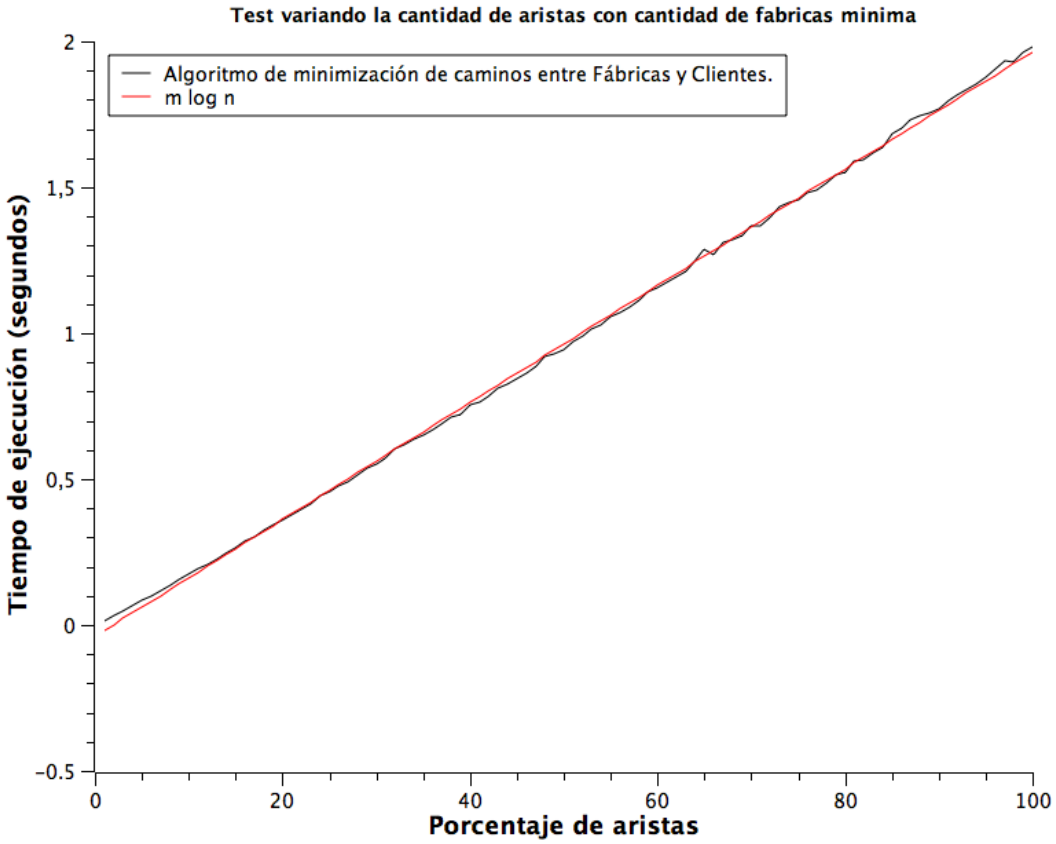
. En este caso, los valores que aproximan la función a nuestros datos son:

$$desde\ x = 1\ a\ x = 100$$

$$a = 1,27748699850107e - 07$$

$$b = -0,0359196477978645$$

Luego, generamos un lote igual al anterior excepto por el porcentaje de fabricas, esto lo hicimos para ver si la complejidad se veía afectada por la cantidad de conexiones de nodos fantasmas que el programa hacia. El porcentaje de fabricas para este caso es de 1 %.



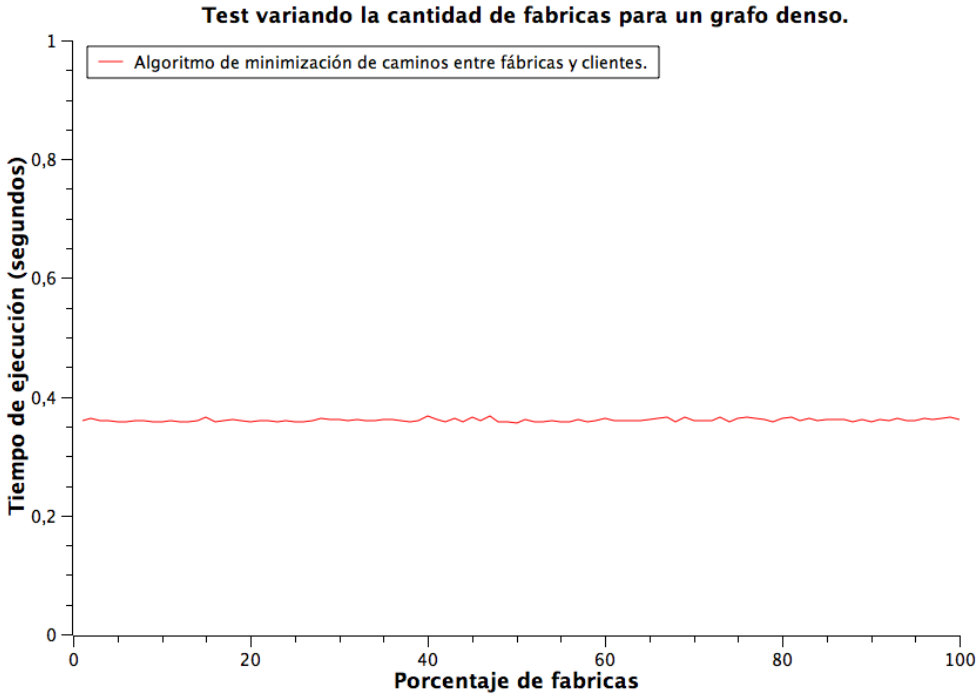
En este caso se puede apreciar como la complejidad se adapta perfectamente a los valores de entrada, y concluimos que la cantidad de fabricas no modifica la complejidad del algoritmo, siento los tiempos del grafico iguales al anterior.
La función utilizada para aproximar nuestros valores resultantes luego de las distintas ejecuciones fue

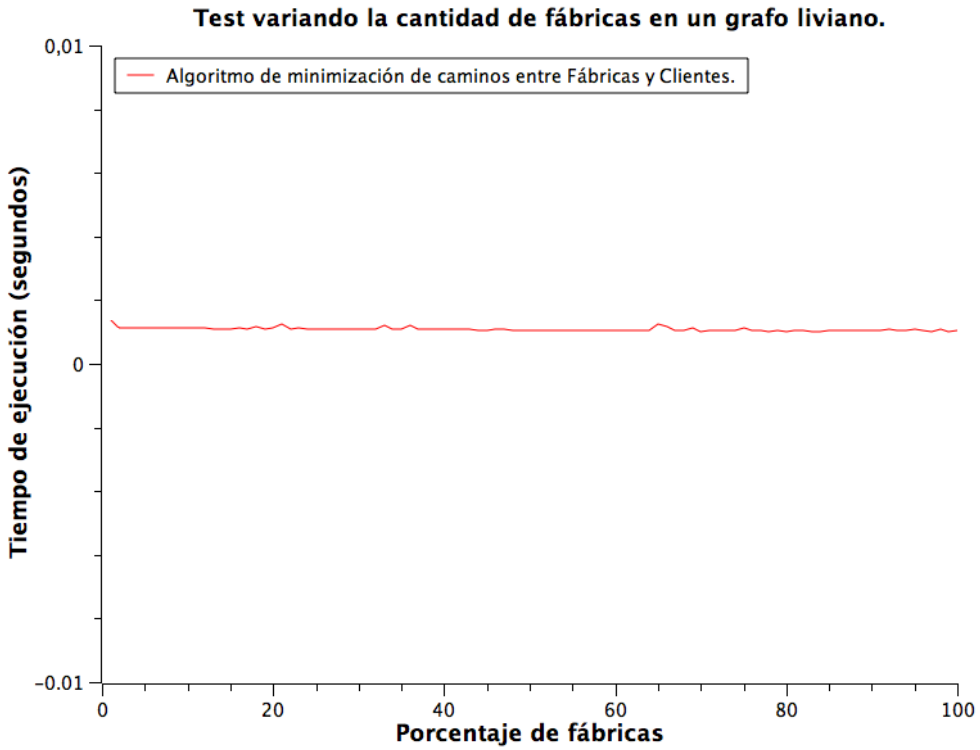
$$a * ((x * (3000 * (3000 - 1)/2))/100) * log(3000) + b$$

. En este caso, los valores que aproximan la función a nuestros datos son:

$$desde\ x = 1\ a\ x = 100$$
$$a = 1,28059038055647e - 07$$
$$b = -0,038836147808835$$

Finalmente, veamos un caso en el cual quisimos analizar el caso en el cual variáramos la cantidad de nodos que representan fabricas, para ver si esto influía en los tiempos de ejecución del algoritmo. Hicimos dos lotes, en el primero, que definimos como denso, la cantidad de aristas esta en función de la cantidad de nodos, esta cantidad es del 20 % y en el segundo lote que definimos como liviano pusimos la mínima cantidad de aristas, siendo esta el mínimo para garantizar que las instancias de los problemas sean validas. La cantidad de nodos de este grafo es de 3000.





5. Conclusión

6. Referencias

- CORMEN, THOMAS H. ; Introduction to Algorithms, Third ed. 2009. The MIT Press.
- http://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal/

7. Código fuente

7.1. Ejercicio 1:

```
int sigTrabajo = max(m1, m2)+1; //calculo el siguiente trabajo
if (sigTrabajo > cantTrabajos)
    return 0;
if (matriz[m1][m2]>-1)
    return matriz[m1][m2];
else {
    int costo1 = cs[sigTrabajo-1][m1] + minimizacionDeCostos(sigTrabajo, m2);
    int costo2 = cs[sigTrabajo-1][m2] + minimizacionDeCostos(m1, sigTrabajo);
    if (costo1 < costo2) {
        matriz[m1][m2] = costo1;
        matriz[m2][m1] = costo1;
        return costo1;
    } else {
        matriz[m1][m2] = costo2;
        matriz[m2][m1] = costo2;
        return costo2;
    }
}
```

Figura 23: Función de minimización de costos.

```
cout << minimizacionDeCostos(0,0) << " ";

int m1=0, m2=0;
vector<int> maquina1;

for (int i = 1; i < cantTrabajos; ++i)
    if (matriz[i][m2] < matriz[m1][i])
        m1 = i;
    else {
        m2 = i;
        maquina1.push_back(i);
    }

if (cs[cantTrabajos-1][m2] < cs[cantTrabajos-1][m1])
    maquina1.push_back(cantTrabajos);
```

Figura 24: Inicialización de minimización de costos.

7.2. Ejercicio 2:

```
while (!cola.empty()){
cabeza = cola.front();
cola.pop();
for (int i = 0; i < listaDeAdyacencia[cabeza-1].size(); i++){
int nodo = listaDeAdyacencia[cabeza-1][i];
if(!loAgregue[nodo-1]){
padre[nodo-1]=cabeza;
loAgregue[nodo-1] = true;
distancias[nodo-1] = distancias[cabeza-1] + 1;
cola.push(nodo);
}
}
}

res.first = cabeza;

int nodo = padre[cabeza-1];
int medio = (distancias[cabeza-1])/2;
for (int i = 0; i < cantNodos; i++){
medio--;
if (medio <= 0)
break;
nodo = padre[nodo-1];
}

res.second = nodo;
```

Figura 25: BFS.

```
int master (int cantNodos, vector<vector<int> >& listaDeAdyacencia){
pair<int, int> resDeBFS;
resDeBFS = bfs(1, cantNodos, listaDeAdyacencia);
resDeBFS = bfs(resDeBFS.first, cantNodos, listaDeAdyacencia);
return resDeBFS.second;
}
```

Figura 26: Función encargada de buscar el master.

```
res = kruskal(cantNodos, aristas);
for (int i = 0; i < res.size(); i++){
int nodoA = res[i].nodo_a;
int nodoB = res[i].nodo_b;
listaDeAdyacencia[nodoA-1].push_back(nodoB);
listaDeAdyacencia[nodoB-1].push_back(nodoA);
}
```

Figura 27: Función encargada de encontrar el AGM.

7.3. Ejercicio 3:

```
vector<Arista> ej3(vector<Arista>& adyacencias)
{
    int ghost = agregarNodo();

    for (int i=1;i<=cantFabricas;++i)
    {
Arista arista;
arista.nodo_a = ghost;
arista.nodo_b = i;
arista.peso = 0;
        adyacencias.push_back(arista); //O(1)
    }
    vector<Arista> res;
    res = kruskal(cant_nodos,adyacencias);
    res = filtrar(ghost,res);
    return res;
}
```

Figura 28: Selección de Rutas.

```
vector<Arista> filtrar(int ghost, vector<Arista>& ady) //O(R)
{
    vector<Arista> res;
    res.reserve(ady.size()-cantFabricas);
    for (int i=0;i<ady.size();++i)
    {
if (ady[i].nodo_a != ghost)
res.push_back(ady[i]);
    }
    --cant_nodos; //elimino el nodo ghost
    return res;
}
```

Figura 29: Función Filtrar.