DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico 2

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Sebastián Fernandez Ledesma	392/06	sfernandezledesma@gmail.com
Fernando Gasperi Jabalera	56/09	fgasperijabalera@gmail.com
Maximiliano Fernández Wortman	892/10	maxifwortman@gmail.com
Santiago Camacho	110/09	santicamacho90@gmail.com

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

Índice

1. Problema 1: Plan de vuelo **4**

1.1. Presentación del problema 4

1.1.1. Formalmente 4

1.2. Resolución 4

1.3. Pseudocódigo 6

1.4. Análisis de complejidad 9

1.5. Tests de complejidad 11

1.5.1. Test de casos borde 11

1.5.2. Test de análisis de peor caso 12

1.5.3. Test de generación de casos pseudo aleatorios 14

1.5.4. Comparación 14

1.6. Compilación y corrida 15

2. Problema 2: Caballos salvajes **16**

2.1. Presentación del problema 16

2.2. Resolución 17

2.3. Pseudocódigo 19

2.4. Demostración de correctitud 19

2.5. Análisis de complejidad 20

2.6. Tests de complejidad 21

3. Problema 3: La comunidad del anillo **24**

3.1. Presentación del problema 24

3.2. Resolución 24

3.2.1. Pseudocódigo 26

3.3. Demostración de correctitud 27

3.4. Análisis de complejidad 29

3.5. Tests de complejidad 31

4. Apéndice: Códigos fuente **33**

4.1. Problema 1: Plan de vuelo 33

4.2. Problema 2: Caballos salvajes 34

4.2.1. Código del algoritmo que resuelve el problema 34

4.2.2. Código del generador de instancias 36

4.2.3. Código del tester 37

4.3. Problema 3: La comunidad del anillo 38

4.3.1. Código del algoritmo que resuelve el problema 38

4.3.2. Código del generador de grafos aleatorios 43

4.3.3. Código del generador de grafos completos 43

4.3.4. Código del tester (sólo lo relevante, el resto es igual) 44

Bibliografía **47**

1. Problema 1: Plan de vuelo

1.1. Presentación del problema

Se quiere crear un sitio web para reservas de pasajes de avión. El sitio debe tener un sistema de búsqueda de itinerarios de vuelo. Un itinerario de vuelo, es una sucesión de vuelos los cuales logran llevar a un pasajero hacia un destino. Dado una ciudad de salida, y una ciudad de llegada, se busca crear un itinirario de vuelos que vaya desde la ciudad de salida hasta la de llegada.

Dentro de este itinerario, cada vuelo tiene un horario de despegue y uno de aterrizaje.

Para poder ser un itinerario válido, la ciudad de destino de un vuelo dentro del itinerario, tiene que ser la ciudad de llegada del próximo o directamente el destino final donde queriamos llegar, y en particular cada vez que llego a una ciudad debo esperar al menos dos horas desde el aterrizaje, para poder tomarme cualquier vuelo que salga desde esa ciudad, lo cual resulta lógico dado que una persona puede necesitar al menos ese tiempo para trámites de aeropuerto.

Es por eso que se nos pide el mejor itinerario, teniendo al menos dos horas entre cada vuelo, donde mejor itinerario es el que primero aterriza en la ciudad destino final.

1.1.1. Formalmente

Sea Ciudad un renombre de String. Sea Vuelo una tupla $\langle despegue, aterrizaje, origen, destino\rangle$ con $despegue, aterrizaje \in \mathbb{N}$ y $origen, destino \in Ciudad$. Dado $vuelos$ un conjunto de vuelos llamaremos $ciudadades$ al conjunto de ciudades tal que:

$$c \in ciudades \Leftrightarrow (\exists v : Vuelo)v \in vuelos \wedge (v.origen = c \vee v.destino = c))$$

Dadas $c_i, c_j \in ciudades$ definiremos un itinerario válido entre ellas como una secuencia de vuelos S con las siguientes características:

$$(S^0_{origen} = a \wedge S^n_{destino} = b) \wedge (\forall i \in \mathbb{N} : 0 \leq i < n)(S^i_{destino} = S^{i+1}_{origen} \wedge S^i_{aterrizaje} + 2 \leq S^{i+1}_{despegue}))$$

Este enunciado indica si una secuencia de vuelos en particular, no vacía, es un itinerario válido para nuestro problema. Haciendo abuso de notación diremos que una secuencia de vuelos S está incluida en un conjuntos de vuelos V y notaremos $S \subseteq V$ si todos los vuelos de la secuencia S pertenecen al conjunto V.

Contando con todas estas definiciones podemos caracterizar a una solución S del problema más formalmente de la siguiente manera:

$$(esItinerarioValido(S, a, b) \wedge S \subseteq V) \wedge$$

$$(\forall S' : esItinerarioValido(S', a, b) \wedge S' \subseteq V)S^n_{aterrizaje} \leq S'^n_{aterrizaje}$$

1.2. Resolución

Habiendo caracterizado la solución a nuestro problema en lenguaje formal, vamos mostrar una manera de obtenerla.

Sea la función:

$$boolean\ ExisteVuelo(Conj(Vuelo)\ vuelos,\ Ciudad\ inicio,\ Ciudad\ destino,\ Nat\ t) \equiv$$
$$(inicio == destino) \vee (\exists v \in vuelos)/(v_{inicio} == inicio\ \wedge v_{despegue} - 2 \geq t) \wedge$$
$$ExisteVuelo(vuelos, v_{destino}, destino, v_{aterrizaje})))$$

Esta expresión nos indica si dado un instante t, en alguna ciudad c existe una secuencia de vuelos válida que nos lleva desde c hasta la ciudad destino que queremos llegar. Solo nos va a interesar un vuelo v tal que exista un itinerario válido hasta tomarnos v y que exista un itinerario válido hasta nuestra ciudad destino a partir de tomarnos v, esta noción va a definir nuestro conjunto de vuelos válidos. Como se puede notar el enunciado de ExisteVuelo, está definido recursivamente. Dados un conjunto de vuelos C y las ciudades *inicio* y *destino* podemos definir el conjunto de vuelos válidos como:

$$(\forall v : Vuelo, v \in Validos(C, inicio, destino)) \Leftrightarrow ((v.origen == inicio \vee$$
$$(\exists v' : Vuelo, v' \in Validos(C, inicio, destino)/v'.destino == v.origen \wedge v'.aterrizaje + 2 \leq$$
$$v.despegue)) \wedge ExisteVuelo(C, v.destino, destino, v.aterrizaje) \wedge v \in C)$$

Dado este conjunto podemos definir el VueloMinimo:Vuelo, tal que VueloMinimo pertenezca al conjunto de vuelos válidos y VueloMinimo sea el primero que aterrice en cada ciudad..

Dados C:Conj(Vuelo), *inicio*:Ciudad, *destino*:Ciudad, *ciudad*:Ciudad.

$$VueloMinimo(C, inicio, destino, ciudad) \equiv v : Vuelo \Leftrightarrow (v \in Validos(C, inicio, destino) \wedge$$
$$v.destino == ciudad \wedge (\forall v' : Vuelo, v' \in Validos(C, inicio, destino) \wedge$$
$$v'.destino == ciudad)\ \Longrightarrow\ v.aterrizaje \leq v'.aterrizaje)$$

Nuestro algoritmo va a obtener este VueloMinimo v para cada ciudad, tal que v pertenezca al conjunto de vuelos válidos desde inicio hasta destino . Las caracteristica de VueloMinimo es que es el primero de los vuelos que llegan a una ciudad por un itinerario válido desde inicio hasta destino.

En particular se puede observar que, cualquier itinerario válido que contenga al vuelo mínimo de la ciudad destino, es decir a VueloMinimo(C,inicio,destino,destino) en su último vuelo va a ser solución de nuestro problema.

Esto es ya que no va existir un itinerario válido que aterrice antes en destino, que cualquier itinerario válido que contenga a ese vuelo.

Luego la solución s:Sec<Vuelo> que se devuelve como itinerario de vuelos por nuestro algoritmo, cumple:

$$esItinerarioValido(s, inicio, destino) \wedge s \subseteq C \wedge$$
$$(\forall v : Vuelo, v \in s)(s == VueloMinimo(C, inicio, destino, s.destino))$$

1.3. Pseudocódigo

Nuestro algoritmo final quedaria de la siguiente manera:

1. Definimos Color como un enum [BLANCO,ROJO,VERDE]
2. Definimos un Aeropuerto como un tupla

$$\langle ciudad : Ciudad, vuelosQueSalen : Conj(Vuelo), vuelosQueLlegan : Conj(Vuelo), \\ id : Nat, primerVueloQueSaleYLlega : Vuelo, ultimoVueloQuellega : Nat \rangle$$

Algorithm 1 Sec⟨Vuelo⟩ itinerario(Conj(Vuelo) vuelos, Ciudad inicio, Ciudad final)

```
1: Aeropuerto[] aeropuertos = construirArrayDeAeropuertos(vuelos,inicio,final)
2: Sec⟨Vuelo⟩ itinerario = []
   /* Esta sesgada la funcion con T = -2 ya que vamos a preguntar si existeVuelo a partir
   de T = 0. En la posicion 0 se encuentra el aeropuerto de la ciudad inicial, y en la posición
   1, el de la ciudad final */
3: if (existeVuelo(aeropuertos, aeropuertos[0],aeropuertos[1],-2)) then
4:    Vuelo min = aeropuerto[1].primerVueloQueSaleYLlega
5:    itinerario.agregar(min,0)
6:    while (min.inicio != inicio)  do
7:       Vuelo min = vueloMinimo.primerVueloQueSaleYLlega
8:       itinerario.agregar(min,0)
9:    end while
10: end if
11: return  itinerario
```

La siguiente función obtiene el conjunto de vuelos válidos desde una ciudad inicio hasta una ciudad destino, en el instante T, si el conjunto existe devuelve TRUE, sino FALSE.

Algorithm 2 boolean existeVuelo(Aeropuerto[] aeropuertos, Aeropuerto inicio, Aeropuerto final, int t)

```
1: if (inicio == destino) then
2:    return TRUE
3: end if
4: if (t + 2 ≤ inicio.ultimoVueloQueLlega()) then
5:    return TRUE
6: end if
7: boolean llego = false;
8: Conj(vuelo) vuelos = inicio.vuelosQueSalen;
9: inicio.vuelosQueSalen = ∅;
10: Conj(vuelo) vuelosNoAnalizados = ∅;
11: for (Vuelo v : vuelos) do
12:    if (v.despegue ≥ t+2) then
13:       if (v.color == BLANCO) then
14:          if (existeVuelo(aeropuertos,v.destino,final,v.aterrizaje) then
15:             llego = true
16:             v.color = VERDE
17:             if (inicio.ultimoVueloQuellega < v.llegada) then
18:                inicio.ultimoVueloQuellega = v.llegada
19:             end if
20:             if (vuelo.destino.primerVueloQueSaleYLlega.llegada > v.llegada) then
21:                vuelo.destino.primerVueloQueSaleYLlega = v
22:             end if
23:          else
24:             v.color = ROJO
25:          end if
26:       end if
27:    else
28:       vuelosNoAnalizados.agregar(v)
29:    end if
30: end for
31: inicio.vuelosQueSalen = vuelosNoAnalizados;
32: return llego
```

Esta función se llama recursivamente con los vuelos de la ciudad inicio. El caso base de la función es cuando llega a la ciudad destino o cuando llega a un camino de vuelos válidos que se pueda tomar desde el instante T, entonces devuelve true.

Si itera por todos los vuelos de ciudad inicio, y si ningún vuelo llega a destino ni sus llamadas recursivas respectivas, entonces devuelve false.

Cada vez que detecte que un vuelo v llega al destino o a un camino de vuelos que me lleven a destino, colorea de verde a v y actualiza dos datos:

Un puntero en v.destino, al primer vuelo que aterriza en v.destino y llega al destino.

Un Nat que representa el instante de aterrizaje del último vuelo que aterriza en Inicio, y llega a destino.

Luego el conjunto de vuelos que estan contenidos en algún itinerario válido son los que marco de verde.

La función es recursiva, para poder cumplir la complejidad requerida, creamos un puntero a la lista de vuelos de un aeropuerto, y en caso de tener que hacer una llamada recursiva, se apunta a null esa lista, para que dado que entro en una recursion de un vuelo, no volver a consultar por los mismos vuelos de una ciudad. Esto se basa en el principio de que si dentro de una recursión visito la misma ciudad, el T de la primera recursión va a ser menor al de la segunda, por lo tanto los vuelos que me pueda tomar en la segunda, ya voy a poder evaluarlos en la primera.

Si existe un vuelo que llega a la misma ciudad desde una recursión, se genera un ciclo.

Ese vuelo en particular no nos interesa, ya que cualquier camino de vuelos que pueda existir a partir del instante de llegada de ese vuelo a la ciudad C, tambien lo voy a poder hacer con el instante de llegada de la primera recursion a C, ya que es estrictamente menor por lo tanto todos los vuelos que estén disponibles en la segunda iteración también van estarlo en la primera.

Por lo tanto dado que se entra en una recursión desde una ciudad inicio, se vacian los vuelos que salen de esa ciudad, para no volver a iterarlos dentro de la recursión y no generar este ciclo.

La siguiente funcion construye un array de aeropuertos, en complejidad O(N.Log(N))

Algorithm 3 Aeropuerto[] construirArrayDeAeropuertos(Conj(Vuelo) vuelos, Ciudad inicio, Ciudad final)

```
1: Conj(Ciudad) ciudades = null
2: ciudades.agregar(inicio)
3: ciudades.agregar(final)
4: for (Vuelo vuelo: Vuelos) do
5:    ciudades.agregar(vuelo.origen)
6:    ciudades.agregar(vuelo.destino)
7: end for
8: Aeropuerto[] aeropuertos = new Aeropuerto[ciudades.size];
9: Mapa⟨Ciudad, Nat⟩ ids = ∅
10: ids.agregar(inicio,0)
11: ids.agregar(final,1)
12: aeropuertos[0] = new Aeropuerto (inicio,∅,∅,0)
13: aeropuertos[1] = new Aeropuerto (final,∅,∅,1)
14: ciudades.remover(inicio)
15: ciudades.remover(final)
16: Nat i = 2
17: for (Ciudad ciudad: ciudades) do
18:    aeropuertos[i] = new Aeropuerto (ciudad,∅,∅,i)
19:    ids.agregar(ciudad,i++)
20: end for
21: i = 1
22: for (Vuelo vuelo: Vuelos) do
23:    vuelo.id = i++
24:    vuelo.color = BLANCO
25:    int id = ids.obtener(vuelo.origen)
26:    aeropuertos[id].vuelosQueSalen().agregar(vuelo)
27:    id = ids.obtener(vuelo.destino)
28:    aeropuertos[id].vuelosQueLlegan().agregar(vuelo)
29: end for
30: return ciudades
```

Esta funcion va a asignarle un id a cada una de las ciudades y luego construir un array donde en la posicion i, se encuentra la ciudad con id = 1.

Implementada sobre un trie o sobre un hashmap, para el diccionario de ids me daria una complejidad al algoritmo de crear el mapa de aeropuertos cercana a O(n) en n numero de vuelos. Los nombres de la ciudad tienden a ser Strings de bajo costo de procesamiento en un trie, y en un hashmap la cantidad total de ciudades con aeropuertos en el mundo, es un numero relativamente bajo, por lo tanto se podría implementar de forma eficiente. Nuestra implementación usa un treeset, asi que la complejidad de crear nuestros aeropuertos es O(n.log(n))

1.4. Análisis de complejidad

Nuestra implementación consta de tres partes:

1. Construir el array de aeropuertos.

2. Encontrar el conjunto de vuelos válidos desde la ciudad origen hasta la ciudad destino.

3. Obtener un secuencia valida de Vuelos mínimos.

1.

Para construir el array de aeropuertos, lo que se hace es contar la cantidad de ciudades para crear un array de ese tamaño. Para ello, se carga cada ciudad en un TreeSet. Como la cantidad de ciudades la podemos acotar por la cantidad de vuelos, ya que a lo sumo tenemos 2 ciudades por vuelo, sea N la cantidad de vuelos, la cantidad de ciudades maximas es de 2.N, por lo tanto el costo de crear este Set es a lo sumo O(N.Log(N))

Crear el array tiene una complejidad temporal de O(N) a lo sumo y espacial de 2.N.

Luego se le asocia un id a estas ciudades, que va a ser la posicion donde se van a encontrar en el array. Para poder asociarle este id se construye un mapa $\langle Ciudad, ID \rangle$. Buscar en este mapa nos va a costar O(Log(N)) y construirlo O(N.Log(N)).

Luego se asocio cada vuelo a la posicion donde se encuentran la ciudad destino e inicio de ese vuelo. Para ello se busca el id en el mapa. Entonces para asociar todos los vuelos tenemos la complejidad es de O(N.Log(N)).

Luego la complejidad final de construir el array es O(N.Log(N)).

2.

Sea i una llamada recursiva del algoritmo.

Sea $Inicio_i$:Ciudad, la ciudad inicial de la recursion i.

Antes de empezar a iterar por los vuelos que salen de $Inicio_i$, se borra el puntero a este conjunto en $Inicio_i$.

Si vuelvo a $Inicio_i$ en una instancia de recursion a partir de i,puedo asegurar que los vuelos que salen de $Inicio_i$, no van a estar incluidos.

Es decir, no existen ciclos en la recursión, donde un ciclo seria volver a iterar por un vuelo que se haya iterado anterioremente.

Entonces para cualquier recursión, se chequean a lo sumo todos los vuelos de cada ciudad.

Todos los vuelos de cada ciudad son exactamente N vuelos.

Cuando se genera una llamada recursiva se colorea el vuelo, por lo tanto ese vuelo nunca más va a ser llamado recursivamente ya que se conoce el estado para cualquer instante de tiempo.

Entonces el algoritmo puedo tener tantas llamadas recursivas como vuelos distintos tenga.

El numero de vuelos distintos es N.

Y para cada llamada recursiva a lo sumo se itera exactamente una vez por cada vuelo.

Por lo tanto la cantidad total de vuelos que se iteran dado que se llego a una ciudad, en un instante T esta acotado por la cantida total de llamadas recursivas y la cantidad de vuelos que a lo sumo se iteran en cada una.

Existen a lo sumo N recursiones ya que existen a lo sumo N vuelos que puedo llamar recursivamente, y una recursion tiene un complejidad a lo sumo de O(N) para conocer su estado.

Luego la cantidad de iteraciones máxima que se realizan tiene una complejidad a lo sumo $O(N^2)$.

3.

Para construir nuestra secuencia de vuelos solución vamos a iterar por los vuelos mínimos de cada ciudad que llegan a destino. Un vuelo minimo para una ciudad C, es de todos los vuelos que llegan a C y pertenecen a un itinerario válido desde inicio hasta la ciudad final, el primero que aterriza en C.

Suponemos que no existe vuelos que ciclan a la misma ciudad, es decir que su destino y origen es el mismo. (informalmente podemos descartar estos vuelos porque si existe una solucion para ellos, existe una solucion para otro vuelo pero que no genere un ciclo).

Sea s un secuencia de vuelos, tal que s es un itinerario válido de vuelos minimos.

$$(\forall i, j : Nat, i, j < S.tamanio \wedge i < j)(S_i.aterrizaje < S_j.despegue \vee S_i.destino == final)$$

Luego si S contiene un ciclo, va a existir al menos un vuelo, que pertenezca a ese ciclo,(en particular va a ser el vuelo que cierre ese ciclo), no va a cumplir este invariante, ya que el aterrizaje de un vuelo es siempre mayor a su despegue (en particular por como lo definimos podría existir más de un vuelo mínimo por ciudad, pero igual fallaría ya que el horario de aterrizaje debería ser igual para todo vuelo mínimo de una ciudad).

Por lo tanto un itinerario válido de vuelos mínimos no tiene ciclos.

El itinerario válido mas largo que puede existir sin ciclos es a lo sumo de tamaño N.

Por lo tanto obtener el itinerario cuesta a lo sumo O(N).

Luego la complejidad total del algoritmo es O(N.Log(N)) + $O(N^2)$ + O(N) = $O(N^2)$

1.5. Tests de complejidad

Para analizar y testear tanto complejidad como correctitud del algoritmo implementado para la resolución del problema realizamos los siguientes testeos:

- Test de casos borde.
- Test de análisis de peor caso.
- Test de generación de casos pseudo aleatorios.

1.5.1. Test de casos borde

Los siguientes test fueron implementados:

- testUnVuelo() = Chequea dado un conjunto de vuelos, tal que todos los vuelos salen de inicio, que la solución sea el de menor aterrizaje.
- testSinVuelos() = Dado un conjunto vacio de vuelos, asegura la solución es "no".
- test2HorasDeDiferenecia() Asegura que la solución contenga 2hs de diferencia entre vuelo y vuelo.
- testCiclosALaMismaCiudad() = Asegura dado que existe un ciclo a la misma ciudad, el algoritmo devuelva una solución válida.
- testLlegoPorUnCaminoAntesQueDerecho() = Asegura que la solución sea un el camino que más temprano llegue y no es mś corto.
- testCiclos = Asegura dado que existen ciclos en mi conjunto de vuelos (es decir vuelvo a la misma ciudad desde un vuelo) el algoritmo retorne una solución válida.

1.5.2. Test de análisis de peor caso

Una instancia donde encontramos que la complejidad iba a tender a nuestra cota de $O(N^2)$, es cuando una la ciudad inicial tiene $\frac{N}{2}$ vuelos, que aterrizan en C:Ciudad, y luego C tiene $\frac{N}{2}$ vuelos que van desde C hasta cualquier otra ciudad con un horario de despegue menor a cualquier vuelo desde la ciudad inicial.

Como el algoritmo no puede saber en cada recursión si va a existir un vuelo válido para el T de entrada, tiene que iterar por todos los vuelos de C. Como desde inicio salen $\frac{N}{2}$ vuelos, la complejidad de la función existeVuelo es $\frac{N^2}{4}$.

Para el gráfico siguiente se generaron entradas diferentes de N y para cada entrada se corrió el algoritmo un numero K de veces.

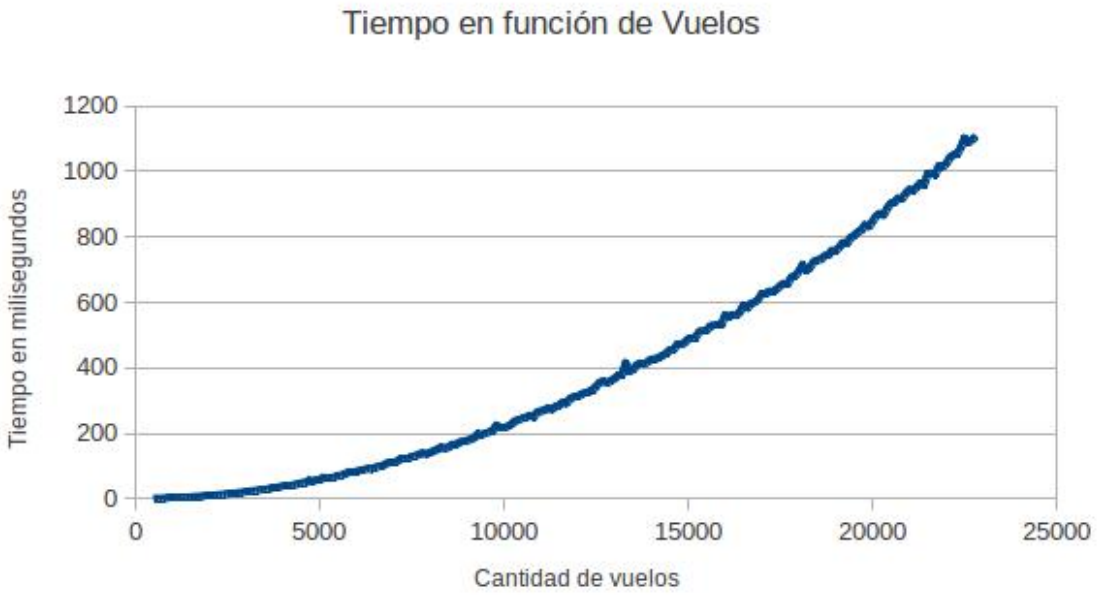
Para cada entrada de tamaño S, se tomo el tiempo de corrida de esa entrada, y de todas las instancias de igual tamaño se considero el menor tiempo en el que el algoritmo devolvía una solución.

Como se corrió dentro de una máquina deterministica, este es el mejor tiempo que se pudo obtener para cada instancia.

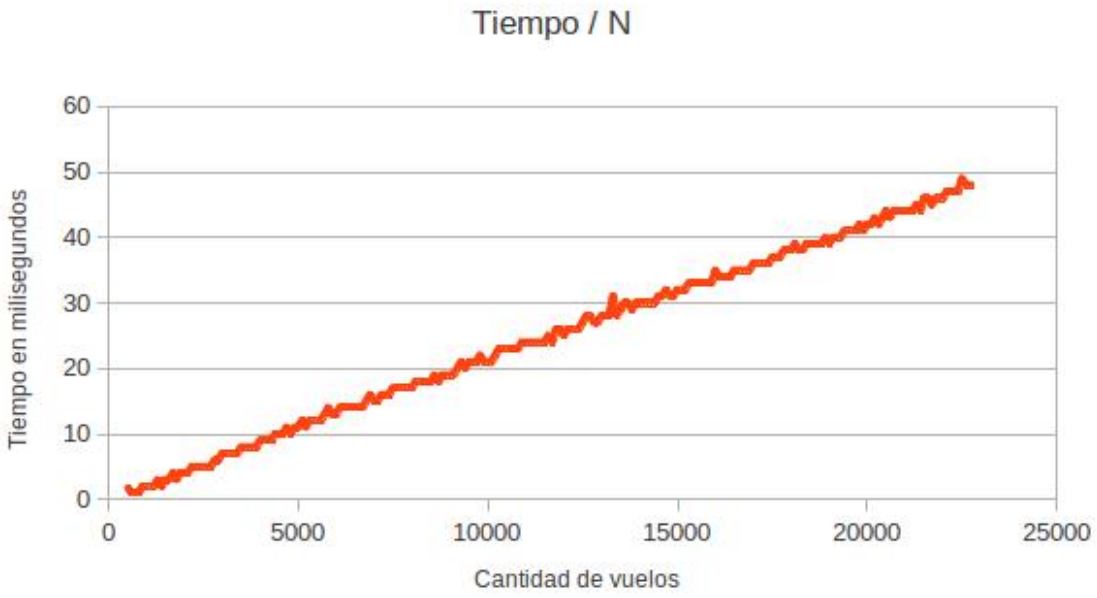
Para un número N creciente de vuelos, se corrio 20 veces cada instancia de N elementos, y se calculo el menor tiempo de resolución de esas corridas.

El número de vuelos se movió entre 500 y 20000.

Luego se obtuvo el siguiente gráfico:



Si se divide la muestra por N debería quedar una recta como se observa en el siguiente gráfico:



1.5.3. Test de generación de casos pseudo aleatorios

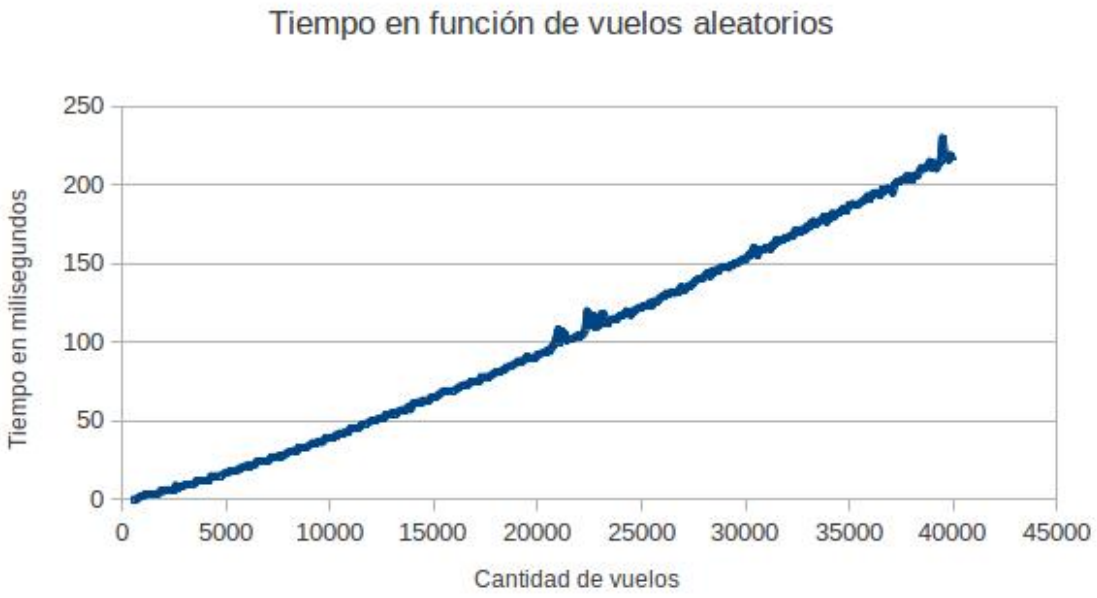
Se generó instancias pseudo-aleatorias de entrada para el algoritmo.

La pseudo-aleatoridad se basaba en generar N vuelos aleatorios, es decir vuelos donde el punto de partida y el de aterrizaje sean un elemento al azar, distintos dentro de un conjunto acotada de ciudades.

Tomamos la cota para la cantidad de ciudades como $\frac{N}{20}$. Esta cota se da para obtener instancias relativamente reales, y no es las cuales haya muchos vuelos que el algoritmo nunca recorra (estos casos deberían tener un tiempo relativamente menor para obtener una solución).

Se llamó al algoritmo k veces, con $k = 20$, y de esas corridas se eligió la de menor tiempo de ejecución.

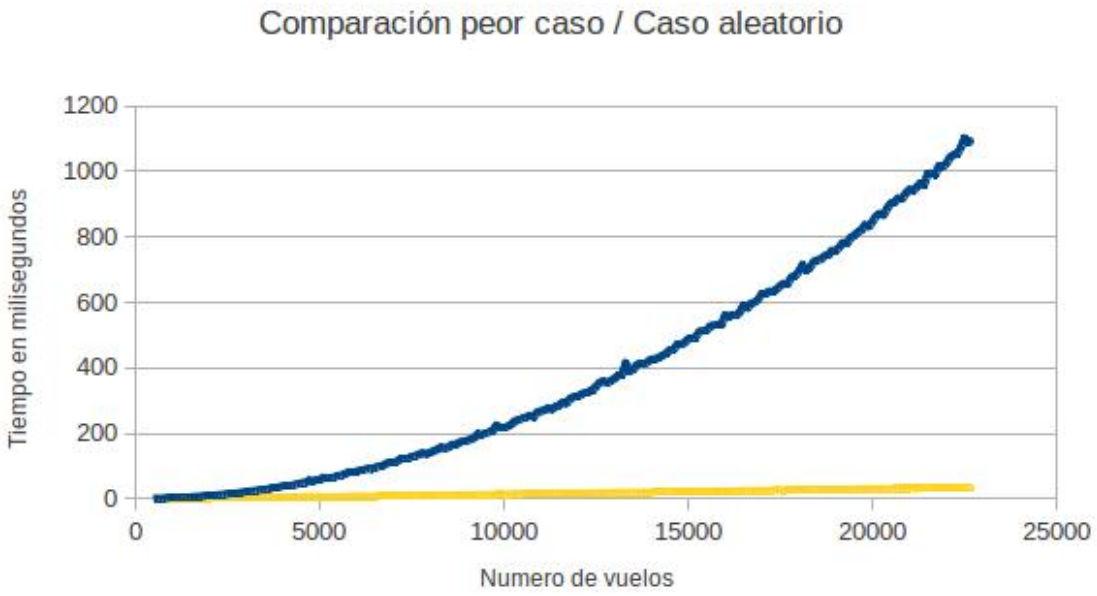
Para N moviendose entre 500 y 40000, y k cantidad de corridas $k = 10$, se obtuvo el siguiente gráfico.



1.5.4. Comparación

Luego se corrió y graficó la salida, nuestro algoritmo sobre una instancia de cada tipo aumentando el tamaño de entrada en cada iteración. Nuevamente se corrio k veces cada instancia, y se tomo el tiempo mínimo de corrida para cada una de esas interaciones. Si nuestras conclusiones eran correctas, las instancias aleatorias debían tener menor o igual tiempo de corrida que cualquiera del peor caso.

Se obtuvo el siguiente gráfico:



Como se puede apreciar en el gráfico, para toda instancia aleatoria el T que obtuvimos fue menor al del peor caso.

1.6. Compilación y corrida

Hay dos clases main que se pueden correr para llamar a nuestro algoritmo, una para cargar los datos de vuelos en un archivo y otra para cargarlos desde consola.

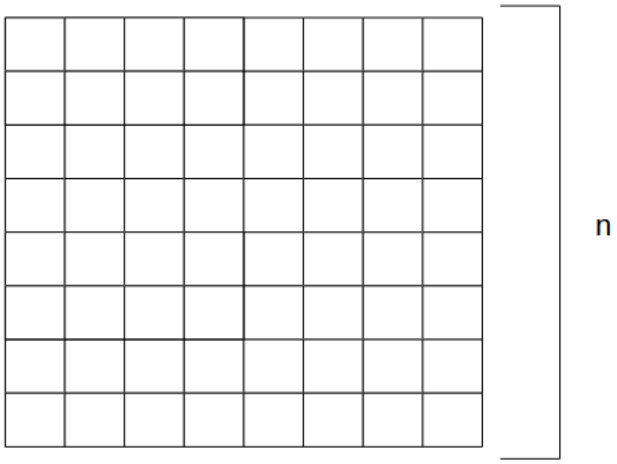
Si se quiere cargar los datos en un archivo y luego llamar al algoritmo la clase la cual debemos correr es Main y deberíamos completar el archivo "vuelos.txt"que se encuentra en el directorio base del proyecto.

Si se quiere cargar los datos en por consola y luego llamar al algoritmo la clase la cual debemos correr es MainFromConsole.

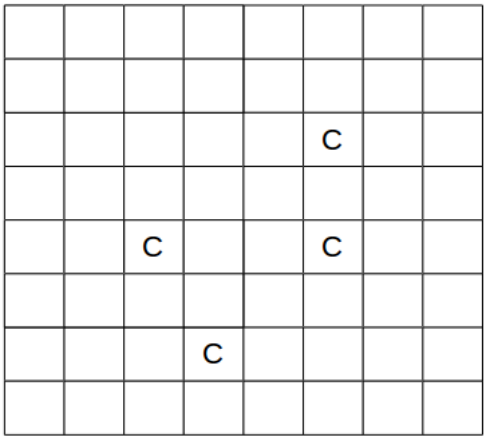
2. Problema 2: Caballos salvajes

2.1. Presentación del problema

En este problema nos presentan un tablero de dimensiones conocidas pero no acotadas del mismo tipo que el de ajedrez, es decir un tablero cuadriculado y cuadrado:



y un conjunto de caballos, piezas de ajedrez, en casillas también conocidas, por ejemplo:



el objetivo es reunir a todos los caballos en una misma casilla del tablero con una cantidad de movimientos total, es decir sumando los movimientos que le tomo a cada uno de los caballos, mínima. Los movimientos permitidos a los caballos son los mismos que los impuestos por las reglas de ajedrez:

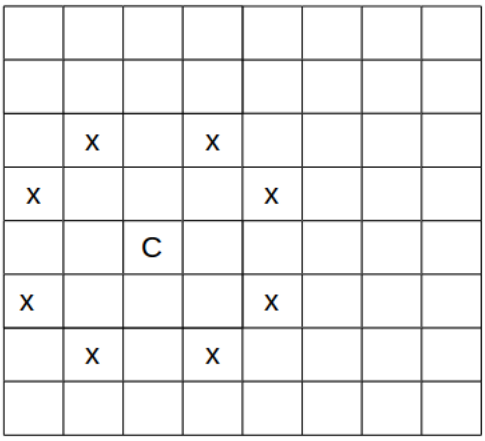


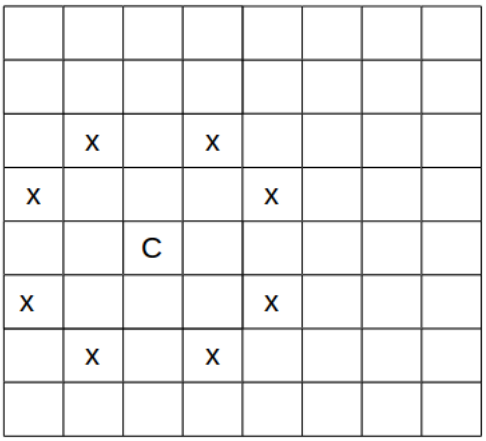
Figura 1: La C representa la casilla donde está ubicado el caballo y X los lugares a los cuales podría saltar.

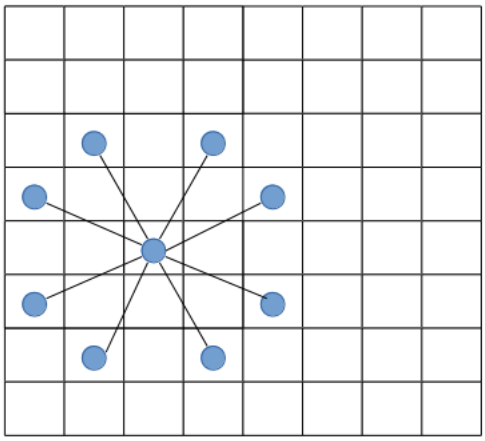
A continuación mostramos un tablero de 4×4 en el cual los caballos comienzan en las esquinas, donde aparecen las C pequeñas. Los números dentro de las casillas representan la suma de las distancias de los caminos mínimos de cada uno de los caballos hasta esa casilla. Por ejemplo, la casilla correspondiente a la fila 2 y columna 4 (contando las filas desde abajo para arriba y las columnas de izquierda a derecha comenzando ambas desde 1) tiene el número 2 porque los dos caballos pueden llegar a ella en sólo 1 movimiento. Por lo tanto la suma en este caso es $1 + 1 = 2$. Las casillas que están coloreadas se corresponden con las posibles soluciones de la instancia, son todas mínimas.

C 2	6	4	10
6	8	2	4
4	2	8	6
10	4	6	C 2

2.2. Resolución

Nosotros modelamos el problema con grafos con lo cual resultó equivalente al de camino mínimo con un origen y múltiples destinos en un grafo no dirigido sin peso en las aristas. En este caso en particular las aristas no tienen peso porque el salto de un caballo siempre tiene el mismo costo. En un grafo con las características mencionadas el algoritmo de Breadth First Search [1, p. 594] obtiene el camino mínimo entre el origen tomado y el resto de los nodos, éste es el camino entre el origen y el nodo en árbol generado. El modelo consiste en tomar a las casillas del tablero como nodos del grafo, las casillas en las que hay caballos como nodos origen y existe una arista entre dos nodos, casillas, si de una a la otra se puede ir con un salto de caballo. Cabe destacar que el salto de caballo es simétrico: si con un salto de caballo se puede ir de la casilla a a la b entonces necesariamente también se puede con un salto de caballo ir de la casilla b a la a:





Los pasos para resolver el problema una vez modelado son bastante directos:

1. Recorrer el grafo una vez por cada caballo presente tomando como origen el nodo, casilla, en la que se encuentra el caballo.
2. Guardar, en cada nodo, la cantidad de saltos que le toma a cada caballo llegar a él.
3. Recorrer todos los nodos y quedarse con alguno de los de suma mínima.

Hay un conjunto pequeño de instancias para los cuales la solución del problema es trivial:

$n = 1$ los caballos sólo pueden estar en una casilla por lo cual la solución es la única casilla que existe y la cantidad de movimientos totales es 0.

$n = 2$ los caballos no pueden realizar movimiento permitido alguno. Sólo existe solución en el caso en el que los caballos se encuentran todos en la misma casilla.

$n = 3$ existe solución siempre excepto cuando un caballo comienza en la casilla central del tablero en cuyo caso no tiene movimientos permitidos y del resto de las casillas no existe forma de llegar a la casilla central mediante movimientos permitidos, lo cual es trivial si vemos que desde la casilla central no existen movimientos permitidos porque como ya mencionamos los movimientos del caballo son simétricos.

$n > 4$ se puede probar fácilmente que de cualquier casilla del tablero se puede acceder a cualquier otra, por lo tanto cuando la dimensión del tablero es mayor a 4 siempre existe solución.

En el pseudocódigo no mostramos el manejo de instancias que poseen un tablero de dimensión menor a 4 porque al ser muy particulares no aportan mayor claridad a la presentación de la idea central del algoritmo.

2.3. Pseudocódigo

Algorithm 4 caballos_salvajes

```
distancias ← Matriz(n, n)
tablero ← Tablero(n, n)
caballos ← Conj(Casilla)
while caballos ≠ ∅ do
  origen ← caballos.next()
  BFS(origen, tablero, distancias)
end while
(suma_min, casilla_min) ← MIN(distancias)
```

Algorithm 5 BFS

```
nodos_novisitados ← Cola
push(nodos_novisitados, origen)
tablero_origen ← 0
vecinos ← Cola
while nodos_novisitados ≠ ∅ do
  nodo_actual ← pop(nodos_novisitados)
  vecinos ← obtener_vecinos(nodo_actual)
  while vecinos ≠ ∅ do
    vecino ← pop(vecinos)
    if neg esta_marcado(tablero, vecino) then
      tablero_vecino ← tablero_nodo_actual+1
      distancias_vecino ← distancias_vecino + tablero_vecino
      push(nodos_novisitados, vecino)
    end if
  end while
end while
```

2.4. Demostración de correctitud

Una instancia del problema está dada por la dimensión del tablero n y la posición de los k caballos. Luego de modelarlo tenemos un grafo $G = (V, E)$ tal que $|V| = n^2$, uno por cada nodo, y $|E| < n^2 * 4$. La cota superior sobre las aristas proviene del hecho de que un caballo sólo pude saltar a 8 casillas diferentes como máximo, asumiendo que todos los saltos posibles caen dentro del tablero. Por lo tanto, la cantidad de aristas debe ser menor a $\frac{n^2*8}{2}$ porque en muchas casillas, como las de las esquinas o las de las bandas, la mayoría de los saltos posibles caen fuera del tablero. Los caballos los representaremos con un multiconjunto de nodos al que llamaremos *origenes*, $|origenes| = |caballos| = k$. Utilizamos un multiconjunto para representarlos porque inicialmente puede haber más de un caballo en una casilla determinada. Sea $PS_{v,u}$ el conjunto de los caminos posibles entre el nodo u y el nodo v. Dado un nodo v el conjunto de soluciones posibles asociado a v es SP_v:

$$SP_v = \left\{ \sum_{u \in origenes} |P_{u,v}| : P_{u,v} \in PS_{u,v} \right\}$$

La solución óptima para un nodo v la denominaremos SP_v^{opt} y está determinada de la siguiente forma:

$$SP_v^{opt} = \text{mín}\,\{s : s \in SP_v\}$$

Lema 1. *Una solución óptima para un nodo v está compuesta por caminos mínimos entre los orígenes y v:*

$$s = SP_v^{opt} \Leftrightarrow s = \sum_{u \in origenes} \text{mín}\,\{|P| : P \in PS_{u,v}\}$$

Demostración. Sea SP_{opt} una solución óptima para el nodo v que contiene un camino $P_{w,v}$ no mínimo. Sea $P_{w,v}^{min}$ un camino mínimo entre $w \in origenes$ y v. Sea SP'_{opt}:

$$SP'_{opt} = SP_{opt} \setminus P_{w,v} \cup \left\{P_{w,v}^{min}\right\}$$

veamos el costo de SP'_{opt} expresado en función del de SP_{opt}:

$$SP'_{opt} = \sum_{u \in origenes} |P_{u,v}| - |P_{w,v}| + |P_{w,v}|$$

pero dado que $P_{w,v}$ no es mínimo:

$$P_{w,v} > P_{w,v}^{min} \Rightarrow P_{w,v} - P_{w,v}^{min} > 0$$

pero entonces volviendo a la ecuación anterior nos queda que:

$$SP'_{opt} < SP_{opt}$$

Lo cual es absurdo porque partimos suponiendo que SP_{opt} era un solución óptima y por lo tanto mínima. □

Finalmente, caracterizaremos a la solución del problema S:

$$S = \min_{v \in V} \left\{SP_v^{opt}\right\}$$

Nuestro algoritmo calcula el peso del camino mínimo entre cada uno de los orígenes y el resto de los nodos [1, p. 594] Luego suma en cada nodo v el peso de los caminos mínimos desde cada uno de los orígenes hasta él que por **Lema 1** es igual a SP_v^{opt}. Por último, recorre todos los nodos y toma el mínimo entre todos los SP_v^{opt} calculados lo cual es precisamente la solución que acabamos de definir.

2.5. Análisis de complejidad

El algoritmo tiene 3 partes importantes a considerar con respecto a su complejidad:

1. generar el grafo que modela la instancia del problema
2. aplicar BFS una vez por cada caballo con el nodo correspondiente al caballo como origen
3. buscar el nodo que cuente con la suma mínima

Generar el grafo que modela la instancia del problema recibida tiene costo $O(|V|) = O(n^2)$. Sólo precisamos una matriz de tamaño $n \times n$. Cada posición representa a un nodo y en él almacenaremos la suma de los caminos mínimos desde cada uno de los caballos hasta él. No es necesario guardar información acerca de las aristas del grafo porque, al ser el salto de del caballo un movimiento conocido, dado un nodo se pueden obtener sus vecinos en tiempo constante $O(1)$ calculándolos *ad hoc*. Tampoco es necesario guardar la distancia desde cada origen hasta cada uno de los nodos ya que sólo nos interesará la suma total sobre cada nodo. Por lo tanto, basta con ir sumando la distancia de cada uno de los orígenes hasta él.
El costo de aplicar BFS es $O(|E| + |V|) = O(n^2 * 8 + n^2) = O(n^2)$ [1, p. 597] porque, como ya vimos, la cantidad de aristas del grafo está acotada por la cantidad de nodos, porque el caballo a lo sumo puede saltar a 8 casillas diferentes.
Por último, buscar el mínimo tiene un costo $O(|V|) = O(n^2)$ porque simplemente consiste en iterar por todos los nodos y quedarse con el que tenga la suma mínima.
Por lo tanto, la complejidad está dada por:

$$O(construir\_grafo) + \#caballos * O(BFS) + O(obtener\_minimo) =$$

$$O(n^2) + k * O(n^2) + O(n^2) = O(k * n^2)$$

La cota obtenida $O(n^2 * k)$ es además válida para $\Theta(n^2 * k)$ porque lo único que puede variar de una instancia a otra del mismo tamaño es la posición de los caballos en el tablero. Sin embargo, la cantidad de nodos y aristas del grafo es independiente de la posición de los caballos en el tablero. Por lo tanto, el costo de construir el grafo, recorrerlo con BFS y luego recorrer todos los nodos para obtener el de suma mínima es el mismo para todas las instancias del mismo tamaño.

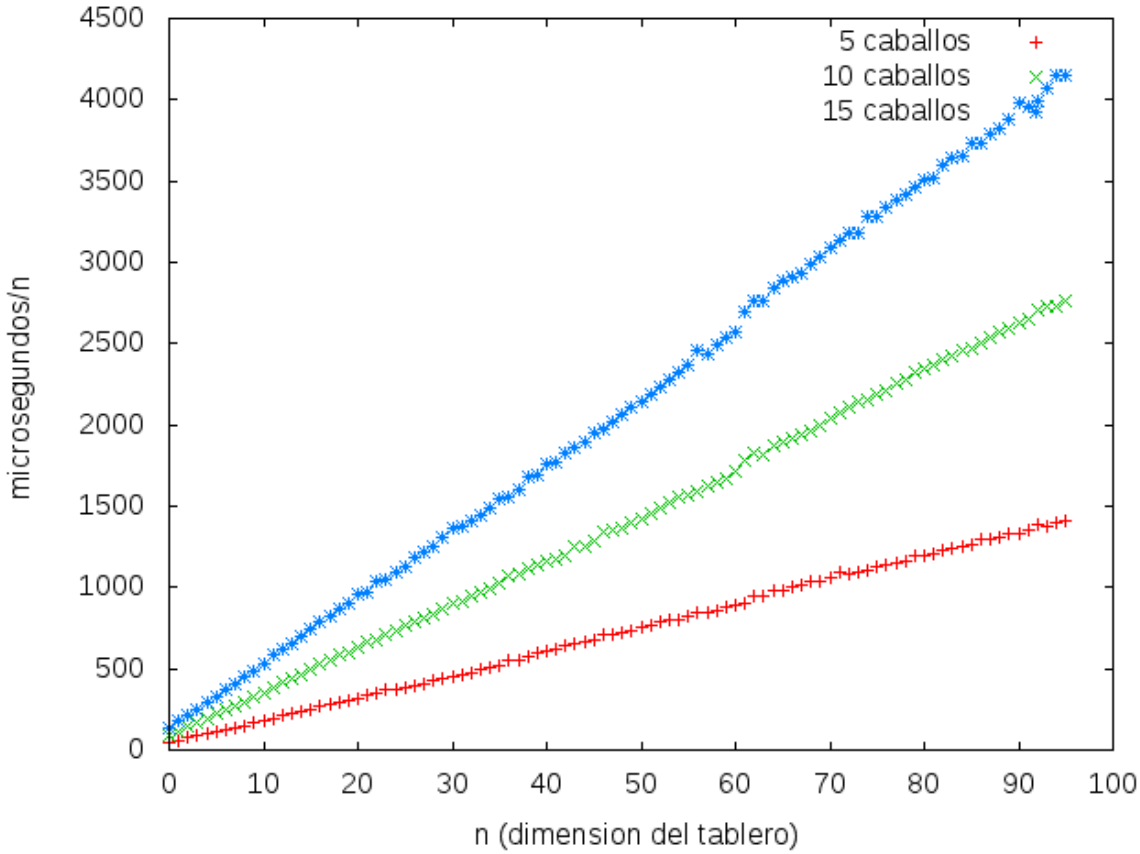
2.6. Tests de complejidad

El objetivo de las experimentaciones que realizaremos es corroborar que las cotas obtenidas por nuestro análisis de complejidad son reflejadas por los resultados empíricos. Como la cota depende de la cantidad de caballos utilizados y las dimensiones del tablero realizaremos dos experimentos:

1. Mantendremos la cantidad de caballos (k) constante e incrementaremos la dimensión del tablero (n) para ver si realmente la relación con el crecimiento de la dimensión del tablero es efectivamente cuadrática.

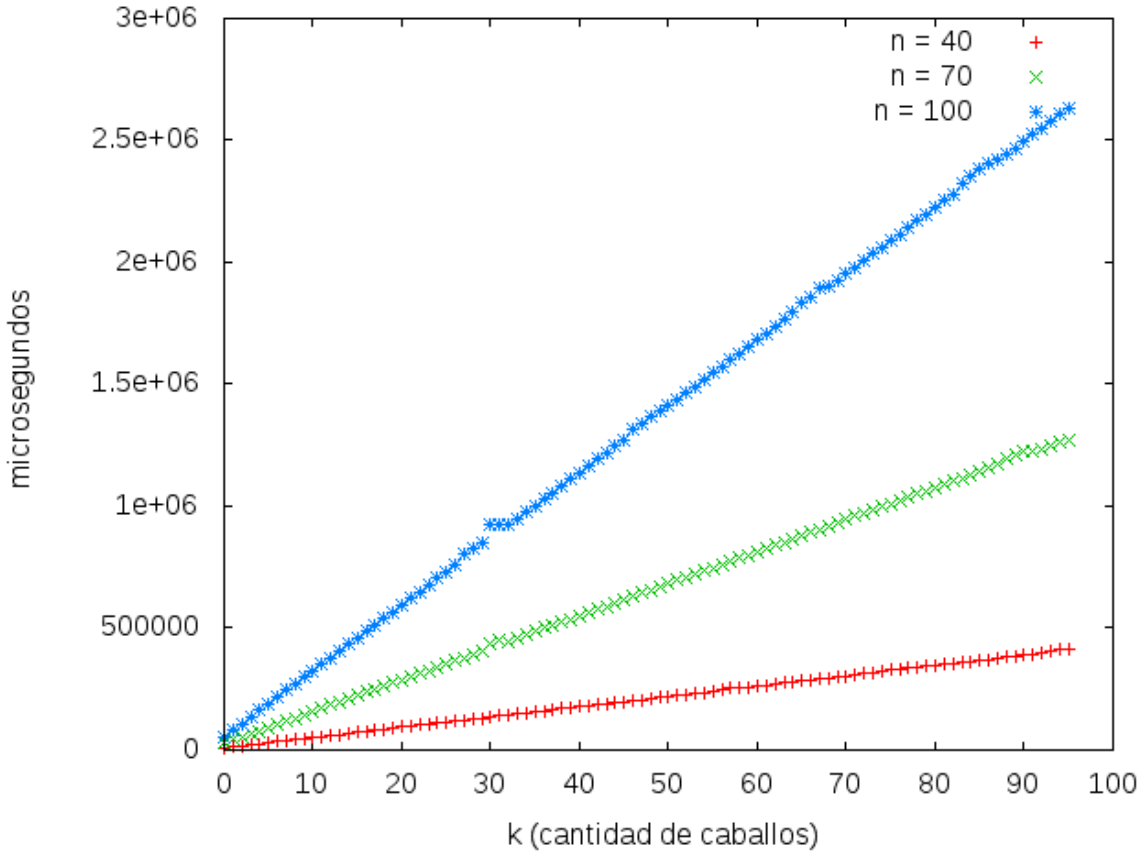
2. Mantendremos el la dimensión del tablero (n) constante e incrementaremos la cantidad de caballos para constatar si la relación con la cantidad de caballos es lineal como calculamos.

En el primer experimento variamos la dimensión del tablero desde 1 hasta 100 con un salto de 1. La posición de los caballos fue elegida pseudo-aleatoriamente, sin restringir que dos caballos tengan la misma posición original. Para cada dimensión n generamos 10 posicionamientos distintos de los caballos, cada posicionamiento lo ejecutamos 10 veces y nos quedamos con la corrida de menor tiempo de cada posicionamiento para finalmente tomar el promedio sobre los mínimos de los 10 posicionamientos distintos. Ésto lo realizamos para $k = 5, 10, 15$. El gráfico que muestra los resultados es el siguiente:



Como se ve en el gráfico la cantidad de microsegundos la dividimos por la dimensión del tablero (n) y las curvas resultantes para los 3 casos de caballos se acercan mucho a una recta.

El segundo experimento consistió en tomar un tablero de dimensión fija, en este caso en particular elegimos $n = 40, 70, 100$, y variar la cantidad de caballos desde 2 a 50. La posición de los caballos en este caso también fue elegida pseudo-aleatoriamente y para cada instancia de k caballos la corrimos con 10 posicionamientos distintos de los cuales tomamos el promedio y cada posicionamiento lo corrimos también 10 veces y nos quedamos con el mínimo. Presentamos los resultados en el siguiente gráfico:


3e+06
2.5e+06
2e+06
1.5e+06
1e+06
500000
0
microsegundos
0
10
20
30
40
50
60
70
80
90
100
k (cantidad de caballos)
n = 40
n = 70
n = 100

3. Problema 3: La comunidad del anillo

3.1. Presentación del problema

Dado un conjunto de computadoras, y un conjunto de enlaces entre pares de ellas (con a lo sumo un enlace para cada par), los cuales tienen un costo asociado, se pide elegir un conjunto de ellos tales que haya una red en forma "anillo", y que todas las computadoras tengan acceso a éste anillo usando enlaces de manera directa o indirecta (es decir, no es necesario que tengan un enlace a una computadora del anillo, sino que pueden tenerlo con una computadora intermedia), de manera de minimizar el costo total de los enlaces elegidos. Veamos por ejemplo las siguientes dos figuras:

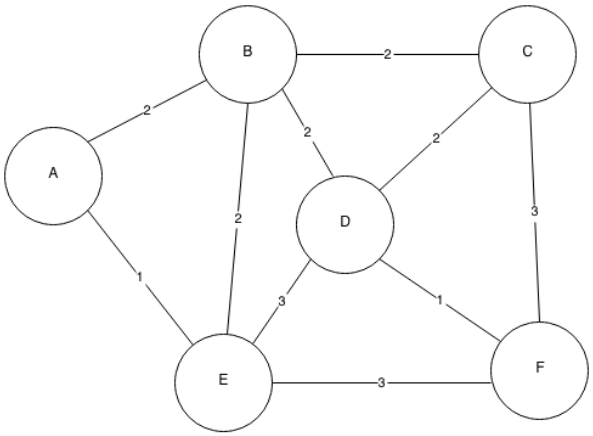


Figura 2: Red original

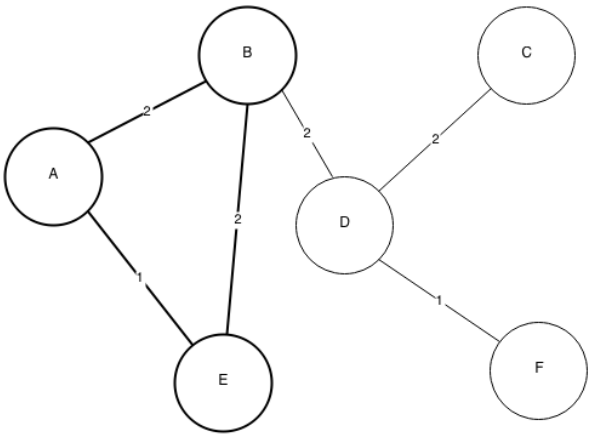


Figura 3: Una solución óptima

En la Figura 2 vemos una red de seis computadoras con sus enlaces y sus costos asociados, y en la Figura 3 tenemos una solución óptima del problema, con el anillo de equipos remarcado.

3.2. Resolución

Primero necesitamos caracterizar al conjunto de soluciones del problema. Como a lo sumo hay un enlace entre dos máquinas, podemos pensar a éstas como vértices, y a los enlaces como aristas. Entonces, tenemos como input un grafo $G = (V, X)$, y una solución (si existe) será un subgrafo de G con las siguientes características:

1. Debe ser **generador**, ya que para toda computadora se cumple que: o forma parte del anillo, o existe un camino simple que lleva a alguna máquina del anillo, y para ésto todas las computadoras deben estar en el subgrafo solución.

2. Debe ser **conexo**, porque si no lo fuera, eso implica que hay alguna máquina que no puede conectarse con el anillo.

3. Debe tener un **circuito simple**, éste será el anillo. Si no puede contruirse, no habrá solución.

4. Más aún, debe tener **exactamente un circuito simple**. El problema no prohíbe que haya más de uno, pero si tuviéramos más de un ciclo, bastaría con quitar la arista más costosa que forme parte de algún circuito para mejorar la solución, ya que el subgrafo

seguiría siendo conexo (los caminos simples que pasaran por la arista que quitamos podrían desviarse por el resto del circuito) y seguiría existiendo al menos un circuito simple (si no, este subgrafo sería un árbol, pero entonces agregar la arista que quitamos nos daría un grafo con exactamente un circuito simple, absurdo). Este procedimiento lo podríamos repetir hasta que sólo quede un circuito. Luego, podemos descartar todos los subgrafos con más de un circuito simple, ya que son soluciones necesariamente subóptimas.

Tenemos entonces caracterizada a una solución como un subgrafo generador de G conexo y con exactamente un circuito simple. Una solución óptima entonces será minimizar el costo de este tipo de subgrafos. Vamos a construir una solución óptima en dos pasos:

1. Hallaremos $T = (V, X_T)$ un árbol generador mínimo de G (si no existe, no hay solución).
2. Tomaremos la arista e de mínimo costo en $X - X_T$ (si el conjunto es vacío, no hay solución) y la agregaremos a T.

Nuestro algoritmo construirá entonces el subgrafo $S = (V, X_T \cup \{e\})$. En la demostración de correctitud veremos que efectivamente S es una solución óptima del problema.

Para obtener el árbol generador mínimo usamos el algoritmo de Prim, luego a este grafo le agregaremos la arista más chica de las que no pertenecen a él, y finalmente para encontrar el ciclo usaremos la función DFS_visit, que hace un Depth-First Search sobre la solución para encontrar al único circuito simple.

La función DFS_visit es la misma que la presentada en el libro de Cormen *Introduction to Algorithms*[1, p. 604] añadiéndole la funcionalidad de distinguir un *back edge* (esto es, una arista que efectivamente cierra un ciclo en el grafo). En el libro, se llama primero a la función DFS, que inicializa todos los vértices con color blanco (esto significa que no fueron visitados), y luego llama a DFS_visit para cada uno de los vértices del grafo que tengan ese color. Como nuestro grafo solución es conexo, alcanza llamar a DFS_visit con un sólo vértice para que DFS recorra todo el grafo. En particular, la llamaremos con un vértice que sabemos que está en el ciclo, esto es, con uno de los vértices en los que incide la arista que agregamos al final. Así, al finalizar DFS_visit, como el vértice tendrá necesariamente un *back edge*, lo usaremos para reconstruir el único ciclo (el *back edge* es una arista del ciclo, y luego alcanza con pedir el anterior del otro vértice hasta que lleguemos al mismo con el que llamamos a DFS_visit originalmente).

La información que nos da DFS_visit la guardamos en un array de la estructura infoVerticeDFS, donde la posición i-ésima corresponde al vértice i-ésimo. Este array inicializa todos los vértices en el color blanco. Luego, cada vez que visitamos un vértice, es decir, cada vez que se ejecuta DFS_visit, el vértice se pone de color gris, hasta que se terminen de considerar sus vértices adyacentes. Una vez hecho esto, se pone de color negro. Entonces, si al visitar un vértice ocurre que entre sus adyacentes hay uno coloreado gris, y no es el anterior al actual (de lo contrario sería como volver por el mismo camino, lo cual no es un circuito simple), esto implica que la arista que conecta con ese vértice es un *back edge* (porque como está gris, todavía estábamos recorriendo caminos que salían de ese vértice, y volvimos a él), lo cual guardaremos en la información del vértice ya visitado. Por lo tanto, si la primera vez que llamamos a DFS_visit lo hacemos con un vértice del circuito, necesariamente tendremos su único *back edge* al finalizar.

3.2.1. Pseudocódigo

Algorithm 6 LaComunidadDelAnillo(aristasGrafo)

```
1: Conjunto(Vertice) verticesAGM ← vacía
2: Conjunto(Arista) aristasAGM ← vacía
3: Conjunto(Arista) aristasCandidatasAGM ← vacía
4: Pongo el vértice 0 en verticesAGM, y sus aristas en aristasCandidatasAGM
5: while |aristasAGM| < n - 1 and |aristasCandidatasAGM| > 0 do
6:    Tomar la arista candidata de menor costo (quitándola además de las candidatas)
7:    if La arista incide en dos vértices del AGM then
8:       No la puedo usar
9:    else
10:      Insertar el nuevo vértice en verticesAGM
11:      Eliminar la arista de aristasGrafo
12:      Insertar la arista en aristasAGM
13:      for each arista del nuevo vértice (que no fue considerada antes) do
14:         Insertar arista en aristasCandidatasAGM
15:      end for
16:   end if
17: end while
18: if |aristasAGM| < n - 1 or |aristasGrafo| == 0 then
19:    No hay solución
20: end if
21: Arista aristaMenor ← Arista de menor costo de aristasGrafo
22: Vertice primero ← El primer vértice de aristaMenor (podría ser cualquiera de los dos
    vértices, no afecta al algoritmo)
23: Insertar aristaMenor en aristasAGM
24: infoVerticeDFS info[n] // Ésto tiene la información del recorrido DFS, se inicializan los
    vértices en BLANCO
25: Llamar a la función DFS_visit(info, primero) sobre el subgrafo solución
26: Arista backEdge ← info[primero].backEdges.front()
27: Lista(Arista) aristasAnillo ← vacía
28: Insertar backEdge en aristasAnillo
29: Eliminar backEdge de aristasAGM
30: Vertice actual ← backEdge.dameElOtroVertice(primero)
31: while actual ≠ primero do
32:    Eliminar info[actual].aristaAnterior de aristasAGM
33:    Insertar info[actual].aristaAnterior en aristasAnillo
34:    actual ← info[actual].verticeAnterior
35: end while
```

Al finalizar el algoritmo, las aristas del circuito de la solución óptima quedan en *aristasAnillo*, y el resto en *aristasAGM*.

Algorithm 7 DFS_visit(infoVerticeDFS * info, Vertice actual)

```
1: info[actual].estado ← GRIS
2: for each arista del vértice actual do
3:    Vertice nuevoActual ← arista.dameElOtroVertice(actual)
4:    if info[nuevoActual].estado == BLANCO then
5:       info[nuevoActual].aristaAnterior ← arista
6:       info[nuevoActual].verticeAnterior ← actual
7:       DFS_visit(info, nuevoActual)
8:    else if info[nuevoActual].estado == GRIS then
9:       if nuevoActual == info[actual].verticeAnterior then
10:         // Si estoy acá es porque estaba volviendo por la misma arista que vine (formando
            un circuito NO simple)
11:      else
12:         // El nuevo vertice ya habia sido visitado, entonces hay back edge
13:         Insertar arista en info[nuevoActual].backEdges
14:      end if
15:   end if
16: end for
17: info[actual].estado ← NEGRO
```

3.3. Demostración de correctitud

Nuestro algoritmo encuentra un árbol generador mínimo del grafo de la red de computadoras usando el algoritmo de Prim, y le añade la arista (conexión) más barata de las que quedaron. Afirmamos que ésa es una solución óptima para el problema de hallar un subgrafo generador conexo con exactamente un circuito simple.

Veamos entonces que nuestro algoritmo devuelve una solución óptima. Demostraremos ésto por el absurdo, suponiendo que hay una solución estrictamente mejor que la nuestra. Pero primero, necesitamos demostrar el siguiente lema.

Lema 2. *Sea $G = (V, X)$ un grafo conexo con exactamente un circuito simple C. Sea e una arista tal que $e \in C$. Sea $X' = X - \{e\}$. Entonces $G' = (V, X')$ es un árbol generador de G.*

Demostración. Supongamos que $G' = (V, X')$ no es un árbol generador de G. Esto es, G' no es generador de G, o no es árbol.
Si no fuera generador, entonces no tendría los mismos nodos que G, lo cual es absurdo porque $G = (V, X)$ y $G' = (V, X')$. Entonces G' no debe ser árbol. Esto es, o G' no es conexo, o tiene al menos un circuito simple.
Si G' no fuera conexo, entonces existen vértices v, w en V tales que no existe un camino simple entre ellos. Sabemos que G es conexo, entonces sea $C_{v,w}$ el camino simple entre v y w partiendo desde v. Hay dos posibilidades: $e \in C_{v,w}$ ó $e \notin C_{v,w}$.

- Si $e \notin C_{v,w}$, entonces $C_{v,w} \subseteq X'$ y por lo tanto el camino está en G'.
- Si $e \in C_{v,w}$, entonces $C_{v,w} \not\subseteq X'$, pero como $e \in C$, es decir, e está en el circuito simple de G, si $e = (a, b)$ donde $a, b \in V$, entonces existen dos caminos entre a y b, a saber: ir por e ó ir por $C - \{e\}$. Podemos escribir $C_{v,w} = C_{v,a} \oplus (a, b) \oplus C_{b,w}$ donde $C_{v,a}$ y $C_{b,w}$ son los subcaminos entre v,a y b,w respectivamente, y además $e \notin C_{v,a}$, $e \notin C_{b,w}$ porque $C_{v,w}$ es camino simple. El camino $C'_{v,w} = C_{v,a} \oplus (C - \{e\}) \oplus C_{b,w}$ es un camino

entre v y w tal que $C'_{v,w} \subseteq X'$, y por lo tanto hay un camino simple entre v y w en G' (si $C'_{v,w}$ no fuera simple, siempre podemos quitar los circuitos que se generan al pasar por un mismo nodo, quedando de esa forma un camino simple).

Luego, G' es conexo. Entonces G' debe tener al menos un circuito simple C', y $e \notin C'$ porque $e \notin X'$. Pero esto implica que G tiene dos circuitos simples distintos porque $C' \subseteq X' \subseteq X$, y $C' \neq C$ pues $e \in C$, lo cual es absurdo porque por hipótesis G tiene exactamente un circuito simple. El absurdo provino de suponer que G' no es árbol generador de G.
Por lo tanto, G' es un árbol generador de G. □

Notación. *Dado un grafo cualquiera $G = (V, X)$ con $x \in X$,*

$$G - \{x\} = (V, X - \{x\})$$

Correctitud del algoritmo. *Sea $G = (V, X_G)$ el grafo de la red de computadoras, donde cada nodo es una computadora, y cada arista es una conexión. Supongamos que existe solución al problema de encontrar un subgrafo generador conexo de G con exactamente un circuito simple. Sea $S = (V, X)$ subgrafo generador de G, la solución construida por nuestro algoritmo, esto es: $T = (V, X_T)$ un árbol generador mínimo de G, más agregar la arista $e \in \{x \in X_G - X_T \mid l(x) \leq l(x') \; \forall x' \in X_G - X_T\}$, esto es, $X = X_T \cup \{e\}$. S es un subgrafo generador conexo con exactamente un circuito simple C tal que $e \in C$, por definición equivalente de árbol. Entonces, no existe $S' = (V, X')$ subgrafo generador de G conexo con exactamente un circuito simple, tal que $costo(S) > costo(S')$, siendo para cualquier grafo $G = (V, X)$*

$$costo(G) = \sum_{x \in X} l(x)$$

Demostración. Supongamos que hay una solución mejor S', esto es, $costo(S) > costo(S')$.
Sea e' una arista cualquiera del circuito de S'. Entonces,

$$costo(S - \{e\}) \leq costo(S' - \{e'\})$$

Esto es así porque $S - \{e\}$ es árbol generador mínimo y S' sin una arista de su circuito es un árbol generador por el Lema 2.
Entonces, para cada arista e' del circuito de S', debe ser $l(e) > l(e')$, de lo contrario sería $l(e) \leq l(e')$ y entonces tendríamos

$$costo(S) = costo(S - \{e\}) + l(e) \leq costo(S' - \{e'\}) + l(e') = costo(S')$$

lo cual no puede ocurrir por hipótesis. Además, cada arista del circuito de S' debe estar en $S - \{e\}$, porque si no, el algoritmo no hubiera elegido a e como arista final, habiendo una de menor costo.
Por lo tanto, el circuito de S' está en $S - \{e\}$. Pero entonces $S - \{e\}$ tiene un circuito siendo un árbol, absurdo. El absurdo provino de suponer que hay una solución mejor que la construida por nuestro algoritmo.
Luego, la solución devuelta por nuestro algoritmo es óptima, y el algoritmo es correcto. □

3.4. Análisis de complejidad

```
1 std::vector< list<Arista> > aristasDeCadaVertice;
2 std::vector< list<Arista> > aristasDeCadaVerticeAGM;
3 std::set<Arista, comparacionArista> aristasGrafo;
4 std::set<Vertice> verticesAGM;
5 std::set<Arista, comparacionArista> aristasAGM;
6 std::set<Arista, comparacionArista> aristasCandidatasAGM;
7 std::list<Arista> aristasAnillo;
```

Empecemos analizando los contenedores que usamos, ya que son clave para el cálculo de complejidad de peor caso. El grafo se define en **aristasDeCadaVertice**, que es un vector de listas de aristas. El i-ésimo elemento corresponde al i-ésimo vértice, y su lista son las aristas que inciden en él. En **aristasDeCadaVerticeAGM** se construirá la solución. En **aristasGrafo**, que es un conjunto ordenado de aristas (compara usando *comparacionArista*, que ordena cada arista pesada (v,w,c) –con v,w vértices y c su peso–, primero por costo, y si son iguales, ordena por vértices, teniendo en cuenta el caso en que $e = (v, w, c)$ y $e' = (w, v, c)$, ya que $e = e'$), se hallarán las aristas originales del grafo input, y se usarán los conjuntos ordenados **aristasAGM** para guardar las aristas de la solución y **aristasCandidatasAGM** para guardar las aristas candidatas que procesará el algoritmo de Prim. De la misma manera, el conjunto ordenado **verticesAGM** será usado por Prim para mantener el registro de los vértices que están en el árbol generador parcial. Por último, **aristasAnillo** es una lista de aristas que usaremos al final para guardar las aristas del único circuito simple de la solución.

Cuando comienza el algoritmo, se completan **aristasDeCadaVerticeAGM** y **aristasGrafo**, lo cual en el peor caso ($m = \frac{n(n-1)}{2}$, esto es, un grafo completo) cuesta

$$
\begin{aligned}
T(n) &= O(m) + O\left(\sum_{i=1}^{m} \log i\right) \\
T(n) &= O(m) + O(\log m!) \\
T(n) &= O(m) + O(m \log m) \\
T(n) &= O(n^2 \log n^2) \\
T(n) &= O(n^2 2 \log n) \\
T(n) &= O(n^2 \log n)
\end{aligned}
$$

pues $\log k! = k \log(k) - k + O(\log k)$ por la aproximación de Stirling del factorial.

Se empieza poniendo el *vértice 0* en **verticesAGM** y sus aristas en **aristasCandidatasAGM**. Insertar en **std::set** cuesta $\log |conjunto|$, entonces en el peor caso hasta ahora tenemos

$$
\begin{aligned}
T(n) &= O(n^2 \log n) + O(\log 1) + O\left(\sum_{i=1}^{n-1} \log i\right) \\
T(n) &= O(n^2 \log n) + O(\log n!) \\
T(n) &= O(n^2 \log n) + O(n \log n) \\
T(n) &= O(n^2 \log n)
\end{aligned}
$$

El algoritmo de Prim es un ciclo que termina cuando ya no hay más aristas candidatas, o cuando ya tenemos $n - 1$ en la solución. El algoritmo va a considerar todas las aristas,

lo que implica que van a hacerse m inserciones, y a lo sumo $n-1$ eliminaciones de aristas en **aristasCandidatasAGM**. En cada iteración se toma la arista candidata de menor costo, lo cual cuesta $O(1)$, se la elimina en $O(\log |aristasCandidatasAGM|) = O(\log m)$ de **aristasCandidatasAGM**, se chequea si incide en dos vértices del árbol parcial en $O(\log |verticesAGM|)$, y si no incide, se inserta el vértice nuevo en **verticesAGM** lo cual cuesta $O(\log |verticesAGM|)$, se elimina la arista de **aristasGrafo** en $O(\log |aristasGrafo|)$, se inserta la misma en **aristasAGM** en $O(\log |aristasAGM|)$ y en las listas de **aristasDeCadaVerticeAGM** de sus vértices incidentes en $O(1)$; luego, para cada una de las aristas del vértice nuevo se chequea si inciden en dos vértices (ya que si lo hacen, estaría poniendo como candidata una arista que ya consideré) en $O(\log |verticesAGM|)$, y en caso contrario la agrego en **aristasCandidatasAGM** en $O(\log |aristasCandidatasAGM|)$.

Entonces, podemos afirmar que:

- Por cada arista va a llamarse a lo sumo tres veces a la función $incideEnDosVertices$: primero al intentar agregarla a las candidatas (lo cual pasa dos veces porque dos vértices la tienen en su lista), y después cuando efectivamente se considera añadirla al árbol. El costo total lo podemos acotar por $O(m \log n)$ que es $O(n^2 \log n)$ en el peor caso.
- Se insertan a lo sumo n vértices en **verticesAGM**, lo cual en total cuesta $O(n \log n)$.
- En **aristasCandidatasAGM**, se insertan m aristas y se eliminan a lo sumo $n-1$. Esto lo podemos acotar por $O(\log m!) + O((n-1) \log m) = O(m \log m) + O(n \log m)$, que en el peor caso es $O(n^2 \log n^2) + O(n \log n^2) = O(n^2 \log n)$.
- Se eliminan a lo sumo $n-1$ aristas de **aristasGrafo**, lo cual en total podemos acotar por $O(n \log m) = O(n \log n^2) = O(n \log n)$.
- Se insertan a lo sumo $n-1$ aristas en **aristasAGM**, lo cual podemos acotar por $O(\log n!) = O(n \log n)$.
- En **aristasDeCadaVerticeAGM**, se hacen a lo sumo $2(n-1)$ inserciones, lo cual en total cuesta $O(n)$.

Por lo tanto, el algoritmo de Prim cuesta en el peor caso $O(n^2 \log n)$. Veamos qué ocurre con el resto del algoritmo.

Ya tenemos armado el árbol generador mínimo. Lo que hacemos ahora es pedir en $O(1)$ la arista de menor costo en **aristasGrafo**, ya que ahí quedaron todas las aristas que no fueron insertadas en el AGM, y la agregamos en **aristasAGM** en $O(\log n)$. Calcular el precio total de la solución es simplemente iterar con un acumulador sobre **aristasAGM**, lo cual cuesta $O(n)$.

Sólo queda identificar el *anillo*, es decir, el único circuito simple de la solución. Para ello, se llama a *DFS_visit* con uno de los vértices de la arista añadida al final. Esta función hace un recorrido DFS sobre el subgrafo solución e identifica el *back edge*, y como visita a cada vértice una sóla vez, considerando para cada uno sus aristas (y añadir el back edge cuesta $O(1)$) el costo total de ejecutar DFS es $O(n+m)$[1, p. 606], y como $m = n$, es $O(n+n) = O(n)$. El costo de diferenciar al ciclo de las demás aristas de la solución es entonces insertar las aristas del circuito en la lista **aristasAnillo**, lo cual cuesta $O(n)$ en el peor caso, y de eliminar estas aristas de **aristasAGM** lo cual cuesta $O(\log n!) = O(n \log n)$ en el peor caso. En total, todo este paso lleva $O(n \log n)$.

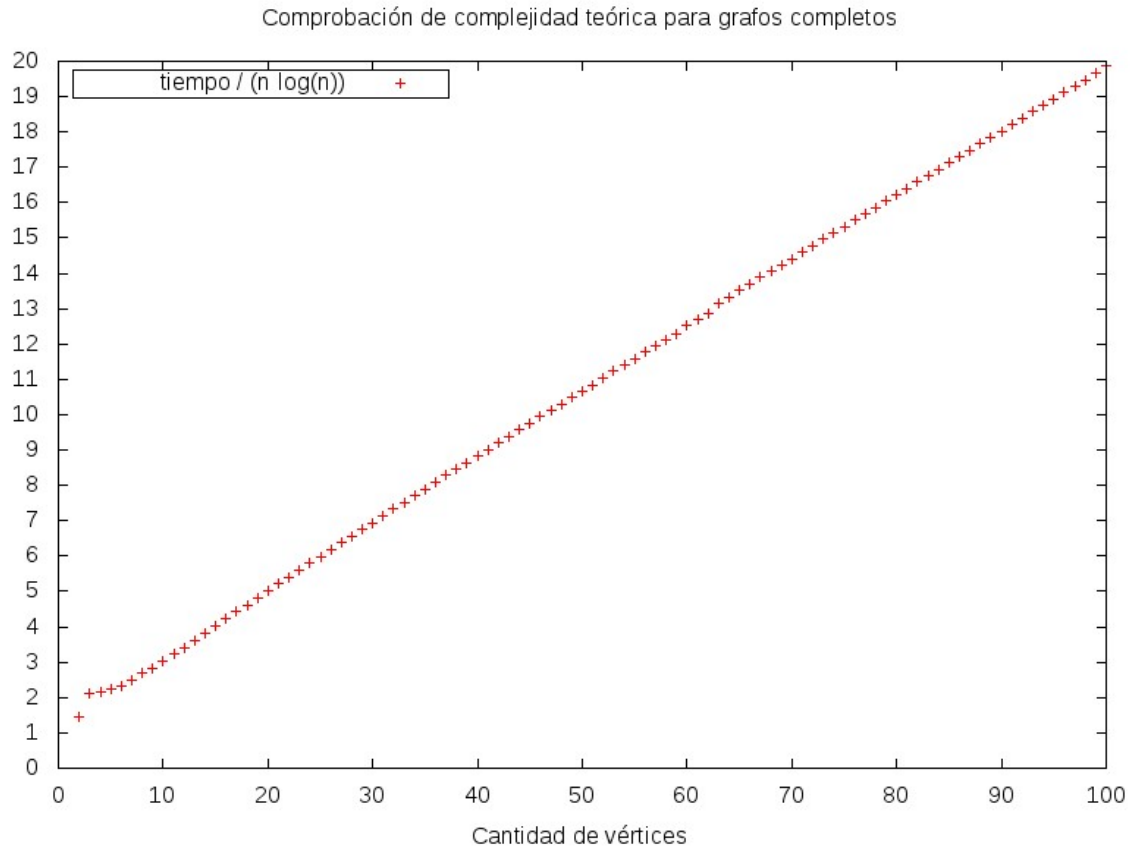
Luego, la complejidad temporal de peor caso del algoritmo es

$$T(n) = O(n^2 \log n) + O(n^2 \log n) + O(n) + O(n \log n)$$
$$T(n) = O(n^2 \log n)$$

lo cual es estrictamente mejor que $O(n^3)$.

3.5. Tests de complejidad

Para testear la complejidad de peor caso, construimos 1000 instancias aleatorias de grafos completos de n vértices, para cada $n = 1, ..., 100$ y calculamos el tiempo de ejecución 10 veces para cada instancia, tomando el mínimo. Luego, promediamos los tiempos de las 1000 instancias para cada n. Finalmente, como afirmamos que la complejidad es $O(n^2 \log n)$, dividimos la muestra por $n \log n$ y esperamos ver una recta, lo cual efectivamente ocurre, como se aprecia a continuación:



Testeamos también para grafos aleatorios, esto es, para cada n cantidad de vértices, hicimos aleatoria la cantidad de aristas (de 0 a $\frac{n(n-1)}{2}$) y las seleccionamos también aleatoriamente (para ello, seleccionamos aleatoriamente un vértice que todavía tenga aristas disponibles, y elegimos una random). Usamos de nuevo 1000 instancias por cada n, calculando cada una diez veces y tomando el mínimo, y luego promediando estos. Como elegimos la cantidad de aristas usando $rand()\,\%(1 + n * (n-1)/2)$, entonces si definimos el espacio muestral $S = \left\{0, 1, ..., \frac{n(n-1)}{2}\right\}$ que representa la cantidad de aristas posibles, sabemos

que vale $P(\{x\}) \approx \frac{1}{p}\ \forall x \in S$ donde $p = 1 + \frac{n(n-1)}{2}$. Sea la variable aleatoria $X\colon S \to \mathbb{R}$, $X(x) = x\ \forall x \in S$, entonces por todo lo anterior el valor esperado de X es

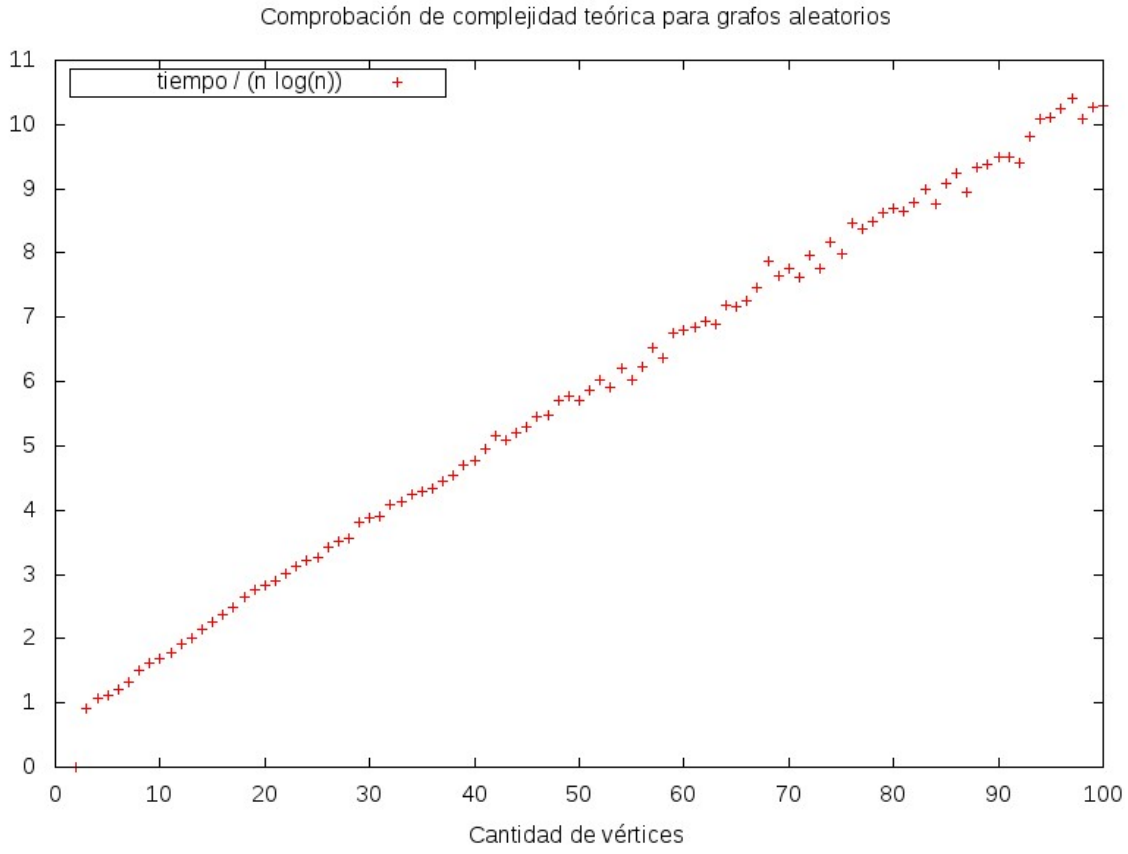
$$E(X) = \sum_{x \in R_x} x\ P(X = x) = \sum_{x \in R_x} x\ P(\{x\})$$

$$E(X) = \sum_{x \in R_x} x\ \frac{1}{p} = \frac{1}{p} \sum_{i=0}^{p-1} i$$

$$E(X) = \frac{(p-1)p}{2p} = \frac{p-1}{2}$$

$$E(X) = \frac{n(n-1)}{4}$$

por lo que el valor esperado de aristas para n vértices es $\frac{n(n-1)}{4} = O(n^2)$ y entonces valen las mismas conclusiones sacadas en la sección de análisis de complejidad, es decir, esperamos que la complejidad sea también $O(n^2 \log n)$. Como antes, dividimos la muestra por $n \log n$:



4. Apéndice: Códigos fuente

4.1. Problema 1: Plan de vuelo

```
public static String mejorVuelo(Aeropuerto[] aeropuertos) {
  if (existeVuelo(aeropuertos, aeropuertos[0], aeropuertos[1], -2)) {
    Vuelo min = aeropuertos[1].primeroEnLlegar();
    StringBuilder builder = new StringBuilder();
    builder.insert(0, "␣" + min.id());
    String llegada = min.llegada() + "␣";
    int k = 1;
    while (!min.origen().equals(aeropuertos[0])) {
      min = min.origen().primeroEnLlegar();
      builder.insert(0, "␣" + min.id());
      k++;
    }
    return llegada + k + builder.toString();
  }
  return "no";
}

public static boolean existeVuelo(Aeropuerto[] aeropuertos,
    Aeropuerto inicio, Aeropuerto destino, int t) {
  if (inicio.equals(destino)) {
    return true;
  }
  if (t + 2 <= inicio.obtenerUltimoVueloQueLlega()) {
    return true;
  }
  boolean llego = false;
  List<Vuelo> vuelos = inicio.vuelosQueSalen();
  List<Vuelo> vuelosNoAnalizados = new LinkedList<Vuelo>();
  inicio.vaciarVuelosQueSalen();
  for (Vuelo vuelo : vuelos) {
    if (vuelo.partida() >= t + 2) {
      if (vuelo.color().equals(Color.BLANCO)) {
        if (existeVuelo(aeropuertos, vuelo.destino(), destino,
            vuelo.llegada())) {
          vuelo.cambiarColor(Color.VERDE);
          if (vuelo.destino().primeroEnLlegar() == null
              || vuelo.destino().primeroEnLlegar().llegada() > vuelo
                  .llegada()) {
            vuelo.destino().agregarPrimeroEnLlegar(vuelo);
          }
          if (inicio.obtenerUltimoVueloQueLlega() < vuelo
              .llegada()) {
            inicio.cambiarUltimoVueloQueLlega(vuelo);
          }
          llego = true;
        } else {
          vuelo.cambiarColor(Color.ROJO);
        }
      }
    } else {
      vuelosNoAnalizados.add(vuelo);
    }
  }
```

```
53    inicio.asignarVuelosQueLlegan(vuelosNoAnalizados);
54    return llego;
55  }
```

4.2. Problema 2: Caballos salvajes

4.2.1. Código del algoritmo que resuelve el problema

```
1   #define NOT_VISITED -1
2
3   struct Square {
4     Square() : row(0), column(0) {};
5     Square(int r, int c) : row(r), column(c) {};
6
7     int row;
8     int column;
9   };
10
11  struct Board {
12
13    vector<vector<int> > sums;
14    vector<vector<vector<int> > > squares;
15
16    Board(int n, int horses) :
17      squares(vector<vector<vector<int> > > (n, vector<vector<int> >
            (n, vector<int> (horses, NOT_VISITED) ) ) ),
18        sums(vector<vector<int> > (n, vector<int>(n, 0)))
19    {};
20
21    bool is_marked(Square s, int horse)
22    {
23      if (squares[s.row][s.column][horse] == NOT_VISITED) {
24        return false;
25      }
26
27      return true;
28    };
29
30    int get_distance(Square s, int horse)
31    {
32      int dist = squares[s.row][s.column][horse];
33      return dist;
34    };
35
36    void set_distance(Square s, int horse, int d)
37    {
38      squares[s.row][s.column][horse] = d;
39      sums[s.row][s.column] += d;
40    };
41
42    void mostrar_sums()
43    {
44      cout << "SUMS" << endl;
45      for (int i = 0; i < sums.size(); i++) {
46        for (int j = 0; j < sums.size(); j++) {
```

```
47          cout << sums[i][j] << ' ';
48        }
49        cout << endl << endl;
50      }
51    };
52
53    vector<int> get_min_sum()
54    {
55      int r, c;
56      int min = std::numeric_limits<int>::max();
57      for (int i = 0; i < sums.size(); i++) {
58        for (int j = 0; j < sums.size(); j++) {
59          if (min > sums[i][j]) {
60            min = sums[i][j];
61            r = i;
62            c = j;
63          }
64        }
65      }
66
67      vector<int> result;
68      result.push_back(r);
69      result.push_back(c);
70      result.push_back(min);
71
72      return result;
73    };
74  };
75
76  int movements[][2] = {
77    {-2, -1}, {-2, 1}, {2, -1}, {2, 1},
78    {1, 2}, {1, -2}, {-1, 2}, {-1, -2}
79  };
80
81  queue<Square> get_neighbours(Square s, int n)
82  {
83    queue<Square> neighbours;
84    int r, c;
85    for (int i = 0; i < 8; i++) {
86      r = s.row + movements[i][0];
87      c = s.column + movements[i][1];
88      if (r >= 0 && c >= 0 && r < n && c < n) {
89        neighbours.push(Square(r, c));
90      }
91    }
92
93    return neighbours;
94  }
95
96  void bfs(int horse, Square origin, Board &board)
97  {
98    queue<Square> nodes_left;
99    nodes_left.push(origin);
100   board.set_distance(origin, horse, 0);
101
102   queue<Square> neighbours;
103   Square node;
104   int node_distance;
```

```
105   while (!nodes_left.empty()) {
106     node = nodes_left.front();
107     node_distance = board.get_distance(node, horse);
108     nodes_left.pop();
109
110     neighbours = get_neighbours(node, board.squares.size());
111
112     while (!neighbours.empty()) {
113       Square neighbour = neighbours.front();
114       neighbours.pop();
115       if (!board.is_marked(neighbour, horse)) {
116         board.set_distance(neighbour, horse, node_distance+1);
117         nodes_left.push(neighbour);
118       }
119     }
120   }
121 }
122
123 int main() {
124  int n, k;
125
126  cin >> n >> k;
127  Board board(n, k);
128  int r, c;
129  for (int i = 0; i < k; i++) {
130    cin >> r >> c;
131    bfs(i, Square(r, c), board);
132  }
133
134  vector<int> result = board.get_min_sum();
135
136  cout << result[0] << ' ' << result[1] << ' ' << result[2] << endl;
137
138  return 0;
139 }
```

4.2.2. Código del generador de instancias

```
1  int main() {
2    // test configuration
3    const int instances = 200;
4    const int start = 4;
5    const bool vary_n = false;
6    const int constant_horses = 5;
7    const int constant_n = 100;
8    srand(time(NULL));
9
10   std::ofstream ofs;
11   std::string test_file;
12   if (vary_n) {
13     test_file = "tests_n.in";
14   } else {
15     test_file = "tests_k.in";
16   }
17   ofs.open(test_file.c_str(), std::ofstream::out);
18
```

```
19    ofs << instances << std::endl;
20
21    for (int i = 2; i < instances; i++) {
22      if (vary_n) {
23        for (int l = 0; l < 3; l++) {
24          for (int m = 0; m < 10; m++) {
25            ofs << i << ' ' << constant_horses+5*l << std::endl;
26            for (int j = 0; j < constant_horses+5*l; j++) {
27              ofs << rand() % i << ' ' << rand() % i << std::endl;
28            }
29          }
30        }
31      } else {
32        for (int l = 0; l < 3; l++) {
33          for (int m = 0; m < 10; m++) {
34            ofs << 40+l*30 << ' ' << i << std::endl;
35            for (int j = 0; j < i; j++) {
36              ofs << rand() % 40+l*30 << ' ' << rand() % 40+l*30 <<
                      std::endl;
37            }
38          }
39        }
40      }
41    }
42
43    ofs.close();
44
45    return 0;
46  }
```

4.2.3. Código del tester

```
1   int main() {
2    // configuration
3    const bool test_n = false;
4
5    std::string in_file;
6    std::string out_file;
7    if (test_n) {
8      in_file = "tests_n.in";
9      out_file = "tests_n.out";
10   } else {
11     in_file = "tests_k.in";
12     out_file = "tests_k.out";
13   }
14
15   std::ifstream ifs;
16   ifs.open(in_file.c_str(), std::ifstream::in);
17   std::ofstream ofs;
18   ofs.open(out_file.c_str(), std::ofstream::out);
19
20   int instances;
21
22   ifs >> instances;
23
24   for (int i = 4; i < instances; i++) {
```

```
25      std::cout << "PROCESSING␣INSTANCE␣" << i << endl;
26      int n, k;
27      ifs >> n >> k;
28      std::cout << "n:␣" << n << "␣k:␣" << k << endl;
29
30      vector<vector<int> > horses (k, vector<int>(2, -1));
31      for (int h = 0; h < k; h++) {
32        ifs >> horses[h][0] >> horses[h][1];
33      }
34
35      long long int minimum = std::chrono::microseconds::max().count();
36      for (int j = 0; j < 2; j++) {
37        std::chrono::high_resolution_clock::time_point t1 =
              std::chrono::high_resolution_clock::now();
38
39        Board board(n, k);
40
41        for (int l = 0; l < k; l++) {
42          bfs(l, Square(horses[l][0], horses[l][1]), board);
43        }
44
45
46        vector<int> result = board.get_min_sum();
47
48        std::chrono::high_resolution_clock::time_point t2 =
              std::chrono::high_resolution_clock::now();
49
50        long long int current =
              std::chrono::duration_cast<std::chrono::microseconds>(t2 -
              t1).count();
51        if (current < minimum) {
52          minimum = current;
53        }
54      }
55
56      if (test_n) {
57        ofs << n << '␣' << minimum/n << endl;
58      } else {
59        ofs << k << '␣' << minimum << endl;
60      }
61    }
62
63    return 0;
64  }
```

4.3. Problema 3: La comunidad del anillo

4.3.1. Código del algoritmo que resuelve el problema

```
1  typedef int Vertice;
2
3  class Arista {
4    private:
5      Vertice v1;
6      Vertice v2;
```

```
7       int l;
8     public:
9       Arista() : v1(-1), v2(-1), l(-1){}
10      Arista(const Vertice & v1, const Vertice & v2, const int & l) :
            v1(v1), v2(v2), l(l){}
11      int costo() const {
12        return l;
13      }
14      bool incideEn(Vertice v) const {
15        return (v == v1 || v == v2) ? true : false;
16      }
17      pair<bool,Vertice> incideEnDosVertices(const set<Vertice> &
            conjVertices) const {
18        pair<bool,Vertice> res;
19        int incideEnV1 = conjVertices.count(v1);
20        int incideEnV2 = conjVertices.count(v2);
21        if (incideEnV1 + incideEnV2 == 2) {
22          res.first = true;
23        } else {
24          res.first = false;
25          res.second = (incideEnV1 == 0) ? v1 : v2;
26        }
27        return res;
28      }
29      pair<Vertice,Vertice> dameVertices() const {
30        return pair<Vertice,Vertice>(v1,v2);
31      }
32      Vertice dameVerticeUno() const {
33        return v1;
34      }
35      Vertice dameVerticeDos() const {
36        return v2;
37      }
38      Vertice dameElOtroVertice(Vertice v) const { // Requiere que (v
            == v1) ó (v == v2)
39        return (v == v1) ? v2 : v1;
40      }
41  };
42
43  struct comparacionArista {
44    bool operator() (const Arista & lhs, const Arista & rhs) const {
45      // Quiero que:
46      // Si tienen = costo, ordene por costo
47      // Si tienen != costo, ordene por vértices (excepto el caso
            particular en que sean e = (a,b,c), e' = (b,a,c); entonces e
            = e')
48      if (lhs.costo() == rhs.costo()) { // (x1,y1,c), (x2,y2,c)
49        if( lhs.dameVerticeUno() == rhs.dameVerticeUno() ) { //
              (a,y1,c), (a,y2,c)
50          return lhs.dameVerticeDos() < rhs.dameVerticeDos();
51        } else { // (a,y1,c), (b,y2,c) con a != b
52          if (lhs.dameVerticeUno() == rhs.dameVerticeDos() &&
                lhs.dameVerticeDos() == rhs.dameVerticeUno()) {
53            return false; // (a,b,c) y (b,a,c) son la misma arista
54          } else {
55            return lhs.dameVerticeUno() < rhs.dameVerticeUno();
56          }
57        }
```

```
58      } else {
59        return lhs.costo() < rhs.costo();
60      }
61    }
62  };
63
64  enum Color {BLANCO, GRIS, NEGRO};
65
66  struct infoVerticeDFS {
67    Vertice verticeAnterior;
68    Arista aristaAnterior;
69    Color estado;
70    list<Arista> backEdges;
71    infoVerticeDFS() : verticeAnterior(-1), aristaAnterior(),
          estado(BLANCO), backEdges() {}
72  };
73
74  void DFS_visit(vector< list<Arista> > & aristasDeCadaVerticeAGM,
       infoVerticeDFS * info, Vertice actual) {
75    info[actual].estado = GRIS;
76    for (auto it = aristasDeCadaVerticeAGM[actual].begin(); it !=
          aristasDeCadaVerticeAGM[actual].end(); it++) {
77      Vertice nuevoActual = it->dameElOtroVertice(actual);
78      if (info[nuevoActual].estado == BLANCO) {
79        info[nuevoActual].aristaAnterior = *it;
80        info[nuevoActual].verticeAnterior = actual;
81        DFS_visit(aristasDeCadaVerticeAGM, info, nuevoActual);
82      } else if (info[nuevoActual].estado == GRIS) {
83        if (nuevoActual == info[actual].verticeAnterior) {
84          /* Si estoy acá es porque estaba volviendo por la misma
                arista que vine (formando un circuito NO simple) */
85        } else {
86          /* El nuevo vertice ya habia sido visitado, entonces hay back
                edge */
87          info[nuevoActual].backEdges.push_back(*it);
88        }
89      }
90    }
91    info[actual].estado = NEGRO;
92  }
93
94  int main(int argc, const char* argv[]) {
95    unsigned int n, m, costoTotal = 0; // n = #vertices, m = #aristas
96    vector< list<Arista> > aristasDeCadaVertice;
97    vector< list<Arista> > aristasDeCadaVerticeAGM;
98    set<Arista, comparacionArista> aristasGrafo;
99    set<Vertice> verticesAGM;
100   set<Arista, comparacionArista> aristasAGM;
101   set<Arista, comparacionArista> aristasCandidatasAGM;
102   list<Arista> aristasAnillo;
103
104   cin >> n >> m;
105
106   if (m < n) { // Si hay menos de n aristas, no existe solución (ver
          Correctitud)
107     cout << "no" << endl;
108     return 0;
109   }
```

```
110
111   for (unsigned int i = 0; i < n; i++) {
112     aristasDeCadaVertice.push_back(list<Arista>());
113     aristasDeCadaVerticeAGM.push_back(list<Arista>());
114   }
115
116   for (unsigned int i = 0; i < m; i++) {
117     Vertice v1, v2;
118     int l;
119     cin >> v1 >> v2 >> l;
120     v1--; v2--; // Como los equipos van de 1 a n, resto uno para que
          v1 y v2 vayan de 0 a n-1. Al devolver la solución sumo 1 y
          listo
121     Arista a(v1, v2, l);
122     aristasDeCadaVertice[v1].push_back(a);
123     aristasDeCadaVertice[v2].push_back(a);
124     aristasGrafo.insert(a);
125   }
126
127   // Arranco poniendo el vértice 0 en verticesAGM, y sus aristas en
       aristasCandidatasAGM
128   verticesAGM.insert(0);
129   for(auto it = aristasDeCadaVertice[0].begin(); it !=
       aristasDeCadaVertice[0].end(); it++) {
130     aristasCandidatasAGM.insert(*it);
131   }
132
133   while (aristasAGM.size() < n - 1 && aristasCandidatasAGM.size() > 0
       ) {
134     auto iterAristaMinima = aristasCandidatasAGM.begin();
135     Arista a = *iterAristaMinima;
136     aristasCandidatasAGM.erase(iterAristaMinima);
137     pair<bool,Vertice> infoIncidencia =
          a.incideEnDosVertices(verticesAGM);
138     if (infoIncidencia.first) { // Si incide en 2 vertices, no la
          puedo usar
139       continue;
140     } else {
141       Vertice nuevo = infoIncidencia.second; // Este es el vértice en
            el que no incide, no está en AGM
142       verticesAGM.insert(nuevo);
143       aristasGrafo.erase(a);
144       aristasAGM.insert(a);
145       Vertice otro = a.dameElOtroVertice(nuevo);
146       aristasDeCadaVerticeAGM[nuevo].push_back(a);
147       aristasDeCadaVerticeAGM[otro].push_back(a);
148       for(auto it = aristasDeCadaVertice[nuevo].begin(); it !=
            aristasDeCadaVertice[nuevo].end(); it++) {
149         if (it->incideEnDosVertices(verticesAGM).first) {
150           continue; // Si estoy acá, iba a agregar una arista
                repetida o ya considerada
151         }
152         aristasCandidatasAGM.insert(*it);
153       }
154     }
155   }
156
```

```
157   if (aristasAGM.size() < n - 1) { // Si el árbol no tiene n - 1
          aristas, no es generador (el grafo original no era conexo)
158     cout << "no" << endl;
159     return 0;
160   }
161
162   // Ahora agrego la arista con menor peso:
163   if (aristasGrafo.size() == 0) { // En aristasGrafo quedaron las
          aristas que no puse en el AGM
164     cout << "no" << endl;
165     return 0;
166   }
167   Arista menor = *aristasGrafo.begin();
168   aristasAGM.insert(menor);
169   for (auto it = aristasAGM.begin(); it != aristasAGM.end(); it++) {
170     costoTotal += it->costo();
171   }
172   Vertice primero = menor.dameVerticeUno();
173   Vertice segundo = menor.dameVerticeDos();
174   aristasDeCadaVerticeAGM[primero].push_back(menor);
175   aristasDeCadaVerticeAGM[segundo].push_back(menor);
176   // Tengo que encontrar el circuito simple, esto es, el anillo
177   infoVerticeDFS info[n];
178   DFS_visit(aristasDeCadaVerticeAGM, info, primero); // Llamo a DFS
          con un vértice que sé que está en el ciclo
179   if (info[primero].backEdges.size() != 1) {
180     cout << "Hay␣algo␣mal,␣el␣vertice␣deberia␣tener␣exactamente␣un␣
            back␣edge." << endl;
181     return 1;
182   }
183   Arista backEdge = info[primero].backEdges.front();
184   aristasAnillo.push_back(backEdge);
185   aristasAGM.erase(backEdge);
186   Vertice actual = backEdge.dameElOtroVertice(primero);
187   while (actual != primero) {
188     aristasAGM.erase(info[actual].aristaAnterior); // En aristasAGM
          van a quedar las aristas fuera del circuito
189     aristasAnillo.push_back(info[actual].aristaAnterior); // Lo
          contrario para aristasAnillo
190     actual = info[actual].verticeAnterior;
191   }
192   cout << costoTotal << "␣" << aristasAnillo.size() << "␣" <<
        aristasAGM.size() << endl;
193   for (auto it = aristasAnillo.begin(); it != aristasAnillo.end();
        it++) {
194     cout << it->dameVerticeUno() + 1 << "␣" << it->dameVerticeDos() +
          1 << endl;
195   }
196   for (auto it = aristasAGM.begin(); it != aristasAGM.end(); it++) {
197     cout << it->dameVerticeUno() + 1 << "␣" << it->dameVerticeDos() +
          1 << endl;
198   }
199
200   return 0;
201 }
```

4.3.2. Código del generador de grafos aleatorios

```
Vertice seleccionarVerticeRandom(set<Vertice> & conjunto) {
  int i = rand() % conjunto.size();
  auto it = conjunto.begin();
  for(int j = 0; j < i; j++) it++;
  return *it;
}

int main(int argc, const char* argv[]) {
  srand(time(NULL) + getpid()); // Seedeo
  for (int n = 1; n <= MAX_VERTICES; n++) {
    for (int i = 1; i <= CANT_INSTANCIAS; i++) {
      int m = rand() % (1 + n * (n - 1) / 2); // m es un valor
         aleatorio entre 0 y n(n-1)/2
      cout << n << "␣" << m << endl;
      set<Vertice> vertices; // Acá voy a tener el conjunto de
         vértices que todavía al menos una arista disponible.
      for (int i = 0; i < n; i++) {
        vertices.insert(i); // Los vertices van a ser enteros entre 0
           y n-1, tengo que sumarles uno al imprimir
      }
      vector< set<Vertice> > vecinosPosibles(n);
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
          if (i != j) {
            vecinosPosibles[i].insert(j);
          }
        }
      }
      while (m > 0) {
        Vertice v = seleccionarVerticeRandom(vertices);
        if (vecinosPosibles[v].size() == 0) {
          vertices.erase(v);
        } else {
          m--;
          Vertice w = seleccionarVerticeRandom(vecinosPosibles[v]);
          vecinosPosibles[v].erase(w);
          vecinosPosibles[w].erase(v);
          cout << v + 1 << "␣" << w + 1 << "␣" << (rand() % 100) + 1
             << endl;
        }
      }
    }
  }
}
```

4.3.3. Código del generador de grafos completos

```
int main(int argc, const char* argv[]) {
  srand(time(NULL) + getpid()); // Seedeo
  for (int n = 1; n <= MAX_VERTICES; n++) {
    for (int i = 1; i <= CANT_INSTANCIAS; i++) {
      int m = n * (n - 1) / 2;
      cout << n << "␣" << m << endl;
      for (int v = 1; v <= n; v++) {
```

```
8           for (int a = 1; a <= (n-1) - (v-1); a++) {
9             cout << v << "␣" << (v + a) << "␣" << (rand() % 100) + 1 <<
                  endl;
10          }
11        }
12      }
13    }
14  }
```

4.3.4. Código del tester (sólo lo relevante, el resto es igual)

```
1   chrono::microseconds algoritmo(unsigned int n, unsigned int m,
      unsigned int costoTotal, vector< list<Arista> >
      aristasDeCadaVertice, vector< list<Arista> >
      aristasDeCadaVerticeAGM, set<Arista, comparacionArista>
      aristasGrafo, set<Vertice> verticesAGM, set<Arista,
      comparacionArista> aristasAGM, set<Arista, comparacionArista>
      aristasCandidatasAGM, list<Arista> aristasAnillo)
2   {
3     auto start_time = chrono::high_resolution_clock::now();
4     // Arranco poniendo el vértice 0 en verticesAGM, y sus aristas en
        aristasCandidatasAGM
5     verticesAGM.insert(0);
6     for(auto it = aristasDeCadaVertice[0].begin(); it !=
        aristasDeCadaVertice[0].end(); it++) {
7       aristasCandidatasAGM.insert(*it);
8     }
9
10    while (aristasAGM.size() < n - 1 && aristasCandidatasAGM.size() > 0
        ) {
11      auto iterAristaMinima = aristasCandidatasAGM.begin();
12      Arista a = *iterAristaMinima;
13      aristasCandidatasAGM.erase(iterAristaMinima);
14      pair<bool,Vertice> infoIncidencia =
          a.incideEnDosVertices(verticesAGM);
15      if (infoIncidencia.first) {
16        continue;
17      } else {
18        Vertice nuevo = infoIncidencia.second;
19        verticesAGM.insert(nuevo);
20        aristasGrafo.erase(a);
21        aristasAGM.insert(a);
22        Vertice otro = a.dameElOtroVertice(nuevo);
23        aristasDeCadaVerticeAGM[nuevo].push_back(a);
24        aristasDeCadaVerticeAGM[otro].push_back(a);
25        for(auto it = aristasDeCadaVertice[nuevo].begin(); it !=
            aristasDeCadaVertice[nuevo].end(); it++) {
26          if (it->incideEnDosVertices(verticesAGM).first) {
27            continue;
28          }
29          aristasCandidatasAGM.insert(*it);
30        }
31      }
32    }
33
34    if ( (aristasAGM.size() < n - 1) || (aristasGrafo.size() == 0)) {
```

```
35      // No hay solucion
36    } else {
37      Arista menor = *aristasGrafo.begin();
38      aristasAGM.insert(menor);
39      for (auto it = aristasAGM.begin(); it != aristasAGM.end(); it++) {
40        costoTotal += it->costo();
41      }
42      Vertice primero = menor.dameVerticeUno();
43      Vertice segundo = menor.dameVerticeDos();
44      aristasDeCadaVerticeAGM[primero].push_back(menor);
45      aristasDeCadaVerticeAGM[segundo].push_back(menor);
46      // Tengo que encontrar el circuito simple, esto es, el anillo
47      infoVerticeDFS info[n];
48      DFS_visit(aristasDeCadaVerticeAGM, info, primero);
49      if (info[primero].backEdges.size() != 1) {
50        cout << "Hay algo mal, el vertice deberia tener exactamente un
             back edge." << endl;
51      }
52      Arista backEdge = info[primero].backEdges.front();
53      aristasAnillo.push_back(backEdge);
54      aristasAGM.erase(backEdge);
55      Vertice actual = backEdge.dameElOtroVertice(primero);
56      while (actual != primero) {
57        aristasAGM.erase(info[actual].aristaAnterior);
58        aristasAnillo.push_back(info[actual].aristaAnterior);
59        actual = info[actual].verticeAnterior;
60      }
61    }
62
63    auto end_time = chrono::high_resolution_clock::now();
64    chrono::microseconds tiempo =
        chrono::duration_cast<chrono::microseconds>(end_time -
        start_time);
65    return tiempo;
66  }
67
68  const int CANT_INSTANCIAS = 1000;
69  const int MAX_VERTICES = 100;
70  const int CANT_REPETICIONES_CALCULO_INSTANCIA = 10;
71
72  int main(int argc, const char* argv[]) {
73    for (int cantVertices = 1; cantVertices <= MAX_VERTICES;
        cantVertices++) {
74      int promedio = 0;
75      for (int instancia = 1; instancia <= CANT_INSTANCIAS;
          instancia++) {
76        unsigned int n, m, costoTotal = 0;
77        vector< list<Arista> > aristasDeCadaVertice;
78        vector< list<Arista> > aristasDeCadaVerticeAGM;
79        set<Arista, comparacionArista> aristasGrafo;
80        set<Vertice> verticesAGM;
81        set<Arista, comparacionArista> aristasAGM;
82        set<Arista, comparacionArista> aristasCandidatasAGM;
83        list<Arista> aristasAnillo;
84
85        cin >> n >> m;
86
87        for (unsigned int i = 0; i < n; i++) {
```

```
88          aristasDeCadaVertice.push_back(list<Arista>());
89          aristasDeCadaVerticeAGM.push_back(list<Arista>());
90        }
91
92        for (unsigned int i = 0; i < m; i++) {
93          Vertice v1, v2;
94          int l;
95          cin >> v1 >> v2 >> l;
96          v1--; v2--;
97          Arista a(v1, v2, l);
98          aristasDeCadaVertice[v1].push_back(a);
99          aristasDeCadaVertice[v2].push_back(a);
100         aristasGrafo.insert(a);
101       }
102
103       chrono::microseconds minTiempo(INT_MAX);
104       for (int rep = 1; rep <= CANT_REPETICIONES_CALCULO_INSTANCIA;
              rep++) {
105         // Notar que todo se pasa por copia para que no se modifique.
106         chrono::microseconds tiempoRep = algoritmo(n, m, costoTotal,
                aristasDeCadaVertice, aristasDeCadaVerticeAGM,
                aristasGrafo, verticesAGM,aristasAGM,
                aristasCandidatasAGM, aristasAnillo);
107         if (tiempoRep < minTiempo)
108           minTiempo = tiempoRep;
109       }
110
111       promedio += minTiempo.count();
112     }
113     promedio = promedio / CANT_INSTANCIAS;
114     cout << cantVertices << "␣" << promedio << endl;
115   }
116
117
118   return 0;
119 }
```

Bibliografía

[1] Thomas H. Cormen, *Introduction to Algorithms*. The MIT Press, Massachusetts, 3rd edition, 2009.