Trabajo Práctico 2

Viernes 7 de noviembre de 2014 — Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Florencia Lagos	313/12	cazon_88@hotmail.com
Martín Caravario	470/12	martin.caravario@gmail.com
Federico Hosen	825/12	fhosen@hotmail.com
Bruno Winocur	906/11	brunowinocur@gmail.com

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

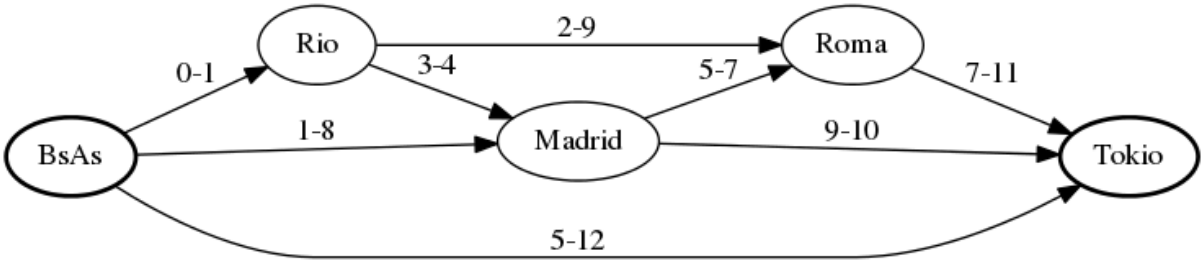
Índice

1. Problema 1

1.1. Descripción del problema

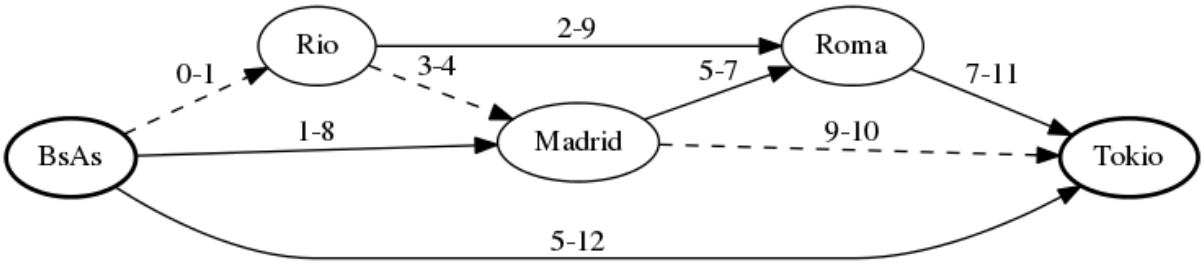
El problema Plan de vuelo pide resolver para una agencia de vuelos una nueva forma de armar itinerarios de viajes. Quieren garantizar a los clientes que indicando una salida y un destino el sistema les encuentra la ruta que, sin importar la cantidad de tramos que requiera, lo hace llegar lo antes posible en el destino final. La única restricción que presenta el problema, es que debe haber por lo menos dos horas entre un vuelo elegido y el siguiente, filtrando de esta forma las opciones de vuelos a tomar.

Se pide entonces buscar la combinación óptima de vuelos de forma tal que se minimize el horario de llegada, sin importar la cantidad de vuelos tomados ni la cantidad de ciudades visitadas. Se puede interpretar al problema como una variante de camino mínimo, siendo la *distancia* la hora de llegada a destino.
Un posible ejemplo en el que *origen* es Buenos Aires y *destino* es Tokio, podría ser el siguiente



Instancia

En esta instancia la solución del problema sería tomar los vuelos de Buenos Aires a Rio, luego de Rio a Madrid y finalmente de Madrid a Tokio.



Solución

Análizando el problema, dimos con una propediedad que nos ayudará a resolver el ejercicio, y que será la base del algoritmo que propondremos. Ésta es, teniendo una posible ruta de viaje y dado un vuelo, si con dicho vuelo no llego a la ciudad destino, entonces éste no me servirá en ninguna otra ruta de viaje posible, pues no importa *cómo* llegue a tomar el vuelo, una vez que lo analizo y no sirve para llegar a destino, no servirá en otra situación.

1.2. Ideas desarrolladas

Como dijimos anteriormente, en este problema necesitamos buscar la combinación óptima de vuelos para minimizar el tiempo de llegada a la ciudad destino.

Para pensar en la resolusión del problema, decidimos utilizar programación dinámica. Con ésto en mente, desarrollamos un algoritmo que no estamos seguros de que sea clasificable como programación dinámica, pero es recursivo y se basa en la idea de que una vez analizada una instancia del algoritmo, nunca será evaluada otra vez.

La idea es tratar de reconstruir una ruta de vuelo desde destino hasta origen. Es decir, tomar el vuelo de menor hora de llegada a destino y desde la ciudad de salida de ese vuelo, fijarse qué vuelos llegan a esa ciudad y tomar alguno de esos (que sea factible en términos de los horarios) y recursivamente ir hacia atrás hasta llegar a la ciudad origen.

Una vez que se elige un vuelo, éste es removido de la lista de vuelos existentes, pues si la ruta construida es fructífera, entonces volver a tomar el mismo vuelo es imposible, pues estaría viajando en

el tiempo, y si la ruta que estoy construyendo no llega a buen puerto, entonces (como mencionamos en la descripción del problema) el vuelo elegido *nunca* llevará al origen.

Tendremos entonces un algoritmo principal que inicializará las estructuras necesarias y luego llamará a una función recursiva que empieze a buscar la ruta de vuelo. Es la función recursiva en cuestión la que hace todo el trabajo.

El objetivo de la función recursiva es que dada una ciudad y un horario, vaya probando para cada vuelo que llega a dicha ciudad (y sea factible haberlo tomado en relación al horario) si hay una ruta que llegue a éste. Es decir, la función eligirá un vuelo, y se llamará a sí misma recursivamente, tratando de armar una ruta de vuelo que con la ciudad de salida de dicho vuelo, y con su horario de salida.

Ahí está el paso recursivo, se llamará a la misma función hasta que se encuentre una ruta de vuelo, y si no se encuentra el algoritmo avisará (de alguna forma) tal cosa.

De esta forma se terminan revisando (potencialmente) todos los vuelos, pero *sólo* una vez, pues una vez revisado un vuelo, éste se quita de la lista de vuelos, la cual es global para cualquier instancia de la función.

A grandes rasgos, lo que hace la función es dada una ciudad y un horario, buscar si existe alguna forma que habilite a llegar a dicha ciudad cumpliendo con el horario; es decir, si existe alguna secuencia de vuelos (que cumplan las restricciones de horarios), que parta desde la ciudad origen y termine en la ciudad pasada como parámetro.

Para hacer ésto, *buscar_ruta* consigue una lista de vuelos predecesores de una ciudad, es decir, aquellos vuelos que llegan a dicha ciudad. La lista está ordenada por horario de llegada, con lo cual el primer elemento es aquel vuelo que llega primero a la ciudad.

Entonces, para que buscar_ruta encuentre el camino óptimo, es necesario que se invoque la primera vez (desde el algoritmo principal) con la ciudad destino como parámetro, y con el máximo valor posible como horario limitante. Pues así, buscará una ruta de vuelo que llegue a origen, y empezará a buscar con el primer vuelo que llegue a la ciudad destino.

Estando en una ciudad, para poder saber qué vuelos puedo tomar (hacia atrás) voy a necesitar el conjunto de los vuelos que llegan a ahí. Además me va interesar tener dicha estructura ordenada, diremos entonces que tendremos un arreglo de secuencias de vuelos, que por cada ciudad me indica cuáles son los vuelos que llegan hasta ahí.

También necesitaremos un lugar donde podamos ir armando la ruta que se va formando a medida que voy eligiendo vuelos. Esto lo haremos con un arreglo, que dada una ciudad nos indique cuál es el vuelo que fue elgido para llegar hasta ahí. De esta forma, para reconstruir el resultado, simplmente basta con ir reconstruyendo la ruta de vuelos, empezando desde la ciudad destino.

Es importante aclarar para la comprensión del pseudocódigo, que los vuelos están numerados (tienen un ID) de 0..#vuelos-1 y las ciudades de la misma forma, siendo la ciudad 0 la del origen, y la #ciudades-1 la destino.

1.3. Pseudocódigo

```
RUTA_DE_VUELO(in lista(vuelo): vuelos, in ciudad: origen, in ciudad: destino)→ (lista(vuelo), hora)
 1  int m ← cantidadCiudades()
 2  arreglo(secuencia(vuelo)) vuelos_hasta(m) ← ubicar_vuelos_por_ciudad(vuelos)
 3  arreglo(vuelo) ruta ← vacio(m)
 4  bool hay_solucion ← buscar_ruta(+∞, DESTINO, vuelos_hasta, ruta)
 5  if hay_solucion
 6     then devolver (reconstruir_ruta(ruta), hora de llegada de ruta[m-1])
 7     else  devolver (vacio(), INVALIDO)
 8  endif
```

```
BUSCAR_RUTA(in hora: h, in ciudad: c, in arreglo(secu(vuelo)): vuelos_hasta, in arreglo(vuelo): ruta)→ bool
 1  if c = ORIGEN
 2     then devolver verdadero
 3     else  secuencia(vuelo) predecesores ← dame_predecesores(h, vuelos_hasta[id de c])
 4           bool hay_ruta ← falso
 5           vuelo candidato
 6           while ¬hay_ruta ∧ haya vuelos en predecesores
 7             candidato ← tomar primero de predecesores y eliminarlo
 8             ruta[id de c] ← candidato
 9             hay_ruta ← buscar_ruta(candidato.hora_salida, candidato.ciudad_salida, vuelos_hasta, ruta)
10           endwhile
11           devolver hay_ruta
12  endif
```

```
RECONSTRUIR_RUTA(in arreglo(vuelo): ruta)→ lista(vuelo)
1  lista(vuelo) res ← vacio()
2  ciudad c_actual ← DESTINO
3  while c_actual sea distinta de ORIGEN
4    agregar ruta[id de c_actual] a res
5    c_actual ← ruta[id de v].ciudad_salida
6  endwhile
7  devolver res
```

Definiciones de constantes y funciones auxiliares:

- **cantidadCiudades**, es una función que devuelve la cantidad de ciudades.
- **ubicar_vuelos_por_ciudad**, es una función que ubica cada vuelo en la posición del arreglo *vuelos_hasta* correspondiente a la ciudad de llegada del vuelo en cuestión, además, los agrega de forma ordenada (crecientemente) según el horairo de llegada.
- **dame_predecesores**, es una función que devuelve una secuencia de vuelos ordenados por orden de llegada, que lo que hace es filtrar la secuencia pasada como parámetro según la hora de salida del vuelo pasado por parámetro. Devuelve los *posibles* vuelos que *se pueden haber tomado* para llegar a la ciudad de salida del vuelo pasado por parámetro y luego tomar éste.
- **vuelo**, es un tipo de dato que tiene toda la información correspondiente al vuelo, inclusive un entero ID que con el cual es indexado en el arreglo *ruta*.
- **ORIGEN**, es un valor unívoco que identifica a la ciudad origen.
- **DESTINO**, es un valor unívoco que identifica a la ciudad destino.
- **INVALIDO**, es una constante que indica que la hora es inválida.

1.4. Análisis de complejidad

Antes de analizar la complejidad del algoritmo que resuelve el problema, veamos qué complejidad tienen las dos funciones auxiliares explicitadas en el pseudocódigo. Para no repertir en varias ocasiones lo mismo, diremos que n es la cantidad de vuelos, y m la cantidad de ciudades, que a su vez es acotable por n.

1.4.1. reconstruir_ruta

En la primer línea se crea una lista vacía, cuyo costo es constante.

A continución nos encontramos con un ciclo, que itera hasta que se encuentre con la ciudad *origen*. En principio no se ve cuántas iteraciones tendrá dicho ciclo, pero se observa que recorre el arreglo de forma recursiva hasta llegar al *origen*.

Asumamos que el algoritmo **ruta_de_vuelo** funciona, y que entonces se puede reconstruir una ruta hasta encontrarse con una ciudad, cuyo vuelo tenga como ciudad de salida a la ciudad de origen. Entonces, a lo sumo recorreré todas las ciudades pues todas fueron visitadas.

Entonces, al recorrer todas las ciudades, el ciclo itera m veces, y las operaciones contenidas dentro del ciclo son de costo constante. Luego, la complejidad del algoritmo es $O(m)$ que es $O(n)$.

1.4.2. buscar_ruta

Al comenzar la función buscar_ruta nos encontramos con un **if**, si la guarda se cumple el costo de la función es constante. Veamos que pasa en caso contrario.

En la línea 3 del pseudocódigo se invoca a la función **dame_predecesores**, que $filtra$ la secuencia pasada como paŕametro. Si fuese éste el caso, esta función sería lineal. Para evitar ésto, es implementada de tal forma que devuelve un iterador al primer *predecesor* y en algún momento alcanza al último predecesor. Pues hay que tener en cuenta que un predecesor p de un vuelo v es aquel cuya ciudad de llegada, es la ciudad de salida de v, y además el horario de llegada de p tiene que ser por lo menos 2 horas menor al horario de salida de v (a menos claro que se trate de un vuelo que sale de origen).

Entonces, como la secuencia que recibe la función ya está ordenada por horario de salida, y tiene *solamente* los vuelos que llegan a la ciudad de salida de v, entonces simplemente hay que buscar el primero y el último que cumplan la condición. Luego, todos los elementos entre éstos son la lista de predecesores.

Cada secuencia dentro de **vuelos_hasta** está implementada con un set[1], con lo cual, la búsqueda tiene costo $log(n)$.

Entonces,por todo lo explicado anteriormente, armar la lista de predecesores cuesta $O(log(n))$.

La parte rebuscada del análisis se da ahora, una llamada recursiva dentro del ciclo, cuya complejidad está relacionada en ambos sentidos con la llamada recursiva. Pues dichas llamadas, modifican potencialmente la cantidad de iteraciones del ciclo de la instancia madre.

Por esta razón no pudimos encontrar una expresión matemática que describa el comportamiento del costo computacional de la función. Pero lo que sí es seguro es que cuando un vuelo es analizado por la función, éste es removido y nunca es analizado otra vez. Entonces a lo sumo voy a revisar todos los vuelos que existen, y para cada uno de ellos buscar su lista de predecesores y eliminarlo de la estructura global que lo contiene.

Entonces, la complejidad de buscar_ruta, instanciada desde la ciudad destino es $O(n * log(n))$.

Éste algoritmo tiene un mejor y un peor caso. Si la instancia es lo suficientemente buena el algoritmo es constante. Pues si el primer vuelo que llega a destino sale de origen el algoritmo termina haciendo sólo una llamada recursiva.

Si en cambio, la instancia es lo suficientemente mala, el algoritmo sí o sí recorre todos los vuelos, quedando la complejidad que analizamos anteriormente. Esta instancia sería una en la cual la ruta óptima se desprenda de resolver la última rama del árbol de recursión, o que directamente no haya solución.

[1] Contenedor set de la librería estándar de c++. Referencia en `http://en.cppreference.com/w/cpp/container/set`.

1.4.3. ruta_de_vuelo

Analicemos ahora al algoritmo principal.

Al comenzar, el algoritmo calcula la cantidad de ciudades en base a la lista de vuelos. En la implentación está hecho con un Trie, pues por otro propósito nos interesa tener guardadas las ciudades con ID entero. Con lo cual esta función simplemente le pregunta al Trie la cantidad de elementos, que es la cantidad de ciudades.

Entonces la complejdiad de la función está dada por el costo de armar el Trie, y como la comparación entre dos strings es constante (pues están acotados), armar el Trie es $O(n)$. Y pedirle la cantidad de elementos es $O(1)$, luego el costo de ésta acción es $O(n)$.

Luego se crea una estructura que contiene a todos los vuelos, organizados por ciudades. En el arreglo **vuelos_hasta** cada posición i indica los vuelos que llegan a la ciudad identificada con i. En la implementación, en cada posición del arreglo hay un *set* de vuelos. Entonces, lo que hace la función que crea la estructura es para cada vuelo, fijarse en $O(1)$ a qué ciudad corresponde que vaya, y por lo tanto a que posición del arreglo, y luego agregarlo en el *set* correspondiente, con un costo de $O(log(n))$.

Cada uno de los n vuelos se agrega a un *set* determinado, obteniendo una complejidad de $O(n * log(n))$.

Luego está la llamada a buscar_ruta, que dijimos que en el peor caso cuesta $O(n * log(n))$.

En la línea 6 hay una llamada a reconstruir_ruta, cuya complejidad es lineal en m, que en el peor caso es n.

La complejidad final es entonces **O(n*log(n))** y no hay peor caso, pues más allá de que la instancia del problema sea buena y por lo tanto buscar_ruta se resuelva en tiempo constante, crear vuelos_hasta cuesta $O(n * log(n))$ siempre.

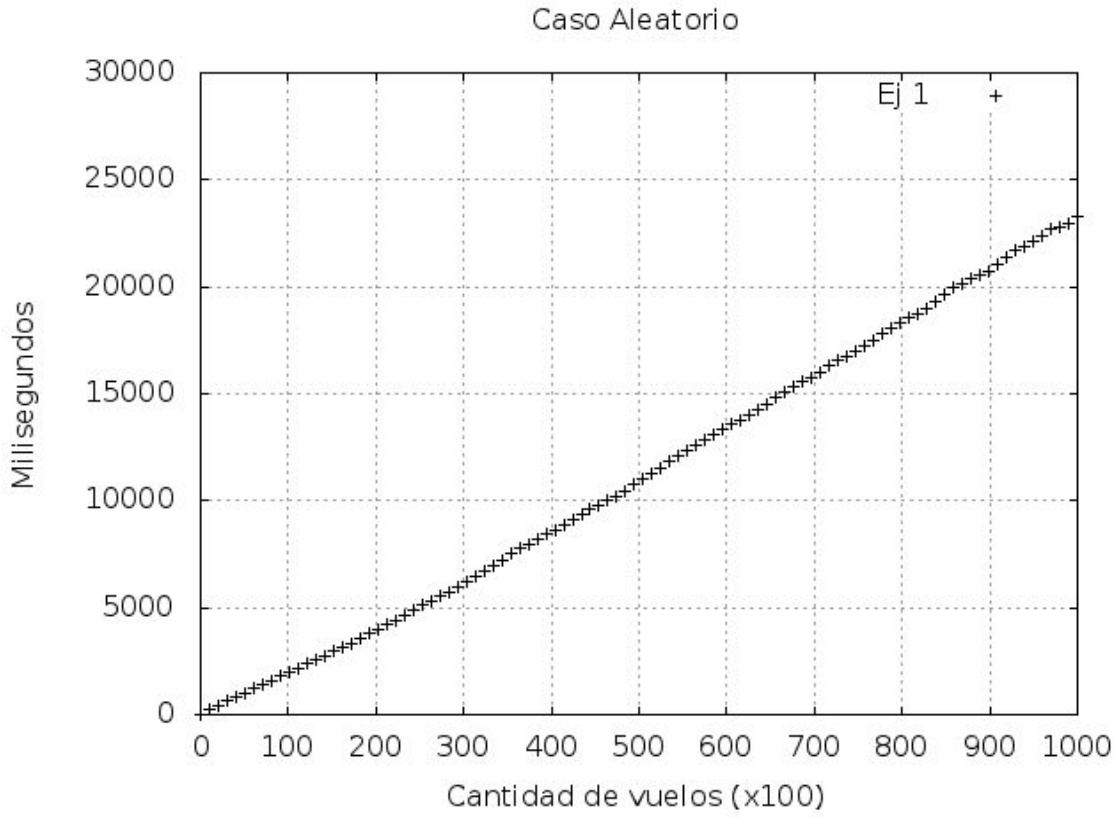
1.5. Experimentación

El método utilizado para medir los tiempos en este ejercicio consistió en correr nuestro algoritmo una cantidad fija de veces, y luego nos quedamos con el minimo tiempo. Para esto utilizamos la librería *chrono*, la cual nos facilitó las funciones adecuadas para medir los tiempos.

Como fue explicado previamente, nuestro algoritmo no presenta un peor o mejor caso, es por eso que decidimos hacer un generador aleatorio de vuelos. Este generador tiene 10 ciudades fijas (BuenosAires, Chaco, Rosario, Bariloche, Baradero, Lincol, MarDelPlata, CarlosPaz, SanPedro, SanNicolas), a las cuales se les van asignando vuelos de forma aleatoria. El generador recibe como parametro la cantidad de vuelos a realizar, un entero que representa la hora maxima de inicio de cualquier vuelo, y una semilla para generar la aleatoriedad en los vuelos. También se encarga de elegir que ciudades serán *origen* y *destino*, pero no asegura que habrá una solución posible para el ejercicio.

1.5.1. Experimento 1

En este experimento comenzamos con una instancia de 10 vuelos, y fuimos aumentando de a 50 hasta llegar a los 100000 vuelos. La hora maxima de inicio de los vuelos utilizada fue 60, y corrimos nuestro algoritmo 25 veces en cada instancia para quedarnos con el minimo. La semilla utilizada para generar aleatoriedad fue el número 18.
El objetivo de este experimento fue el de comprobar empíricamente que la complejidad teorica de nuestro algoritmo es efectivamente $O(n * log(n))$. A continuacíon se presentan los resultados.



Para realizar el gráfico, decidimos dividir a nuestra función por f(x) = log(x), esto provocó que la función se comporte como lineal, facilitandonos el análisis y confirmando que la complejidad teorica calculada se corresponde con la del algoritmo.
Se puede deducir entonces que la complejidad de nuestro algoritmo termina siendo $O(n * log(n))$, con n siendo la cantidad de vuelos.

1.6. Anexo reentrega

Para poder combinar un vuelo luego de otro, se pedía originalmente que la diferencia entre la llegada de uno y la partida del otro sea de al menos dos horas. En una nueva versión de la herramienta desarrollada, se pretende relajar esta restricción a sólo una hora de diferencia pero sólo cuando los vuelos a combinar correspondan a la misma aerolínea. Es decir, cada vuelo corresponde a una aerolínea. Al combinar un vuelo luego de otro, si ambos vuelos son de la misma aerolínea, se permite que la cantidad de horas entre estos vuelos sea de una hora en lugar de dos. ¿Cómo afecta eso a su algoritmo?

Para desarrollar estos cambios, antes que nada necesitaríamos asociar a un vuelo una aerolínea en particular. Lo que cambiaría sería la búsqueda del vuelo a tomar, en el algoritmo *buscar_ruta* al momento de fijarse cuáles son los *vuelos predecesores*, al ser la condición más relajada para vuelos de la misma aerolínea se agregarían (potencialmente) nuevos vuelos disponibles.

La complejidad teórica sería la misma, y la estructura general de todos los algoritmos también.

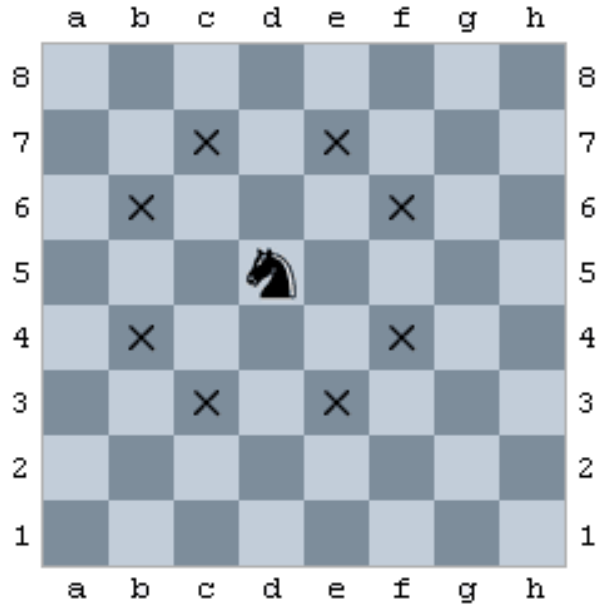
2. Problema 2

2.1. Descripción del Problema

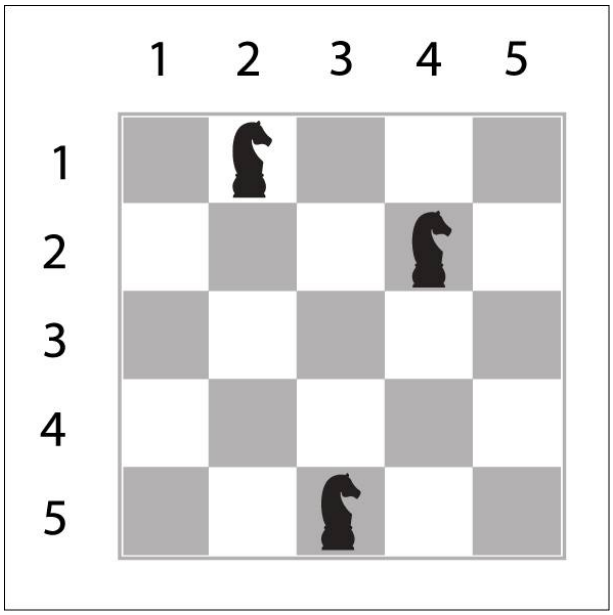
En este ejercicio nos encontramos con caballos salvajes dispersos por un tablero de ajedréz. Cada uno empieza en un casillero determinado y el problema consiste en encontrar un casillero que oficie de punto de encuentro para todos, con la condición de que para llegar al lugar, se recorra la mínima cantidad de casilleros.

Es decir, nos encontramos ante un problema de optimización (en el cual quizás no existe solución), donde tenemos que minimizar la cantidad de movimientos totales de los caballos para llegar al punto de encuentro, o sea, que la suma de los movimientos de cada caballo sea mínima.

Cada caballo tiene 8 movimientos posibles dentro de un tablero de ajedrez, siempre y cuando los margenes del tablero y la posición de este lo permitan. Un caballo puede moverse de la siguiente forma.

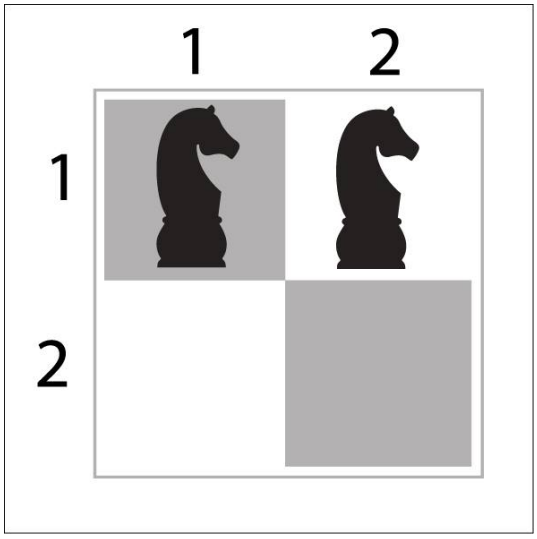


A continuación se presentan dos instancias del problema a modo de ejemplo:



Instancia 1

En este ejemplo, la solución es 2 4 3, ya que el caballo ubicado en la posición (1,2) debe realizar un movimiento y el que esta ubicado en la posición (5,3) realiza dos para moverse hacia el caballo ubicado en (2,4). Por lo tanto la posición final de todos será en el casillero (2,4) utilizando 3 movimientos entre los dos.



Instancia 2

Este ejemplo no tiene solución, ya que ninguno de los dos caballos podrá moverse hacia algún casillero por la naturaleza de sus movimientos, previamente explicados.

2.2. Ideas Desarrolladas

Como dijimos en el apartado de la descripción del problema, tenemos que encontrar un casillero que haga de punto de encuentro para los caballos de tal forma que la suma de los movimientos de cada caballo sea la mínima.

Para ésto, primero debemos saber, para cada caballo, cuál es la mínima cantidad de movimientos que le toma llegar a una determinada posición del tablero; si es que siquiera puede llegar, ya que podría no existir una combinación de movimientos que lo lleve.

Entonces, antes de pensar en la solución de problema, tenemos que armar un algoritmo que dado un tablero y un caballo ubicado en una posición inicial, me indique cuántos movimientos necesita dicho caballo para moverse a cualquier lugar del tablero.

Con lo cual habrá una matriz por caballo, y cada elemento de la matriz representará la cantidad de movimientos que le toma al caballo llegar hasta ahí desde la posición inicial. Y si hay posiciones que no pueden ser accedidas por el caballo, tendrán un valor especial que reflejará eso.

Una vez que estén armadas todas las matrices lo único que hay que hacer es chequear si *hay* solución y sumar todas las matrices. La matriz resultante tendrá en cada casillero la suma total de los movimientos de cada caballo, con lo cual basta buscar el mínimo el elmento válido (supo+ngamos que los casilleros inválidos son aquellos donde al menos un caballo no pudo acceder) y éste será la solución.

En definitiva, lo que queremos hacer es una especie de *camino mínimo* entre la posición inicial del caballo y todos los demás casilleros. Como ya vimos en las clases teóricas, dicho problema cumple el principio de optimalidad, con lo cual, es una buena idea utilizar la técnica algorítmica de programación dinámica para poder armar la matriz descripta. La técnica de la programación dinámica se ve reflejado en que para calcular los movimientos de una posición del tablero, debo utilizar casilleros previamente calculados.

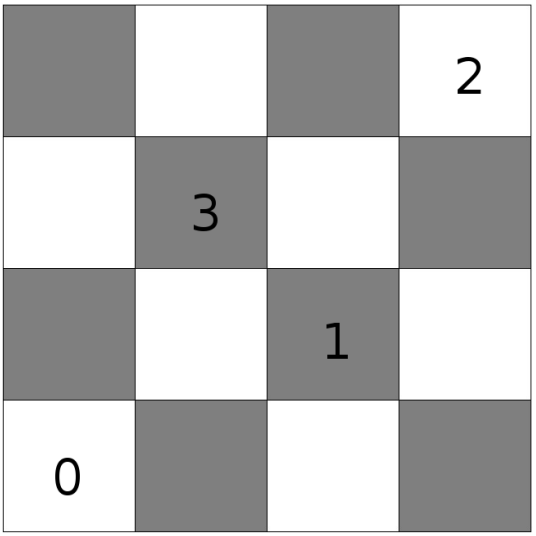
Podríamos describir el comportamiento definiendo la siguiente función recursiva:

$$f(x) = \begin{cases} min(f(v), v \in vecinos(i,j)) + 1 & si\ 0 < i \leq n\ \wedge\ 0 < j \leq n\ \wedge (i,j) \neq inicio \\ 0 & si\ (i,j) = inicio \\ \infty & sino \end{cases}$$

Siendo *inicio* la posición inicial del caballo y $vecinos(i,j) =$
$(i-2, j-1),\ (i-1, j-2),\ (i+1, j-2),\ (i+2, j-1),\ (i-2, j+1),\ (i-1, j+2),\ (i+2, j+1),\ (i+1, j+2))$

Es así que el grueso del algoritmo está en armar las k matrices [2]. Una primera idea fue implementar la función recursiva descripta anteriormente pero en forma *bottom-up*, que según los movientos de los caballos, escriba en los casilleros correspondientes la cantidad de movimientos de donde estaba parado más uno, y luego llame recursivamente sobre dichos casilleros.

[2] Una por caballo.



(a) DFS

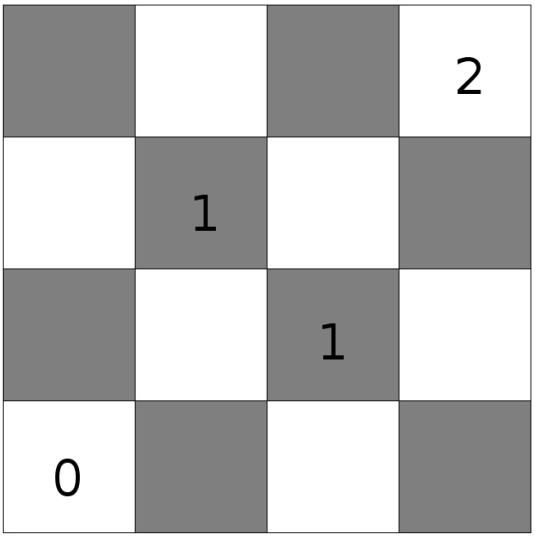
El problema de ésta solución es que las llamadas recursivas se ejecutan en un orden que no es el deseado. Como el algoritmo que arma la matriz śolo escribe cuando el elemento es inválido, si ya hay un valor computado, entonces deja ese.

O sea que, se termina ejecutanto una rama de la recursión, dejando casilleros con valores erróneos, que serían menores si fuesen llenados por recursiones de *niveles* anteriores.

En la en la figura (a) se ve cómo si vamos resolviendo las llamadas recursivas en el orden normal, hay varios casilleros que no tienen la cantidad mínima de movimientos necesarios para acceder a él desde la posición inicial.

Entonces lo que queremos hacer es establecer de manera correcta el orden en el que se va llenando la matriz. Esta forma sería primero resolver las *recursiones de primer nivel*, luego las de *segundo nivel*, etc. Si pensamos a cada casillero como un nodo, y cada movimiento como una arista, podríamos pensar al ejercicio como un grafo, en donde una arista entre dos nodos existe si hay un movimiento que los conecte. Más precisamente, podríamos pensarlo como un árbol, donde la raíz es la posición inicial.

Con ésto en mente, queremos recorrer al *grafo* como lo hace el algoritmo de BFS [3] , o sea, a lo ancho. De ésta forma me aseguro de recorrer en primera instancia todos los primeros casilleros a los que llego desde la posición inicial, luego, los casilleros a los que llego desde los segundos, y así hasta recorrer a todos los que se pueda. Ésto se ve reflejado en la figura (b).



(b) BFS

De esta forma, sabiendo que BFS nos servirá para recorrer los casilleros de la forma correcta, el algoritmo que calcula los movimientos será una modificación de este famoso algoritmo.

[3] Referencia en `http://en.wikipedia.org/wiki/Breadth-first_search`

Vamos a necesitar entonces k matrices, donde k es la cantidad de caballos, y una matriz final para acumular la suma de todas las matrices:

- **tableros_caballos**, arreglo de matrices de enteros. En la posición k tiene la matriz del $k - esimo$ caballo. La posición i, j de la matriz k representa la cantidad mínima de movimientos que le toma al caballo k llegar hasta el casillero i, j desde su posición inicial, en caso de que el caballo no pueda llegar al casillero, se le asigna el valor -1.
- **matriz_res**, matriz de enteros donde la posición i, j tiene la suma de los movimientos mínimos que le toma a cada caballo llegar hasta allí desde su posición inicial. Tiene un -1 si algún caballo no puede llegar a dicha posición.
- **pos_ini_caballos**, arreglo de duplas de enteros donde la posición i indica el casillero del tablero donde empieza el caballo i.

2.3. Pseudocódigo

```
CABALLOS_SALVAJES(in cant_caballos: int, in tam: int, in pos_ini_caballos: arreglo(int, int)) → (int, int, int)
 1  arreglo(matriz(int)) tableros_caballos ← crear_un_tablero_por_caballo(tam, cant_caballos)
 2  for i ← 0 hasta cant_caballos
 3    inicializar_casilleros_con(tableros_caballos[i], −1)
 4    escribir_casillero(tableros_caballos[i], pos_ini_caballos[i], 0)
 5    calcular_movientos(tableros_caballos[i], pos_ini_caballos[i], tam)
 6  endfor
 7  matriz(int) suma_total ← suma_especial_matrices(tableros_caballos)
 8  if son_todos_negativos(suma_total)
 9     then return (−1, −1, −1)
10     else  return minimo_no_negativo(suma_total)
11  endif
```

```
CALCULAR_MOVIMIENTOS(in/out m: matriz(int), in pos: (int, int), in tam: int)
 1  cola(int, int) proximos_casilleros ← vacia()
 2  proximos_casilleros.enconlar(pos)
 3  while ¬cola.vacia
 4    (int, int) actual ← proximos_casilleros.primero() //Esta operación desencola el primer elemento.
 5    lista(int, int) vecinos ← calcular_vecinos(actual, tam)
 6    for v ∈ vecinos
 7      if m[v] < 0  //Si el valor es negativo, el casillero no fue visitado.
 8         then m[v] ← m[actual] + 1
 9              proximos_casilleros.encolar(v)
10      endif
11    endfor
12  endwhile
```

Funciones Auxiliares:

- arreglo(matriz(int)) **crear_un_tablero_por_caballo**(int tam, int cant), crea *cant* matrices de tamaño $tam * tam$ en un arreglo.
- **inicializar_casilleros_con**(matriz(int) m, int valor), inicializa todas las posiciones de la matriz m con el *valor* pasado por parámetro.
- **escribir_casillero_con**(matriz(int) m, (int,int) pos, int valor), escribe en la posición determinada por *pos* el entero *valor*.
- matriz(int) **suma_especial_matrices**(arreglo(matriz(int)) as), devuelve una matriz que es la suma de todas las matrices del arreglo as, con la particularidad de que si al momento de sumar dos posiciones, una de ellas es un valor negativo, entonces en vez de dejar la suma de los dos valores, deja un -1.
- bool **son_todos_negativos**(matriz(int) m), devuelve verdadero si todos los elementos de la matriz m son valores negativos.
- (int,int,int) **minimo_no_negativo**(matriz(int) m), devuelve el mínimo elemento no negativo de la matriz m junto con sus coordenadas.

- lista(int,int) **calcular_vecinos**((int,int) pos, int tam), devuelve una lista, de a lo sumo 8 elementos, que tiene las coordenadas de los posibles movimientos de un caballo en un tablero de ajedrez desde *pos*.

2.4. Análisis de correctitud

Llamaremos:

- M^k a la matriz correspondiente al caballo k.
- $origen_k$ a la posición inicial del caballo k en su matriz.
- $\delta((i,j),(a,b))$ a la distancia de (i,j) a (a,b).

Para la demostración, lo que queremos probar es que al buscar el mínimo elemento dentro de nuestra matriz *suma_total*, obtendremos la mínima cantidad de movimientos totales que deben realizar los caballos para encontrarse.

Para ésto, necesitamos ver que en la matriz de cada caballo, en cada posición tenemos la cantidad de minima de movimientos necesarios para llegar desde la posición inicial hasta allí.

Primero, establezcamos que nuestro algoritmo es una modificación de **BFS**. Dado un nodo, los nodos adyacentes a él son los casilleros del tablero alcanzables por él, es decir, los (a lo sumo) 8 casilleros a los cuales se puede mover el caballo. Luego, el nodo inicial es el casillero donde se encuentra el caballo al comenzar el algoritmo. Entonces, siendo éste el grafo sobre el cual trabajar, *calcular_movimientos* es una modificación de BFS, pues recorre el grafo a lo ancho, encolando los nodos adyacentes y calculando la distancia en aristas según la distancia del padre.

Ya establecido que nuestro algoritmo trabaja como un BFS, podemos afirmar que en cada casillero (alcanzable desde la posición inicial) está la distancia mínima de aristas recorridas (es decir, de movimientos). La demostración de que BFS calcula la mínima distancia en aristas desde un nodo origen a todos lo demás se encuentra en el capítulo 22, páginas 599-600, tercera edición en el teorma 22.5 del libro *Introduction to Algorithms de Cormen 3th edition, Thomas H., Leiserson Charles E., Rivest Ronald L.*.

Podemos concluir entonces que por cada posición (i,j) de cada matriz k, valdrá que:

- $M^k_{i,j} = \delta(origen_k,(i,j))$

Queda ver entonces que al sumar las k matrices, obtendremos en la posición (i,j) la sumatoria de las minimas cantidades de movimientos entre todos los caballos para alcanzar dicha posicion.

Llamemos R a la matriz que resulta de sumar las k matrices. Entonces:

- $R_{i,j} = \sum_{h=1}^{k} M^h_{i,j}$

Lo que es equivalente,

- $R_{i,j} = \sum_{h=1}^{k} \delta(origen_h,(i,j))$

Entonces, como en cada posición de la matriz resultante se encuentra la mínima cantidad de movimientos necesarios para que los caballos se encuentren en dicho casillero, si buscamos el mínimo entonces tendremos la mínima cantidad de movimientos para que los caballos se encuentren, que es exactamente lo que queríamos probar.

2.5. Análisis de la complejidad

Antes de analizar la complejidad del algoritmo que resuelve el problema, analizaremos la complejidad de una función auxiliar que es la que hace casi todo el trabajo.

2.5.1. Complejidad calcular_movimientos

El algorimo comienza creando una cola vacía, cuyo costo es constante, y luego encolando un elemento, cuyo costo es también constante.

Luego nos encontramos con un ciclo cuya guarda no nos da mucha información sobre *cuántas* veces se itera, sólo vemos que lo hará hasta que la cola esté vacía, con lo cual primero establezcamos qué cantidad de elementos tendrá, a lo sumo, dicha cola.

Vemos que lo primero que se hace en el ciclo es desencolar el primer elemento y ésto pasa *siempre*. Luego una función auxiliar crea una lista de *a lo sumo 8 elementos*, y para cada uno de ellos si el valor es negativo, se encola y se le asigna un valor positivo, si no *no se lo encola*.

Entonces, como todos los elementos empiezan en negativo menos uno de ellos que se evalúa en la primera iteración, en el peor caso (si desde cada casillero se puede acceder a sus 8 posibles vecinos) se recorren todos los elementos, ya que cuando está en negativo se encola y luego se marca como *visto*, con lo cual la próxima vez que se lo analize no se lo tendrá en cuenta. Luego, la cantidad de iteraciones es $O(n^2)$.

Ya establecida la cantidad de iteraciones del ciclo, veamos la complejidad de las operaciones que están en él.

En la línea 4 se realiza la operación **primero** de la cola, que tiene un costo constante.

Luego se invoca a la función **calcular_vecinos** que también es constante, pues para los 8 posibles movientos verifica si son válidos (es decir, si no se pasan del tamaño del tablero) y se los agrega a una lista.

A continución hay un ciclo anidado, que *itera a lo sumo 8 veces* pues es esa la cantidad máxima de elementos que devuelve calcular_vecinos. Y como la operación encolar es $O(1)$, como lo es la asignación y el chequeo del if, el costo de este for es constante.

Con lo cual la complejidad del ciclo es $O(n^2)$, pues itera n^2 veces haciendo operaciones de costo $O(1)$.

2.5.2. Complejidad caballos_salvajes

Analicemos la complejidad del algoritmo **caballos_salvajes**. Llamemos k a la cantidad de caballos y n a la cantidad de filas y columnas del tablero, siendo el tamaño de éste de $n * n$.

En el pseudocódigo se observa que en la primer línea se crean k matrices de tamaño $n * n$, diremos que el costo de crear una matriz es constante, con lo cual al creasrse k matrices ésta operación es $O(k)$.

Ahora nos encontramos con un for que itera k veces. Calculemos la complejidad de este ciclo, que hace 3 cosas:

1. **inicializar_casilleros_con**, que toma como parámetro en cada iteración una matriz de $n * n$ e inicializa cada elemento con el valor indicado. Diremos que el costo de dicha función es lineal en la cantidad de elementos de la matriz, o sea, $O(n^2)$.
2. **escribir_casillero**, que recibe como parámetros una matriz, una posición, y un valor para luego escribir en dicha posición el valor. Diremos que el costo es constantes pues acceder a una posición de una matriz es $O(1)$.
3. **calcular_movimientos**, que recibe una matriz y calcula la cantidad de movimientos mínima para llegar desde la posición inicial a cualquier casillero (alcanzable). Como mencionamos anteriormente, la complejidad de ésta función es lineal en la cantidad de elementos de la matriz, o sea $O(n^2)$.

Luego, la complejidad de este ciclo es $O(k * n^2)$.

Veamos ahora el resto del algoritmo. En la línea 7 se llama a la función axiliar **suma_especial_matrices** que suma todas las matrices con la particularidad de al toparse con un -1 en un casillero, en vez de dejar la suma de ambos casilleros, deja un -1. No se incluye el pseudocódigo de ésta función porque se considera simple. Diremos que la complejidad de ésta es lineal en la cantidad de casilleros, pues por cada matriz se recorren todos sus casilleros y se realiza una suma ($O(1)$); luego, como la cantidad de elementos de cada matriz es de $n * n$ y hay k matrices, la complejidad es $O(k * n^2)$.

Nos encontramos ahora con un if cuya guarda es invocar a la función **son_todos_negativos** que toma una matriz de tamaño n^2. Como esta función verifica el signo de cada elemento de la matriz, su complejidad es la de revisar a todos los elementos, o sea es $O(n^2)$.

Si la guarda da verdadero, se devuelve un valor *fijo*, con lo cual es constante.

Si por el contrario, no se verifica la condición, se devuelve el resultado de la función **mínimo_no_negativo**, que recorre linealmente a la matriz cuyo tamaño es n^2, con lo cual su costo computacional es $O(n^2)$.

Luego, la complejidad del algoritmo es $O(k) + O(k * n^2) + O(k * n^2) + O(n^2) + O(n^2)$, lo que es $O(k * n^2)$.

2.6. Experimentación

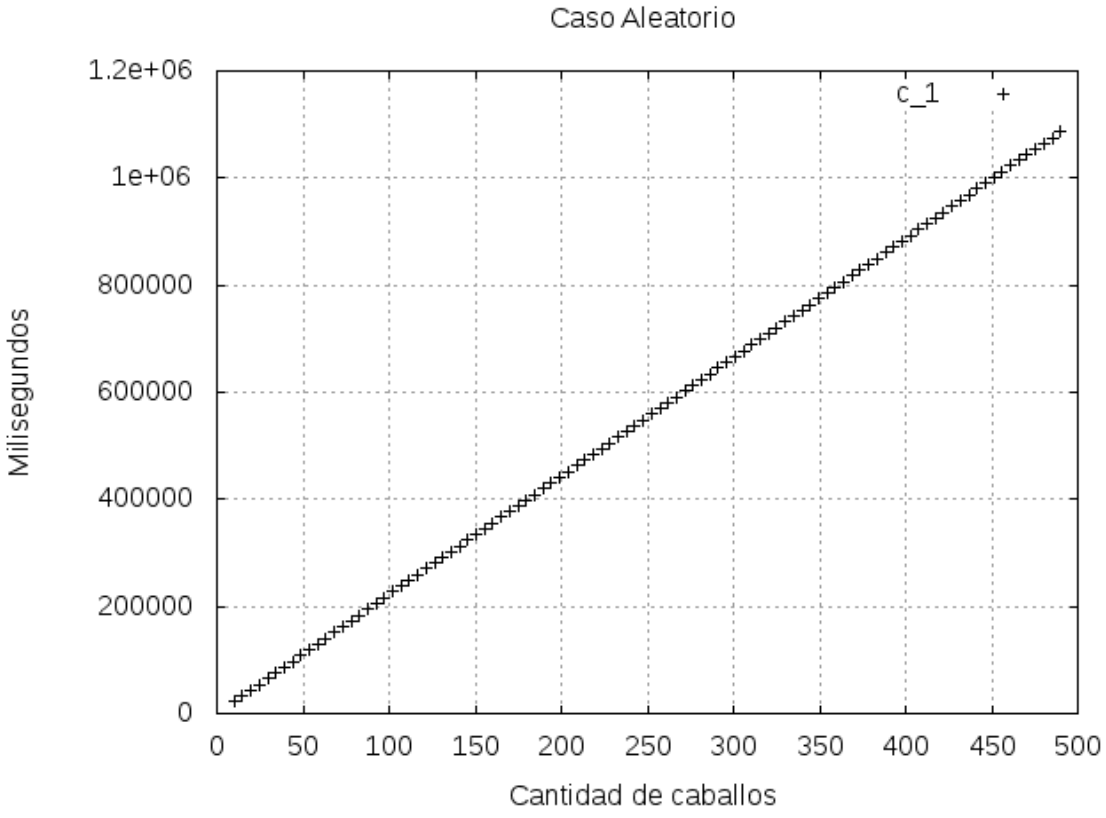
Para la experimentación de este ejercicio decidimos tomar los tiempos de la misma forma que en el ejercicio 1, es decir, utilizando la libreria *chrono*. Ambos experimentos fueron llevados a cabo corriendo nuestro algoritmo una cantidad fija de veces, y luego quedandonos con el menor tiempo que tardó nuestro algoritmo.

Para poder generar instancias aleatorias, lo que hicimos fue crear un generador, al cual le pasamos como parametro la cantidad de casilleros de nuestro tablero, la cantidad de caballos y una semilla. Este generador se encarga de crear una instancia aleatoria válida, utilizando como semilla inicial la que le pasamos como parametro. El generador crea instancias tanto solubles como no solubles, pero siempre válidas.
Decidimos utilizar un generador aleatorio, ya que nuestro algoritmo no tiene un mejor o peor caso, como fue explicado anteriormente.

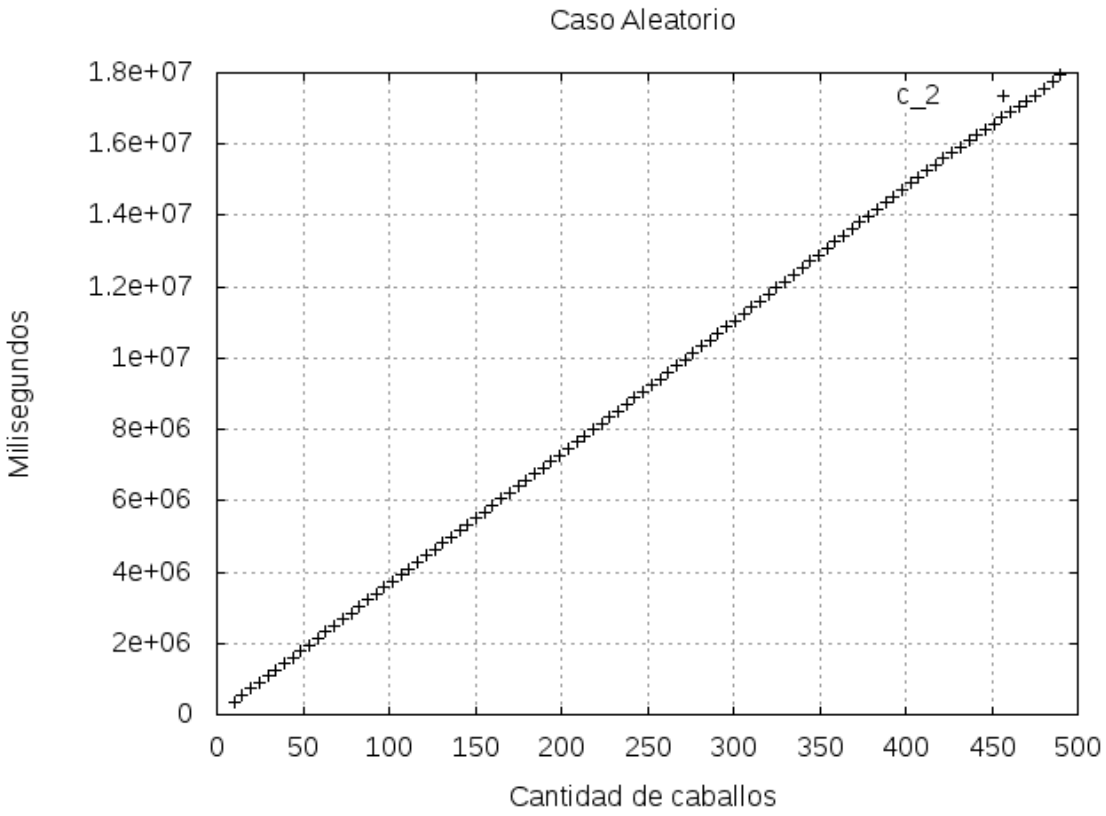
Llevamos a cabo dos experimentos, el primero consistió en fijar la cantidad de casilleros del tablero e ir aumentando la cantidad de caballos sobre el tablero.
Para el segundo experimento decidimos realizar lo inverso, es decir, fijar la cantidad de caballos e ir aumentando la cantidad de casilleros del tablero. A continuación se detallan los experimentos.

2.6.1. Experimento 1

Para este experimento fijamos la cantidad de casilleros en 50 y luego en 200, comenzamos nuestro algoritmo con 10 caballos y fuimos aumentando de a 15 hasta llegar a los 500. Se corrio el algoritmo 25 veces y se utilizó al numero 15 como semilla para las instancias. El objetivo de este experimento fue comprobar que la complejidad de nuestro algoritmo es lineal con respecto a la cantidad de caballos, sin importar la cantidad de casilleros del tablero, es por esto que el n fue fijado. Los resultados fueron los siguientes:



instancia con 50 casilleros fijos

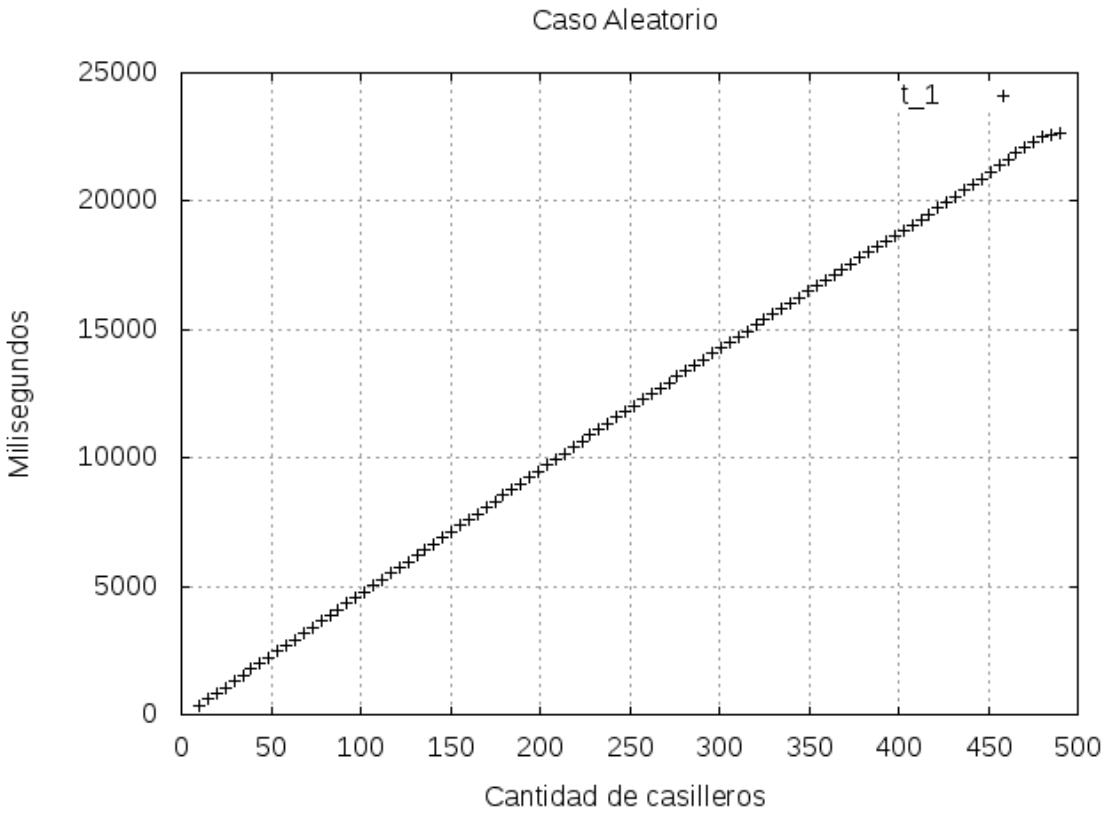


instancia con 200 casilleros fijos

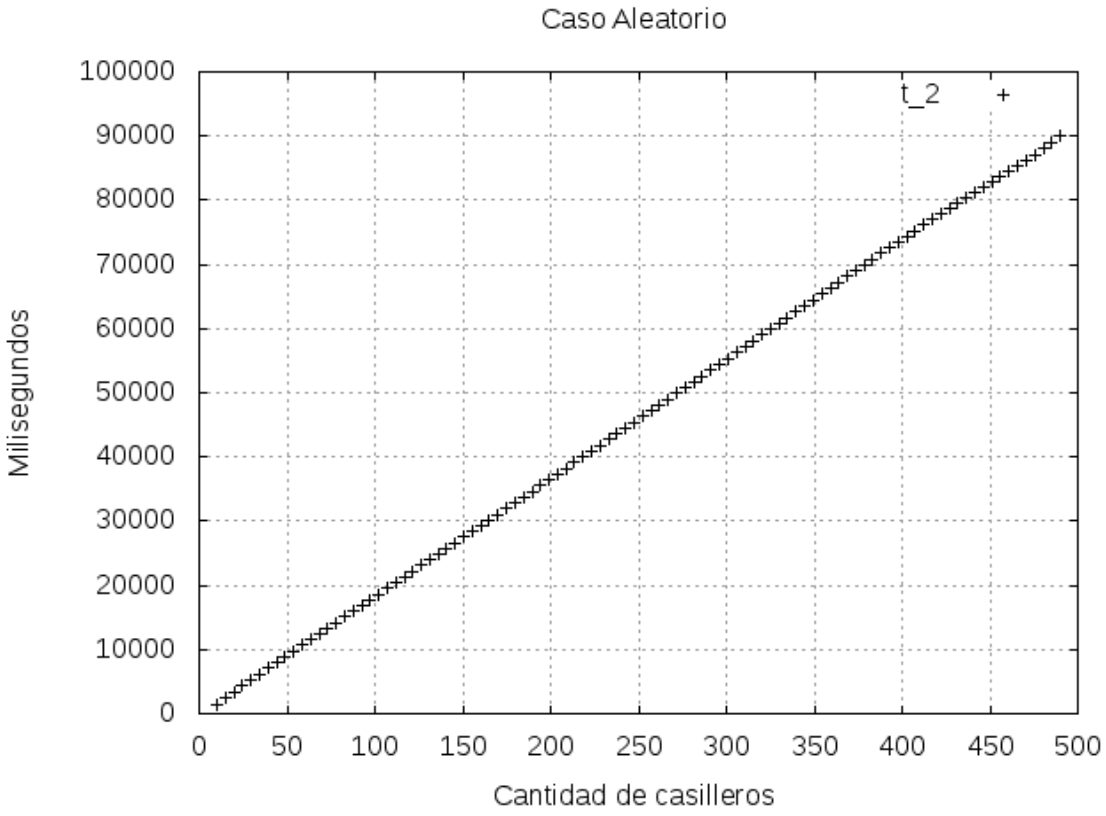
Se puede observar en los resultados que la función resultante es lineal. Esto se debe a que nuestro algoritmo es de orden $k * n^2$, con k caballos y n casilleros, por lo tanto si fijamos el n (ya sea en 50 o 200) la función del tiempo será lineal con respecto a k.
Como nuestro algoritmo es de orden $k * n^2$, es de esperar que el segundo experimento tarde más que el primero ya que el n fijado es el cuadruple, pero aún asi sigue siendo lineal con respecto a k.

2.6.2. Experimento 2

Para este experimento, decidimos fijar la cantidad de caballos en 50 y luego en 200, para poder analizar que sucede al aumentar la cantidad de casilleros del tablero. Lo que hicimos fue comenzar con una instancia de 10 casilleros, e ir aumentandolos de a 15 hasta llegar a 500. Cada instancia fue corrida 25 veces, utilizando el método previamente explicado para medir tiempos. La semilla utilizada fue el número 15.
En este caso, el objetivo del experimento fue el de comprobar empíricamente que la complejidad de nuestro algoritmo era cuadrática con respecto a la cantidad de casilleros del tablero, es por esto que fijamos la cantidad de caballos. Los resultados fueron los siguientes :



instancia con 50 caballos fijos



instancia con 200 caballos fijos

Para graficar las funciones de este experimento, lo que hicimos fue dividirlas por la función $f(n) = n$, esto generó que las funciones sean de tipo lineal, ya que como fue explicado anteriormente, nuestro algoritmo es de orden $k * n^2$. Por lo tanto si dividimos la función de tiempo de nuestro algo ritmo por n, el grafico de la función resultante será de tipo lineal, como se observa en ambas figuras. Esto no hizo falta en el experimento anterior, ya que al tener fija la cantidad de casilleros, lo unico que variaba era el k, y como aumentaba de forma lineal el gráfico resultante también lo era.
Al estar fijada la cantidad de caballos, ya sea en 50 o 200, al aumentar la cantidad de casilleros

nuestra función crece de forma cuadrática con respecto a n (en el gráfico se ve lineal por lo explicado previamente).
Con respecto a la comparación entre ambos gráficos podemos decir que el que tiene fijado un k mayor tarda más tiempo, ya que nuestro algoritmo es lineal con respecto a k, tal como lo explicamos previamente.
Estos resultados obtenidos confirman que la complejidad de nuestro algoritmo es de orden $k * n^2$

2.7. Anexo Reentrega

La nueva versión de este juego, utiliza alfiles en lugar de caballos. Es decir, los movimientos de las piezas cambian y sólo se permiten los movimientos permitidos a los alfiles. ¿Qué impacto tiene este cambio en su algoritmo?

En primera instancia, la idea general del algoritmo se mantiene. Ésto es, se deben calcular los movimientos para cada alfil, y luego sumar las matrices (las que contienen la cantidad mínima de movimientos para acceder a cada casillero) y sacar el mínimo.

Lo que sí cambiaría es la forma en la que se calculan los movientos, es decir el algoritmo *calcular_movimientos*. No es necesario implementar un BFS para recorrer los distintos *niveles* de movientos, pues al ser un alfil, partiendo desde un color (blanco o negro) se puede acceder a cualquier posición (del mismo color) con un máximo de 2 movimientos.

Con 0 movimientos se accede a la posición inicial, con 1 movimiento se accede a cualquier casillero comprendido en alguna de las dos diagonales que se proyectan desde la posición inicial. Y con 2 movimientos se accede a cualquier posición que se desprenda (en forma de dos diagonales) de las diagonales proyectadas inicialmente.

La complejidad del algoritmo sería $O(k * n)$, pues tendría que calcular los movimientos para los k alfiles, y me costaría $O(n)$ hacer dicha tarea para cada uno.

Y decimos que es lineal calcular los movimientos porque es simplemente escribir un 0 en la posición inicial, un 1 en las dos diagonales proyectadas desde el inicio, y luego un 2 en todos los demás casilleros que sean del mismo color (blanco, o negro) que el casillero inicial.

3. Problema 3

3.1. Descripción del Problema

El problema "La Comunidad del Anillo" presenta una situación donde se requiere armar una red cableada con varios requerimientos:

- Cada enlace tiene un costo asociado
- El costo acumulado del uso de la red debe ser mínimo
- Debe existir una topologia de anillo en la red (serán los servidores de la misma)
- Todos los equipos de la red deben tener un camino de conexiones que llege al anillo

Será necesario cuando se devuelve el resultado especificar cuantos y cuales enlaces fueron utilizados para el anillo y cuantos y cuales fueron los enlaces que quedaron para conectar el resto de la red, siendo una solución valida que todos los enlaces de la red formen el anillo de servidores.

Aparte de revisar si las instancias tienen solución, el algoritmo deberá retornar la o una de las soluciones óptimas siendo, como se indicó previamente, una solucion óptima aquella que minimize el costo acumulado de uso de toda la red.
De no existir una forma de conectar todos los equipos en una única red con los enlaces disponibles o en caso de que no exista una forma de armar un anillo con los mismos el algoritmo retornará un "no" indicando que la instancia no tiene solución.

3.2. Ideas desarrolladas

Hablaremos a continuación de los fundamentos teóricos que usamos como base para presentar la serie de pasos con los que resolvimos el problema.

Vamos a aprovechar varias propiedades sobre grafos:

- Sabemos que los árboles no tienen ningún ciclo y que estan conformados por componentes conexas
- Se demostró en la teórica que a cualquier árbol al que se le agrege una arista conectada a vértices del árbol, genera si ó si un ciclo en su estructura
- Finalmente sabemos que un Arbol Generador Mínimo sera aquél sub-grafo árbol que cumpla que tiene el costo total de las aristas optimizado y es árbol generador.

Con estas propiedades ideamos una estrategia basándonos en el algoritmo de árboles generadores mínimos de Primm.

Para explicar mejor, vamos a ver la red de computadoras como grafos, donde la computadoras son nodos y los enlaces son las aristas.

Comenzamos armando una matriz, a la cual voy a llamar la matriz de enlaces. Es esta matriz la que contiene la información primordial para resolver este problema. En principio la matriz iba a solamente ser matriz de enteros, donde iba a guardar los costos de las aristas. Pero planteamos el algoritmo de esa forma, necesitabamos otra estructura donde poder obtener la informacion si en enlace había sido utilizado o no. La estructura que habíamos elegido eran listas de enlaces, donde los enlaces son tuplas<entero,entero>. El inconveniente que nos generaba tener armadas así las estructuras, fue con la complejidad ya que nos quedaba $O(n^3)$ y necesitamos una complejidad estrictamente menor.

Es entonces cuando la matriz de enlaces paso a tener más información. Dejo de contener enteros para ahora contener una estructura que llamamos ïnfo".

Info es una estructura que tiene:

- Costo: entero que guarda el costo de la arista. En las posiciones donde no existe arista se coloca el valor -1.
- Arista: tupla<nodo,nodo>, donde nodo es representado por un entero.
- Usado: bool que indica si la arista se encuentra formando parte del anillo o de la red.
- En_anillo: bool que indica si la arista se encuentra formando parte del anillo.

Una vez que armamos la matriz, pasamos a nuestro algoritmo principal *armar_AGM*. Esta función recibe la matriz, el maximo costo de las aristas y la cantidad de nodos. Comienza armando un AGM en base al algoritmo de Prim. Luego marca en la matriz a los enlaces usados en el AGM como usados. Para crear el AGM, este algoritmo necesito crear un vector, *padre_de*, de tamaño cantidad de nodos. Este vector lo guarda en su posición padre_de[nodo a] es el padre del nodo a.

Finalmente llama a la siguiente función principal, *armar_resultado*, a la que le pasa la matriz, el vector padre_de, la cantidad de nodos y el maximo de los costos de las arista.

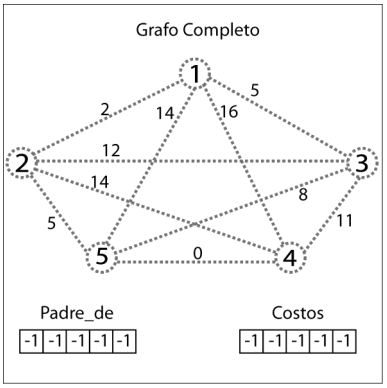
El resultado en este algortimo es una estructura que se compone de:

- Cant_anillo: cantidad de aristas que componen el anillo.
- Cant_red: cantidad de aristas que componen la red.
- Anillo: secuencia de aristas que conforman el anillo.
- Red: secuencia de aristas que conforman la red.
- C: costo de todas las aristas utilizadas en la red y el anillo.

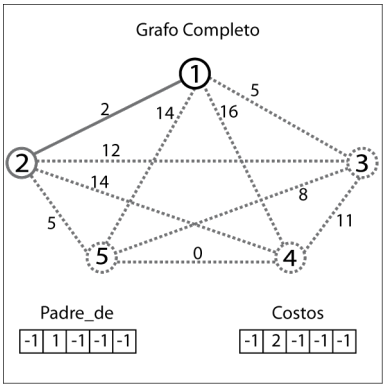
Esta función lo que hace es encangarse de armar el anillo y la red. Para armar el anillo posee funciones auxiliares que primero busca, entre las aristas que quedaron sin utilizar en el AGM, la de menor costo. Esa va a ser la arista por donde vamos a comenzar el anillo. Luego crea empezando desde los dos nodos de la arista que comienza el anillo, las dos mitades del anillo, recorriendo desde cada nodo del comienzo del anillo hasta unirse formando el ciclo.
La red la construye consultando la matriz que arista fue utilizada y no pertenece al anillo.
Luego consulta las longitudes de Red y Anillo, para completar los datos Cant_anillo y Cant_red. Y por úlitmo consulta la matriz preguntando por los costos de las aristas que fueron utlizadas para completar el dato C.

Ejemplo con ilustraciones:

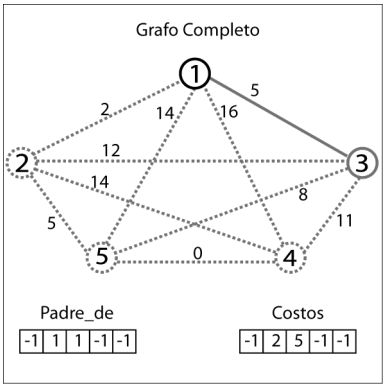
Instancia:
Cantidad de Nodos: 5 , Cantidad de aristas15
Enlace: 1-2 Costo: 2
Enlace: 1-3 Costo: 5
Enlace: 1-4 Costo: 16
Enlace: 1-5 Costo: 14
Enlace: 2-3 Costo: 12
Enlace: 2-4 Costo: 14
Enlace: 2-5 Costo: 5
Enlace: 3-4 Costo: 11
Enlace: 3-5 Costo: 8
Enlace: 4-5 Costo: 0



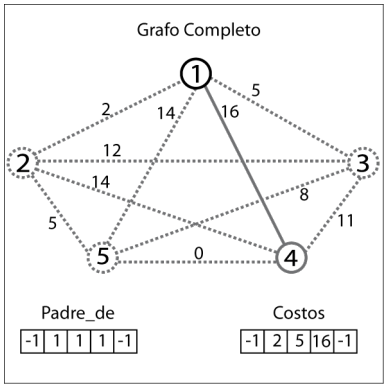
(c) Al comenzar arma dos vectores, padre_de y costos



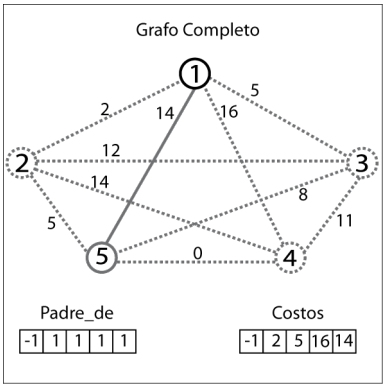
(d) Empieza a evaluar desde el nodo 1. Marca como agregado al nodo 1. Evalua el enlace 1-2. Actualiza los vectores padre_de(2-1)=1 y costos(2-1)=2



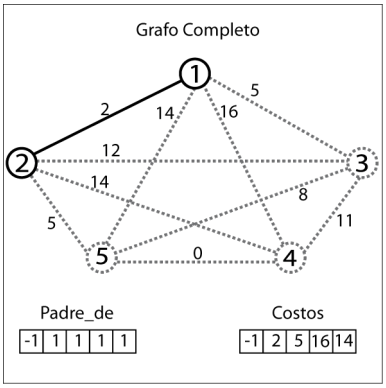
(e) Evalua el enlace 1-3.Actualiza los vectores padre_de(3-1)=1 y costos(3-1)=5



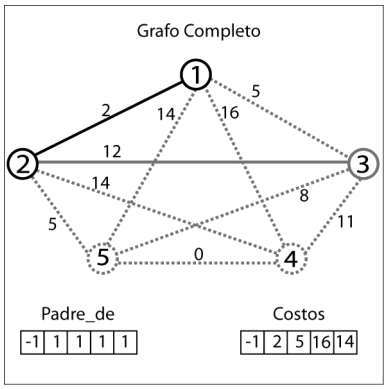
(f) Evalua el enlace 1-4.Actualiza los vectores padre_de(4-1)=1 y costos(4-1)=16



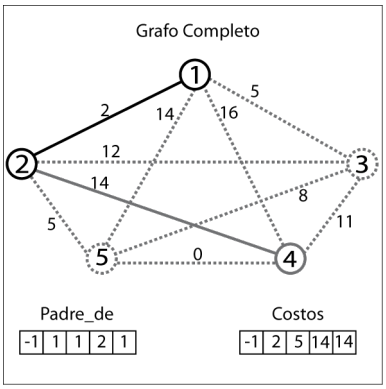
(g) Evalua el enlace 1-5.Actualiza los vectores padre_de(5-1)=1 y costos(5-1)=14



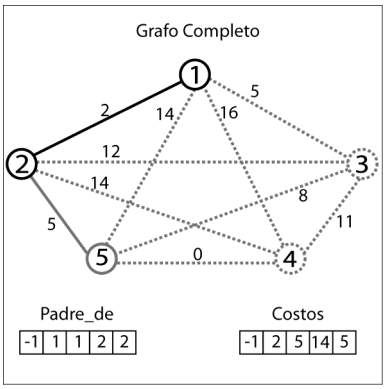
(h) Recorrer el vector costos buscando el menor valor de cuyo nodo que no se encuentre agregado. En este caso ese valor es un 2 y corresponde al nodo dos. Luego se fija en el vector padre_de, que padre le corresponde al nodo 2, y es el nodo 1. Luego arma el enlace 1-2 que se marca como usado, y el siguiente nodo para seguir analizando es el 2.



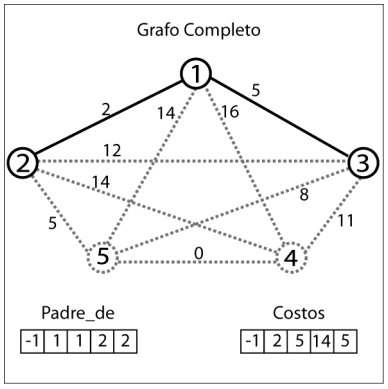
(i) Continua a evaluando desde el nodo 2. Marca como agregado al nodo 2. Evalua el enlace 2-3. Como el valor en costos(3-1) es menor al costo del enlace 2-3 los vectores no se modifican



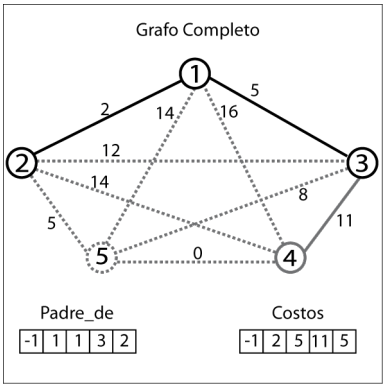
(j) Evalua el enlace 2-4.Actualiza los vectores padre_de(4-1)=2 y costos(4-1)=14



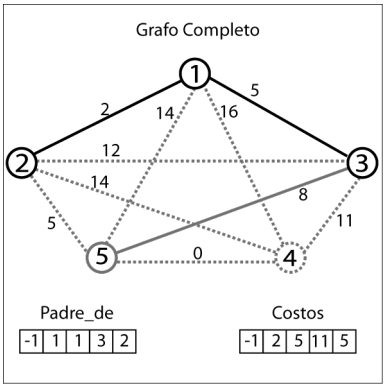
(k) Evalua el enlace 2-5.Actualiza los vectores padre_de(5-1)=2 y costos(5-1)=5



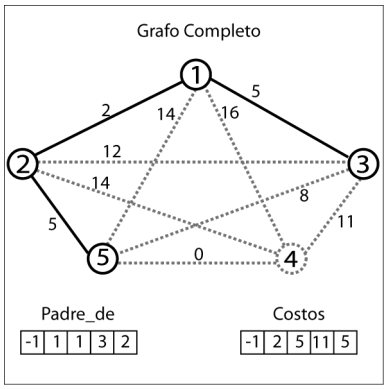
(l) Nuevamente recorre el vector costos buscando el menor valor. Y en este caso corresponde al enlace 1-3 que es marcado como usado.



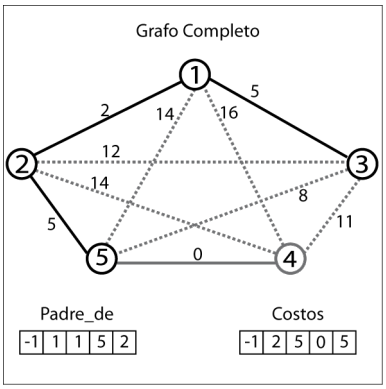
(m) Continua a evaluando desde el nodo 3. Marca como agregado al nodo 3. Evalua el enlace 3-4. Actualiza los vectores padre_de(4-1)=3 y costos(4-1)=11



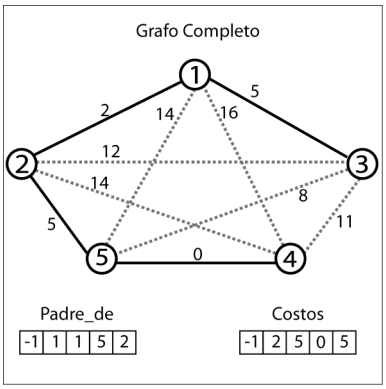
(n) Evalua el enlace 3-5. Como el valor en costos(5-1) es menor al costo del enlace 3-5 los vectores no se modifican



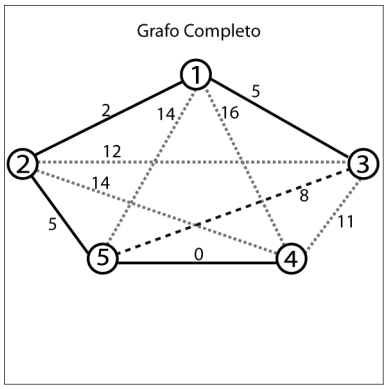
(ñ) Recorre el vector costos buscando el menor valor. Y en este caso corresponde al enlace 2-5 que es marcado como usado.



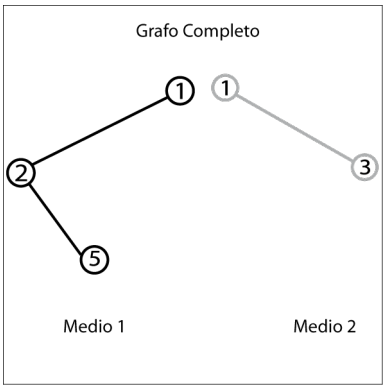
(o) Continua a evaluando desde el nodo 5. Marca como agregado al nodo 5. Evalua el enlace 5-4. Actualiza los vectores padre_de(4-1)=5 y costos(4-1)=0



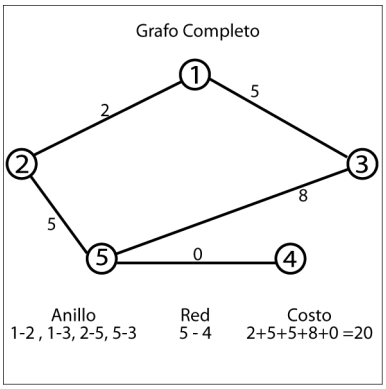
(p) Recorre el vector costos buscando el menor valor. Y en este caso corresponde al enlace 5-4 que es marcado como usado. Aquí ya tenemos armado nuetro AGM.



(q) Se busca el menor enlace no utilizado, 5-3.



(r) Partiendo de los dos nodos 5 y 3 que forman el enlace encontrado en la imagen anterior, se van a armar dos mitades del anillo. Luego se van a unir por el nodo que tiene en comun, el 1. Y el anillo se va a cerrar con la arista 5-3



(s) Backbone

3.3. Pseudocódigo

Se acortaron los nombres por motivos de diseño.

men_no_ut corresponde a *menor_no_utilizado*.

MEN_NO_UT(**in/o** *ady*: $matriz < info >$, **in** *padre_de*: $vec < int >$, **in** *nodos*: *int*, **in** *max*: *int*)→ $< int, int >$

```
1  for (int i = 0; i < nodos ; i++)
2    for (int j = 0; j < nodos ; j++)
3      if (enlace (i,j) no es usado, no tiene costo infinito,
4         Su costo es el mejor revisado y no esta enlazado dos nodos ya enlazados)
5         then Me guardo el nuevo enlace encontrado
6         else  Sigo buscando
7              endif
8      endfor
9    endfor
10 Devuelvo el enlace guardado
```

ARMAR_MEDIO(**in** *comienzo*: *int*, **in** *padre*: $vec < int >$)→ $list(int)$

```
1  while No llegue al nodo raiz (sin padre)
2    Agrego al resultado el nodo revisado
3    endwhile
4  Retorno la lista de nodos revisados
```

ARMAR_ANILLO(**inout** *uno*: $list(int)$, **inout** *dos*: $list(int)$, **inout** *ady*: $matriz < info >$)→ $list(enlace)$

```
1  while no termine de recorrer la lista uno
2    Busco el elemento de la lista uno a revisar en la lista dos
3    Si el elemento existe, ese va a ser el nodo final del anillo, break al ciclo
4    endwhile
5  Utilizando las dos listas y el primer nodo que tiene en comun armo el anillo
6  Retorno el anillo
```

ARMAR_RESULTADO(**in** *padre_de*: $vec < int >$, **in** *cant*: *int*, **inout** *ady*: $matriz < info >$, **int** *max*: *int*)→ *res*

```
1  Comienzo con un resultado vacío
2  Busco el enlace menor no utilizado en el AGM, que será uno de los enlaces del anillo
3  Si no encuentro nigun enlace que cumpla lo requerido finalizo devolviendo un resultado vacío
4  Armo desde las dos puntas del enlace encontrado el camino hacia la raiz con la función armar_medio
5  Armo el anillo con los dos caminos creados
6  Armo la red con los enlaces usados pero que no se encuentran en el anillo
7  Calculo el costo del armado de la red
8  Retorno el resultado completo
```

ARMAR_AGM(**inout** *ady*: $matriz < info >$, **in** *cant*: *int*, **in** *max*: *int*)→ *resultado*

```
1  Se crea el vector de int padre_de con longitud cant (cantidad de nodos del problema) y en todas sus
2  posiciones guarda un -1
3  Se crea el vector de int costos con longitud cant y en todas sus posiciones guarda max+1
4  (el mayor costo de los enlaces más uno)
5  Se crea el vector de bool agregado con longitud cant y en todas sus posiciones guarda false
6  int u = 0
7  for (int i = 0; i < cant -1 ; i++)
8    agregado[u] = true
9    for (int v = 0; i < cant ; i++)
10     if v no pertene al vector agregado y si el enlace u-v existe y si el costo del enlace u-v es menor
11        al que se encuentra en costos[v]
12        then
13             padre_de[v] = u
14             costos[v]= costo del enlace u-v
15             endif
16     endfor
17   u = el nodo con el menor costo guardado en el vector costo
```

```
18     Se marca el enalce v-u como usado
19     endfor
20   if Existe camino entre que una todos los nodos
21     then
22           Devuelve resultado = armar_resultado(padre_de,cant,ady,max)
23     else  El problema no se puede resolver
24           endif
```

3.4. Análisis de Correctitud

Algunas puntos a tener en cuenta[4]:

1. **Infinito** será el mayor costo de todos los enlaces +1
2. Sea **T** árbol, existe un único camino simple entre todo par de nodos
3. G es un grafo sin circuitos simples, pero si se agrega cualquier arista e a G resulta un grafo con exactamente un circuito simple, y ese circuito contiene a e
4. En un grafo G, la concatenación de dos caminos distintos entre un par de vértices contiene un circuito simple
5. Se define el peso de un grafo como la función $c : grafo \rightarrow int$ con C(G) = $\sum_{(u,v)\in X_G}$ l(u,v)

Como enunciamos en Ideas Desarrolladas, nuestra implementación utiliza al algoritmo de Prim para generar un Arbol Generador Mínimo como base y se trabaja sobre eso.

Para que Prim devuelva un AGM correcto se debe cumplir que la entrada sea un grafo conexo, de lo contrario al final de la ejecución (por como lo implementamos) quedarán nodos con costo **Infinito** [5]. Es por eso que luego de que se corre la implementacion de Prim se realiza una verificación sobre los costos de los nodos, de no pasar esta verificación el algoritmo finaliza.

Una vez que el AGM esta generado, y completo, se procede a generar el anillo solicitado por el enunciado.
Para eso utilizamos la función *menor_no_utilizado* que de todos los enlaces no utilizados para la creacion del AGM devuelve el que tiene mínimo costo, enlaza dos computadoras del AGM y no enlaza a dos que ya se encontraban enlazadas.

Ahora, como un árbol es un grafo sin circuitos simples, si agregamos el nuevo enlaze **e**, por item 3, se forma un circuito simple que contiene a **e**. Para encontrar el resto del anillo desde ambas computadoras enlazadas por **e** generamos el camino hasta la raiz quedandonos C_1 y C_2 caminos simples, entonces por item 4 la concatenacion de C_1 y $C_2 \cup \mathbf{e}$ contiene un circuito simple, osea el anillo que estamos buscando. Sabemos que C_1 y $C_2 \cup \mathbf{e}$ forman un circuito, por lo que si recorremos C_1 y $C_2 \cup \mathbf{e}$ desde sus puntas en algun momento se llegara a la misma computadora, y esos enlaces recorridos formaran el anillo solicitado.
Esto lo realizamos con las funciones *armar_medio* y *armar_anillo*, la primera genera C_1 y C_2 mientras que la segunda encuentra la intersección entre ambos para así generar el anillo que se retornará.

Sabemos que el costo del AGM generado es mínimo, y por el principio de optimalidad de Bellman el costo del AGM + **e** va a ser menor o igual al costo del AGM + cualquier otro enlace no utilizado que es lo que solicita el enunciado.

Notese que como el AGM no es único, **e** puede ser que tampoco lo sea y es por estos dos puntos que el ejercicio permite más de una solución.

3.5. Análisis de Complejidad

Nuestra función principal en este ejercicio es el algoritmo de Prim, que luego de construir el AGM llama a la funcion *armar_resultado*, que a su vez tiene dentro las llamadas a las siguiente funciones: *menor_no_utilizado*, *armar_medio*, *armar_anillo*. Iremos analizando su complejidad algoritmica para conocer la complejidad de nuestro algoritmo.

Llamaremos n a la cantidad de computadoras (nodos) y m a la cantidad de enlaces (aristas) entre ellas.

[4] los puntos 2,3 y 4 fueron demostrados en la teórica
[5] en otras implementaciones de Prim el algoritmo continua ciclando hasta que todos los nodos son unidos al AGM

3.5.1. MENOR_NO_UTILIZADO

Una vez que tenemos armado nuestro AGM, para poder cerrar el anillo lo que necesitamos es la arista con el costo mínimo de las que quedaron fuera del AGM. Para buscar esta arista, vamos a recorrer la matriz de enlaces.
El costo de este algoritmo esta dado por este recorrido, $O(n^2)$.

3.5.2. ARMAR_MEDIO

Esta función recibe una nodo y un vector con nodos padres, y lo que devuelve es el camino comenzando por dicho nodo hasta el nodo raiz del AGM. Vamos a ir recorreindo en vector de padres hasta llegar al nodo -1, que es el que representa al nodo raiz. Esta algoritmo es $O(n)$, ya que su complejidad esta dada por el recorrido del vector con nodos padres.

3.5.3. ARMAR_ANILLO

Nuestro anillo esta representado por una lista de aristas. Para poder crear esta lista, el algoritmo recibe dos vectores de nodos, que son los nodos que componen al anillo. Estos vectores solo comparten un nodo en común, que llamaremos *enlace_unión*. Para buscar este nodo, el algoritmo recorre primero el primer vector, buscando cada uno de sus nodos en el segundo vector. Hasta aca ya tenemos una complejidad $O(n^2)$, ya que por cada nodo en el primer vector, va a recorrer en el peor caso toda el segundo vector. Y además en el peor caso estos dos vectores tienen longitud n.
Luego, como hasta ahora solo tenemos nodos, tenemos que crear los enlaces en de nuestro anillo. Para eso recorreremos los dos vecotres creando los enlacen entre sus nodos. Los enlaces se van a crear entre un nodo y su siguente. Recorrer estas listas tiene una complejidad $O(n)$, por lo que nuestro algoritmo ARMAR_ANILLO tiene una complejidad $O(n^2) + O(n) = O(n^2)$.

3.5.4. ARMAR_RESULTADO

El resultado esta compuesto por un anillo, la cantidad de aristas del anillos, la red(los nodos fuera del anillo), la cantidad de nodos que forman parte de la red, y el costo total de las aristas utilizadas. Para armar este resultado, el algoritmo lo primero que va a hacer es obetener el comienzo del anillo, usando a la funcion MENOR_NO_UTILIZADO. Luego creados los dos medios del anillo con la función ARMAR_MEDIO. Y después crea el anillo usando la funcion ARMAR_ANILLO a la que le pasa las dos mitades. Hasta ahora tenemos la siguiente complejidad: $O(n2)+O(n)+O(n)+ O(n^2) = O(n^2)$.
Luego totamos la longitud de la lista anillo, para conocer la cantidad de aristas que lo conforman. Esto sería $O(1)$, ya que estamos utilizando C++11.
Por último, calcula el costo de las aristas utilizadas, y arma la lista de aristas que comforman la red recorriendo la matriz de enlaces y consultando si fueron utilizadas y si son parte del anillo o no. La complejidad de este recorrido es $O(n^2)$.
La complejidad de ARMAR_RESULTADO que obtenemos es $O(n^2)$.

3.5.5. ARMAR_AGM

Lo primero que hace este algoritmo es construir 3 vectores. El vector padre_de, el vector costos y el vector agregado. Los tres vectores se crean con longotud n, que es la cantidad de nodos del problema, así que hasta ahora tenemos una complejidad de $O(n)$.
El algoritmo continúa con dos cilcios. El primero se itera n veces, recorreindo todos los nodos. Por cada nodo, va a primero comenzar otro ciclo, que completa el vector y el vector padre_de, y luego va a seleccionar la arista a utilizar recorriendo el vector costos para buscar el menor. La operacion de completar los vectores costos y padre_de, y la operacion de buscar la arista a utilizar tienen una complejidad $O(n + n) = O(n)$. Estas operaciones al estar dentro del ciclo, se repiten n veces, lo que nos da la complejidad $O(n^2)$. Hasta aca nuestro algoritmo tiene una complejidad de
$O(n) + O(n^2) = O(n^2)$.
Finalmente, el algoritmo llama a la función ARMAR_RESULTADO, que ya vimos que su complejidad es $O(n^2)$, lo que nos indica que la complejidad de ARMAR_AGM es $O(n^2) + O(n^2) = O(n^2)$.

Conclusión: Podemos decir entonces que la complejidad de nuestro algoritmo, que resuelve el problema *La comunidad del anillo*, es $O(n^2)$

3.6. Experimentación

Para la experimentación de este ejercicio decidimos tomar los tiempos de la misma forma que venimos haciendo tanto en el ejercicio 1 como en el 2, utilizando la libreria *chrono*. Ambos experimentos fueron

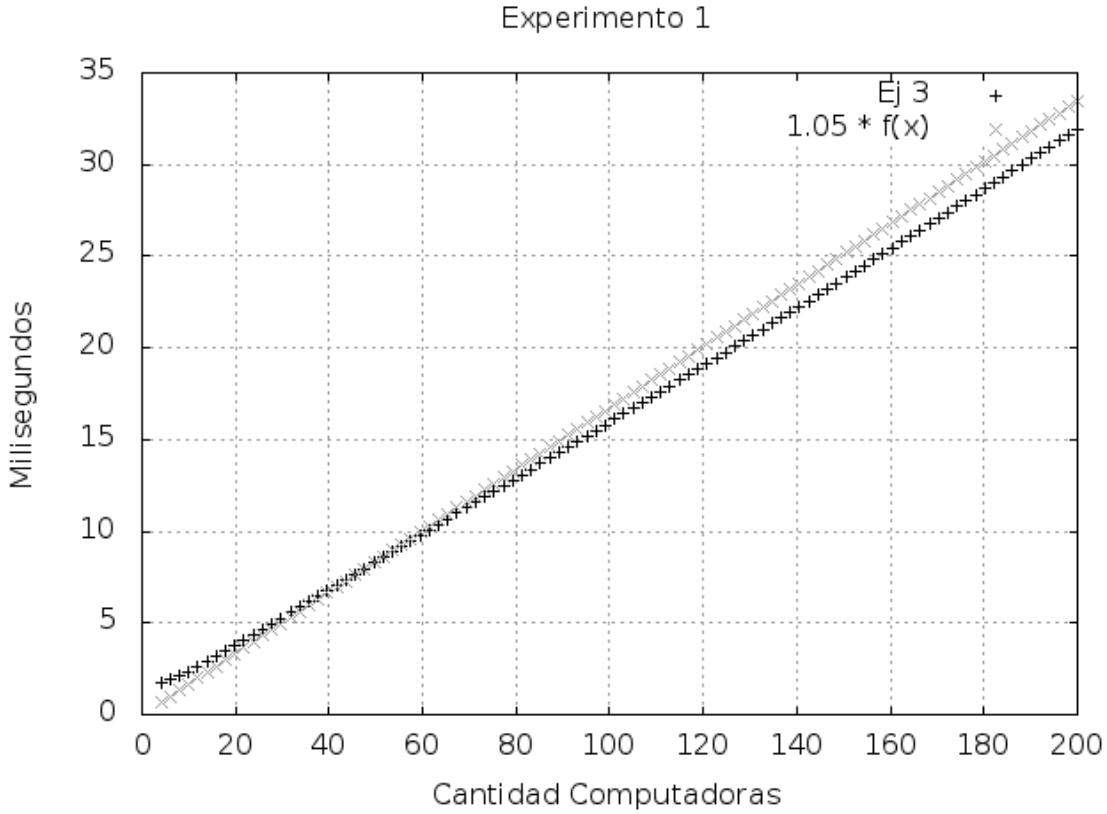
llevados a cabo corriendo nuestro algoritmo una cantidad fija de veces, y luego quedandonos con el menor tiempo que tardó nuestro algoritmo.

Para poder generar instancias aleatorias, lo que hicimos fue crear un generador, al cual le pasamos como parametro la cantidad de computadoras y la cantidad de enlaces. Este generador se encarga de crear una instancia aleatoria válida y soluble.
Hay que tener encuenta que la cantidad de enlaces depende de la cantidad de compuadoras, para que las instancias sean solubles. De la cantidad de computadoras, vamos a obtener un rango en el que se puede mover la cantidad de enlaces. Para explicar mejor, vamos a ver la red de computadoras como grafos, donde la computadoras son nodos y los enlaces son las aristas.
El grafo más pequeño, en cantidad de aristas, que se puede tener, es un grafo conexo. No podríamos armar un anillo ni la red si existiesen nodos en el grafo a los cuales no se pueden llegar a ellos por un camino. Entonces llamando n a la cantidad de nodos y m a la cantidad de aristas, la cota inferior que obtenemos es $n - 1 \leq m$. El grafo más grande es el grafo completo. De aquí obtenemos que la cota superior es $m \leq \sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$.

La complejidad de nuestro algoritmo solo depende de la cantidad de computadoras y no de los enlaces entre ellas. Por lo que en este caso, el peor caso siempre va a ser el Caso Promedio. Para ver si realmente solo es afectado por la cantidad de nodos llevamos a cabo dos experimentos, el primero consistió generar grafos completos y el segundo consitió en generar grafos conexos pero no completos A continuación se detallan los experimentos.

3.6.1. Experimento 1

Para este experimento creamos un scrip al cual le pasamos la cantidad de computadoras, el tope de la cantidad de computadoras, en cuanto vamos incrementando esta cantidad, y por último la cantidad de veces que vamos a iterar el algoritmo para quedarnos con su minimo.
Como ya aclaramos la cantidad de enlaces depende de la cantidad de computadoras. Para estos casos el script es el que va a calcular esta cantidad. Al ser Completos el script usa $m = \frac{n \times (n+1)}{2}$ (n cantidad de computadores, m cantidad de enlaces). Los resultados fueron los siguientes:

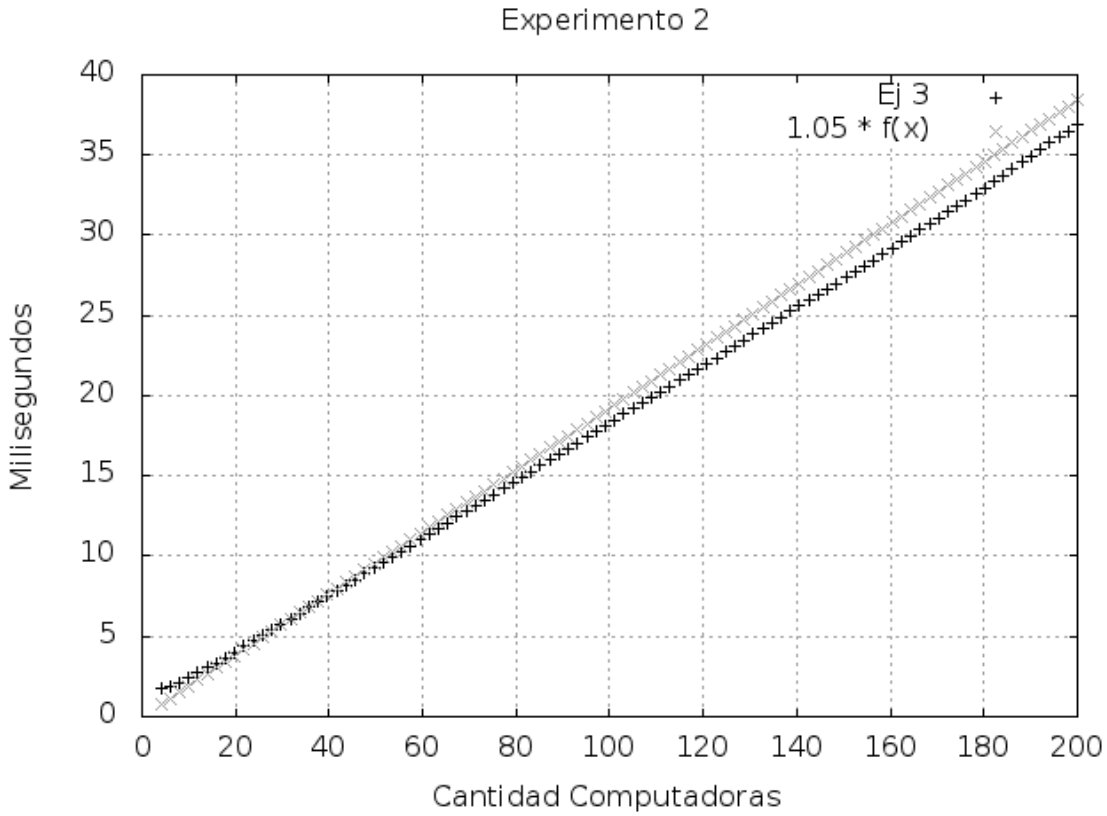


Como nuestro algoritmo es de orden n^2, es de esperar que el segundo experimento se encuentre muy aproximado a este.

3.6.2. Experimento 2

Para este experimento, nuevamente es el script el que se encarga de calcular la cantidad de enlaces, en base a la cantidad de computadoras que le pasamos. Al ser Conexos no Completos el script usa $m = n$

(n cantidad de computadores, m cantidad de enlaces). Usa $m = n$ y no $m = n - 1$, ya que para que exista un anillo necesitamos un enlace más. Sino tendríamos solamente un AGM. En este caso, el objetivo del experimento fue el de comprobar empiricamente que la complejidad de nuestro algoritmo era cuadrática con respecto a la cantidad de computadoras, sin importar la cantidad de sus enlaces.



Como podemos ver, los gráficos son muy aproximados. Por lo que podemos decir, que en nuestro algorito su complejidad solo se ve afectada por la cantidad de computadoras.

Aclaración: Para graficar las funciones de este experimento, lo que hicimos fue dividirlas por la función f(n) = n, esto generó que las funciones sean de tipo lineal, ya que como fue explicado anteriormente, nuestro algoritmo es de orden n^2. Por lo tanto si dividimos la función de tiempo de nuestro algo ritmo por n, el gráfico de la función resultante será de tipo lineal, como se observa en ambas figuras.

3.7. Anexo Reentrega

Si una vez ya construida nuestro backbone el cliente modifica un único enlace , modificando su costo, reutilizariamos nuestro algoritmo de la siguiente forma:
Como datos de entrada tendríamos ahora el backbone ya armado y el enlace a modificar con el nuevo costo.

Lo primero que debería evaluar es si el costo es mayor o menor al que tenía anteriormente. Si es menor simplemente cambia el costo de este enlace y el backcone sigue armado como se encontraba anteriormente. Ahora, si el costo sube, tiene que evaluar si el enlace que se va a modificar forma parte del anillo o de la red.

Si el enlace forma parte del anillo, el código debe utiliar la función *menor_no_utilizado* y comparar el costo del enlace que encontró la función, con el costo del enlace que se modificó. Si el costo del enlace encontrado es mayor, se deja el backcone como está y sólo se le actualiza el nuevo costo. En caso contrario, se deja de usar en enlace con costo modificado y el nuevo enlace ahora forma parte del anillo esto podemos hacerlo por lo demostrado en la teórica [6], como el enlace que removemos pertenecia al anillo del AGM este continuaría siendo conexo. Se vuelve a utilizar la función *armar_anillo* para saber que enlaces forman parte del nuevo anillo. Para conocer los nuevos enlaces que forman parte de la red, recorremos la matriz de enlaces buscando los enlaces utilizados que no pertenecen al anillo.

Si el enlace que se modifica forma parte de la red, tenemos que comparar si nos conviene seguir utilizando ese enlace. Para esto vamos a dividir el backbone en dos componentes conexas. Una va a contener la parte de la red que quedo desvinculada del anillo y la otra la que contiene al anillo. Lo que

[6] Sea G = (V, X) un grafo conexo y e ∈ X. G − e es conexo si y solo si e pertenece a un circuito simple de G

el algoritmo debería hacer, es usar una función parecida a menor_mo_utilizado, que llamaremos *menor_no_utilizado_conexion_con_la_otra_cc*. La diferencia sería que ahora tenemos que buscar el menor enlace no utilizado que conecte ambas componentes conexas. En caso de que el enlace encontrado tenga un costo mayor al del enlace modificado, se deja el backcone como está y sólo se le actualiza el nuevo costo. En caso contrario, simplemente quitamos el enlace viejo de la red y colocamos el nuevo.

Respecto a la complejidad, teníamos que nuestro algoritmo es $O(n^2)$. Las modificaciones que se le realizarían al código sería, no utilizar los dos ciclos que arman el AGM, porque el backbone viene armado. Luego, si el enlace modificado ahora tiene un costo mayor y pertenece al anillo, la única modificacion que se haría es en la función *armar_resultado*, agregar una comparacion para saber si utilizar el enlace modificado o no. Porque si hay que cambiarlo seguiriamos utilizando las funciones que arman el anillo y la red. Hasta aca nuestra complejidad seguiría igual ya que *menor_no_utilizado* y *armar_anillo* nos dan la complejidad $O(n^2)$.
Por otra parte, si el enlace a modificar por un costo mayor forma parte de la red, tendríamos que crear dos vectores de computadoras (nodos), para saber a que componente conexa pertenece cada nodo. Luego deberíamos usar la función *menor_no_utilizado_conexion_con_la_otra_cc*, que por cada nodo en el vector de la CC que contiene la red, busco todas las adyacencias que tiene ese nodo con los nodos de la CC que contiene al anillo. Si alguno es menor que el enlace que fue modificado, entonces simplemente reemplaza en la secuencia de enlaces de la red, al enlace encontrado por el enlace modidificado. Como a la longitud de los vectores los podemos acotar por n, la complejidad de estos dos ciclos sería $O(n^2)$, por lo que finalmente nuestro algoritmo quedaría con una complejidad de $O(n^2)$.
Al quedar con la misma complejidad, concluímos que podríamos volver armar todo el backbone de cero si se modifica algún costo de un enlace.