_AGM _anillo

DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Grupo Theta

Integrante	LU	Correo electrónico
Rey, Martin	483/12	marto.rey2006@gmail.com
Marta, Cristian Gabriel	079/12	cristiangmarta@gmail.com
Lebedinsky, Alan	802/11	alanlebe@gmail.com
Coronel, Ezequiel	352/08	ecoronel@dc.uba.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

Contents

Introducción **3**

1 Problema 1: Plan de vuelo **4**
1.1 Introducción . . . 4
1.2 Desarrollo . . . 5
1.3 Complejidad . . . 9
1.4 Testing . . . 9
1.5 Experimentación . . . 10
1.6 Conclusión . . . 13

2 Problema 2: Caballos salvajes **14**
2.1 Introducción . . . 14
2.2 Desarrollo . . . 14
2.3 Demostración de correctitud . . . 15
2.4 Complejidad . . . 16
2.5 Testing . . . 17
2.6 Experimentación . . . 17
2.7 Conclusión . . . 20

3 Problema 3: La comunidad del anillo **21**
3.1 Introducción . . . 21
3.1.1 Ejemplo práctico: . . . 21
3.2 Idea General . . . 22
3.3 Desarrollo . . . 22
3.4 Demostración de correctitud . . . 23
3.5 Complejidad . . . 25
3.6 Testing . . . 26
3.6.1 Conexo sin ciclos . . . 26
3.6.2 No conexo con ciclos . . . 26
3.6.3 Cn . . . 26
3.6.4 Todos los enlaces con el mismo coste . . . 27
3.6.5 Caso general . . . 28
3.7 Experimentación . . . 28

A Código fuente: Problema 1 **33**

B Código fuente: Problema 2 **34**

C Código fuente: Problema 3 **36**

Introducción

En el presente trabajo práctico se intenta resolver mediante algoritmos ciertos problemas brindados por la cátedra. Se decidió implementar dichas soluciones usando el lenguaje *C++*[**?**] y también se utilizó algunas características de *C++11* como auto y la biblioteca chrono[**?**] para las mediciones de tiempos. Junto con las implementaciones, se adjunta el presente informe que especifica los detalles sobre el desarrollo de las soluciones, así como pruebas, cálculos de complejidad temporal, mediciones, etc.
Para calcular la complejidad de los algoritmos involucrados, utilizamos el **modelo de costos uniforme**, que es el solicitado por el enunciado.

1 Problema 1: Plan de vuelo

1.1 Introducción

Una empresa nacional en la cual la presidenta está involucrada, nos contrató para resolver un problema el cual creen les hará ganar mucho dinero **legal**. El problema consiste en agregarle una nueva funcionalidad al sitio web de la empresa tal que el usuario pueda seleccionar una ciudad **A** como origen y otra ciudad **B** como destino. Esta generará el itinerario de vuelos entre la ciudad **A** y la ciudad **B**. Para ello se cuenta con **n** distintos vuelos. De cada vuelo se conoce la ciudad de origen, la ciudad de destino, el momento de la partida y el momento de la llegada. Los momentos de partida y llegada representan la cantidad horas transcuridas desde que el usuario realizó la consulta. El itenerario de vuelos deberá iniciar en la ciudad **A** y llegar a la ciudad **B** los antes posible.

Por ejemplo, el usuario quiere ir de **Rosario** a **Madrid**. Al momento de la consulta, se cuenta con sólo 6 vuelos. Lo siguiente es la entrada que recibirá nuestro problema:

Rosario Madrid 6
Rosario Buenos_Aires 2 3
Buenos_Aires Madrid 6 18
Buenos_Aires Madrid 7 20
Rosario San_Pablo 3 6
San_Pablo Madrid 8 17
San_Pablo Madrid 7 16

Por ejemplo, existe un vuelo para ir desde Rosario a San Pablo que parte en 3hs y llega en 6hs posterior al momento de realizada la consulta. Otro vuelo disponible es ir de San Pablo a Madrid y parte dentro de 8hs y arribará dentro de 17hs a Madrid.

Dada esta instancia del problema, para ir desde Rosario y llegar a Madrid lo antes posible, la solución óptima sería tomar el vuelo de Rosario a San Pablo y luego el vuelo a Madrid (el que parte 8hs y arribará 17hs), llegando a destino 17hs luego de realizada la consulta.

Este es un ejemplo de salida de nuestro algoritmo:

17 2 4 5

Cabe destacar, que debe haber una diferencia de 2hs entre la llegada de un vuelo y el horario de partida del próximo vuelo en el itinerario. Por tal motivo, en la solución óptima propuesta, el vuelo número 6 no fue considerado y se utilizó el vuelo número 5.

1.2 Desarrollo

Dado que el problema trata de minimizar una magnitud (la cantidad de horas transcurridas desde realizada la consulta), nuestra primera idea fue modelar tal problema como un grafo y luego tratar de utilizar un algoritmo de camino mínimo como Dijsktra. En realidad, una variación de dicho algoritmo adaptado al problema en cuestión. A continuación describiremos como modelaremos el problema con un grafo G.

Vértices: Serán las distintas ciudades existentes en el listado de vuelos. Ejemplo: Rosario, Lima, Bogotá, etc.

Aristas: Conetaremos las ciudades con los vuelos disponibles. Como tendremos aristas dirigidas (ciudad origen $\rightarrow$ ciudad destino), en principio G será un grafo dirigido. Pero atención al siguiente detalle. Podríamos tener situaciones en las que existan mas de un vuelo con el mismo origen y destino. Entonces, por dicho motivo G será un multigrafo (sin bucles) dirigido.

En la figura 1 tenemos una representación gráfica de G que incluye algunas ciudad de Asia y algunos posibles vuelos entre dichas ciudades.

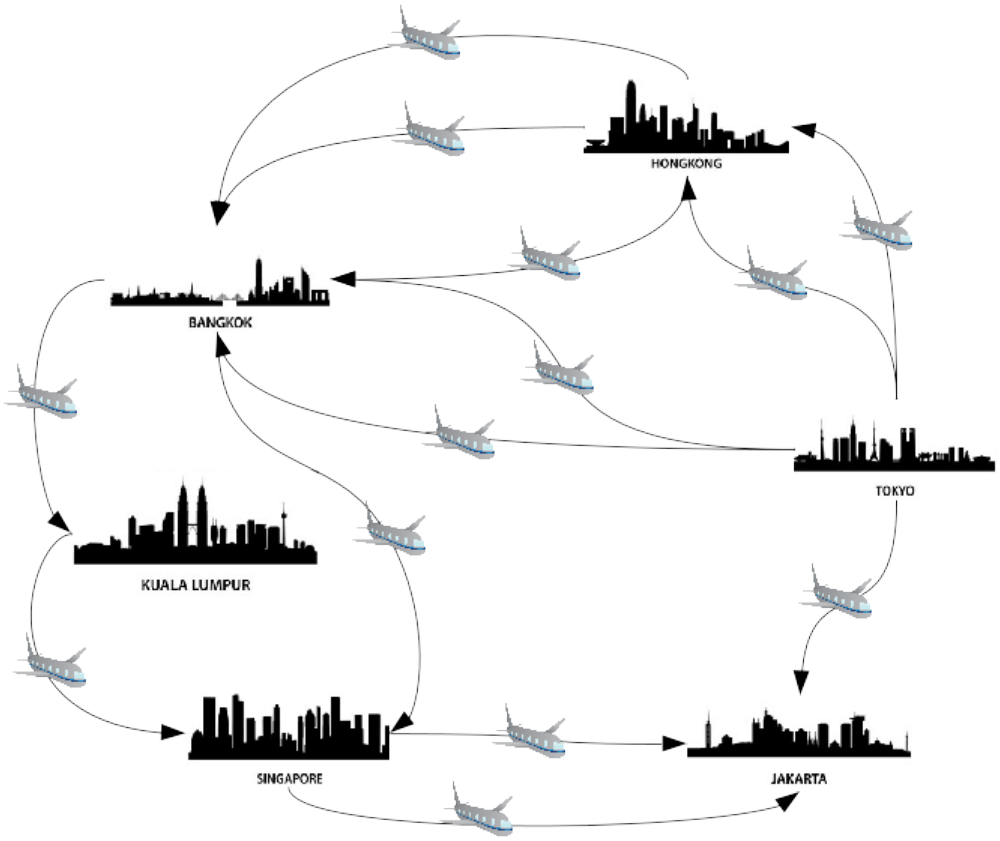

Figure 1: Ejemplo del grafo G con as ciudades como vértices y los vuelos como aristas

Ahora nos falta definir las magnitudes o valores asociados a las aristas (distancia) y el objetivo.

Distancia: Cada arista tendrá asociada la hora de llegada del vuelo, un valor entero no negativo que representa la cantidad de horas transcurridas desde el momento que se realizó la consulta.

Objetivo: Ir de un vértice (ciudad) a otro minimizando la distancia (la hora de llegada).

En la figura 2 tenemos una representación del problema de ir desde **Tokyo** a **Singapore** y queremos minimizar la hora de llegada. Las horas de llegadas se encuentran en las aristas (junto a los aviones).

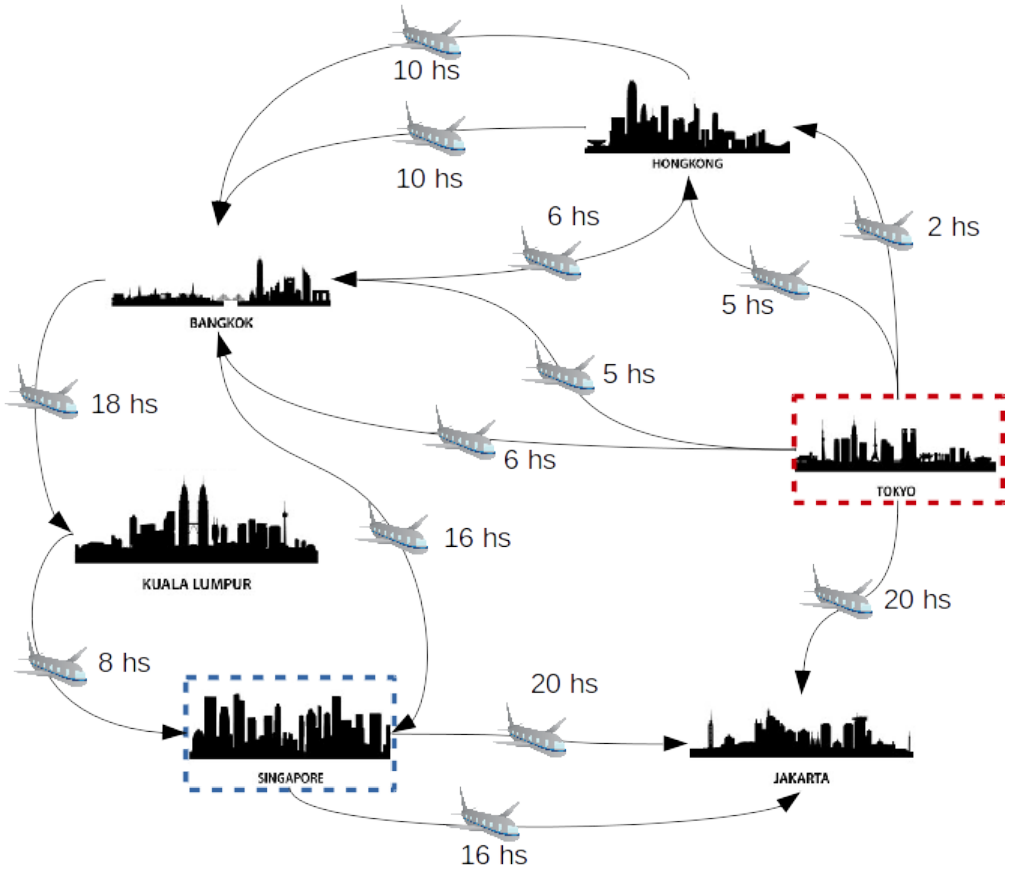

Figure 2: Ejemplo del grafo G para ir desde Tokyo a Singapore y llegar cuanto antes

Ahora tenemos nuestro problema modelado como un problema de grafos. Pero nos están quedando algunas cosas afuera como por ejemplo la restricción de 2hs entre vuelo y vuelo. Otro detalle al que hay que prestar atención es que si a este grafo G le aplicamos al algoritmo de camino mínimo de Dijsktra no nos va a dar la mínima hora de llegada al destino final. Entonces, para solventar estos 2 obstaculos que acabamos de encontrar realizaremos 2 pequeños cambios en el algoritmo original (Algoritmo 1).

Algoritmo 1: Dijsktra

Data: **origen**: vértice, **destino**: vértice, **k**: cant. ciudades, **adyacencias**: vector de adyacencias

```
1  begin
2      cola ⇐ ColaPrioridad()
3      distancia ⇐ Vector()
4      for i ∈ k do
5          distancia[i] ⇐ ∞
6      distancia[origen] ⇐ 0
7      cola.agregar(tupla(origen, distancia[origen]))
8      while ! cola.vacia() do
9          u ⇐ cola.minimo()
10         for v ∈ adyacencias[u.prim] do
11             if distancia[v.prim] ≥ distancia[u.prim] + v.terc then
12                 distancia[v.prim] ⇐ distancia[u.prim] + v.terc
13                 cola.agregar(tupla(v, distancia[v.prim]))
14     return distancia[destino]
```

El vector de adyacencias está indexado por la ciudad de origen y cada posición contiene una terna:
prim: Ciudad destino.
segu: Hora de partida.
terc: Hora de llegada.

En la cola prioridad (ordenada de menor a mayor) contiene tuplas, donde cada una contiene:
prim: Ciudad (actual).
segu: Hora de llegada.

El primer cambio que realizaremos en el algoritmo 1 será en vez acumular la distancia mínima en el vector *distancia* (línea 12), guardar la hora de llegada. Esto también nos obliga a modificar la línea 11 para ir guardando la hora mínima de llegada. De esta manera iremos gurdando la hora mínima de llegada para cada ciudad alcanzable desde la ciudad *origen* en el vector *distancia*. Cuando el algoritmo termine, en la posición *destino* del vector *distancia* tendremos la hora mínima de llegada para la ciudad *destino*. En caso de no haber un camino, esta posición tendrá ∞.

El siguiente cambio es la restricción de 2 horas entre vuelo y vuelo. Es decir, la hora de partida del próximo vuelo a tomar debe ser por lo menos mayor o igual en 2 horas a la hora de llega del vuelo previo. Los próximo vuelos con cidad origen $u.prim$ son los $v \in adyacencias[u.prim]$. Como $v.segu$ es la hora de partida y $u.segu$ la hora de llegada (notar que también puede utilizar $distancia[u.prim]$), basta con agregar la condición $distancia[u.prim] \leq v.segu - 2$ dentro del bucle mas anidado antes de que actulice la distincia (línea 11, algoritmo 2)

Algoritmo 2: DijsktraMod

```
   Data: origen: vértice, destino: vértice, k: cant. ciudades, adyacencias: vector de adyacencias
 1 begin
 2     cola ⇐ ColaPrioridad()
 3     distancia ⇐ Vector(k)
 4     for i ∈ k do
 5         distancia[i] ⇐ ∞
 6     distancia[origen] ⇐ 0
 7     cola.agregar(tupla(origen, distancia[origen]))
 8     while ! cola.vacia() do
 9         u ⇐ cola.minimo()
10         for v ∈ adyacencias[u.prim] do
11             if distancia[u.prim] ≤ v.segu - 2 then
12                 if distancia[v.prim] ≥ v.terc then
13                     distancia[v.prim] ⇐ v.terc
14                     cola.agregar(tupla(v, distancia[v.prim]))
15     return distancia[destino]
```

Al finalizar el algoritmo 2 devolverá la hora mínima de llegada a *destino* saliendo desde *origen*. En el problema original nos solicitan saber los números de los vuelos, es decir, el itinerario de vuelos para llegar a destino cuanto antes. Por tal motivo agregaremos algunas estructuras mas para poder reconstruir la solicitan. Agregaremos 2 vectores, uno para recordar al padre (ciudad) y otro para guarda el vuelo que le corresponde a dicha ciudad. Una vez que finaliza el bucle **while** y si este tiene una distancia que no sea ∞, entonces agregamos al itinerario el vuelo que nos lleva a *destino*, es decir *vuelo*[*destino*]. Luego buscamos la *ciudad* de la que partimos hacia destino con *padre*[*destino*] y si esta ciudad es distinta de *origen*, entonces agrego el vuelo al itinerario. Nuevamente busco la ciudad de la que partí para llegar a *ciudad* y si esta ciudad es distinta de *origen*, entonces agrego. Repetimos esto hasta encontrar la ciudad *origen*.

Algoritmo 3: DijsktraFinal

```
Data: origen: vértice, destino: vértice, k: cant. ciudades, adyacencias: vector de adyacencias,
      itinerario: lista de vuelos
1  begin
2      cola ⇐ ColaPrioridad()
3      distancia ⇐ Vector(k)
4      padre ⇐ Vector(k)
5      vuelo ⇐ Vector(k)
6      for i ∈ k do
7          distancia[i] ⇐ padre[i] ⇐ vuelo[i] ⇐ ∞
8      distancia[origen] ⇐ 0
9      cola.agregar(tupla(origen, distancia[origen]))
10     while ! cola.vacia() do
11         u ⇐ cola.minimo()
12         for v ∈ adyacencias[u.prim] do
13             if distancia[u.prim] ≤ v.segu - 2 then
14                 if distancia[v.prim] ≥ v.terc then
15                     distancia[v.prim] ⇐ v.terc
16                     padre[v.prim] ⇐ u.prim
17                     vuelo[v.prim] ⇐ v.id
18                     cola.agregar(tupla(v, distancia[v.prim]))
19     if distancia[destino] ≠ ∞ then
20         itinerario.agregar(vuelo[destino])
21         ciudad = padre[destino]
22         while ciudad ≠ origen do
23             itinerario.agregar(vuelo[ciudad])
24             ciudad = padre[destino]
25     return distancia[destino]
```

1.3 Complejidad

Para realizar el análisis de complejidad temporal utilizaremos el algoritmo 3.
La creación de la cola de prioridad vacía tiene una complejidad constante c. En cambio la creación e inicialización de los vectores tiene una complejidad de k cada 1, entonces tenemos $3 * k$ por la creación y k para inicializar. Un acceso al vector y una asignación (línea 8), constante. Agregar un elemento a la cola de prioridad, en este caso particular podemos considerarlo constante (porque la cola está vacía), no así mas adelante. Hasta aquí tenemos $4 * k + c$.
Luego inicia un bucle while. Analicemos primero la complejidad del cuerpo del bucle y luego determinaremos cuantas veces se ejecuta. Extraer el mínimo elemento de la cola de prioridad tiene una complejidad logarítmica[**?**] en la cantidad de elementos de la cola. En el peor de los casos, la cola puede llegar a tener k elementos (todas las ciudades), entonces, en el peor caso es $log\ k$. Esto también nos dice que el bucle $while$ se ejecutará k veces. Hasta aquí tenemos: $(4*k+c)+(k*log\ k)$. Ahora nos agregar la complejidad del bucle for. Como este bucle se va a ejecutar 1 vez para cada ciudad y para cada una se recorren sus vuelos, al final de todo se van a recorrer todos los vuelos. Es decir que se va a ejecutar n veces. En el cuerpo del bucle for hay varias comparaciones y asignaciones, las cuales podemos considerar de complejidad constante. No sucede lo mismo con el agregar a la cola de prioridad que en el peor caso tiene k elementos, lo que nos deja una complejidad de $log\ k$. Entonces el bucle tiene una complejidad en el peor caso de $n * log\ k$. Esto nos deja la complejidad por ahora en: $(4 * k + c) + (k * log\ k) + (n * log\ k)$.
Por último y no menos importante, falta analizar como se reconstruye la solución. Esta parte es sencilla, ya que si hay una solución, se agregan 1 a 1 los vuelos a una lista, agregar es constante y en el peor de los casos agrego los n vuelos.
La complejidad nos queda en: $(4*k+c)+(k*log\ k)+(n*log\ k)+(n+c) = k*(log\ k)+n*(log\ k)+4*k+n+2*c$.

Veamos que k (la cantidad de ciudades) está acotado por n (la cantidad de vuelos). A lo sumo puedo tener $2 * n$ ciudades distintas (2 por cada vuelo).
Dicho esto, nos queda: $2*n*(log\ 2*n)+n*(log\ 2*n)+4*2*n+n+2*c = 3*n*(log\ n)+3*n*(log\ 2)+9*n+2*c$.
Es decir $O(n * (log\ n))$.

1.4 Testing

En el archivo ej1/tester.cpp escribimos varios casos de prueba que nuestro algoritmo debía pasar.

El primer caso es para probar la restricción de 2hs entre vuelo y vuelo. Queremos ir desde Rosario a Madrid y llegar cuanto antes.

```
Rosario Madrid 6
Rosario Buenos_Aires 2 3
Buenos_Aires Madrid 6 18
Buenos_Aires Madrid 7 20
Rosario San_Pablo 3 6
San_Pablo Madrid 8 17
San_Pablo Madrid 7 16
```

El vuelo que llega antes a Madrid es el 6 pero no podemos tomarlo debido a que llegaríamos a a San Pablo desde Rosario a las 6 (vuelo 4) y como debemos esperar 2 horas hasta tomar el próximo vuelo, debemos tomar el vuelo número 5. La salida de nuestro programa debería ser:

```
17 2 4 5
```

Ahora, tenemos otro caso de prueba. Tenemos 10 vuelos disponibles para ir de Rosario a Paris y llegar cuanto antes.

Rosario Paris 10
Rosario Buenos_Aires 2 3
Buenos_Aires Madrid 6 18
Buenos_Aires Madrid 7 20
Rosario San_Pablo 3 6
San_Pablo Madrid 8 17
San_Pablo Madrid 7 16
Madrid Paris 17 19
Barcelona Paris 19 20
Madrid Paris 18 20
Madrid Paris 19 20

Noten que la mayoría de los vuelos que arriban a Paris lo hacen em 20hs salvo 1 que lo hace en 19hs, procedente de Madrid pero para llegar a tomar ese vuelo deberíamos llegar a Madrid cuando mucho en 15hs y no hay ningún vuelo que cumpla eso. Queda descartado. Vamos de Rosario a San Pablo y luego a Madrid (vuelos 4 y 5). Desde Madrid sólo tenemos una posibilidad (vuelo 10).

20 3 4 5 10

Por último, un caso en el que no existe solución. Tengo 4 vuelos disponibles para ir de Rosario a Madrid.

Rosario Madrid 4
Rosario Buenos_Aires 2 3
Rosario San_Pablo 3 6
San_Pablo Barcelona 8 17
Barcelona Madrid 7 16

Para salir de Rosario tengo 2 vuelos: ir a Buenos Aires o ir a San Pablo. Desde Buenos Aires no hay vuelos, mientras que desde San Pablo puedo tomar el vuelo a Barcelona. Pero una vez en Barcelona no hay vuelos que me lleven a Madrid, pues llegaría a Barcelona en 17hs y el vuelo a Madrid sale en 7hs. Para este caso no hay solución y nuestro programa debería imprimir:

no

1.5 Experimentación

Para la experimentación decidimos tomar ciertos conjuntos de datos. Los conjuntos de datos se caracterizan de la siguiente manera:

k = n/2: La cantidad de ciudades distintas (k) es alrededor la mitad de la cantidad de vuelos (n).
k = n: La cantidad de ciudades distintas (k) es aproximadamente igual cantidad de vuelos (n).
k = 2n: La cantidad de ciudades distintas (k) es alrededor el doble de la cantidad de vuelos (n).

La medición de tiempo se realizó de la siguiente manera: Para cada instancia/problema, se ejecutó 50 veces el algoritmo sobre dicha instancia, acumulando los tiempos utilizando la biblioteca chrono. Luego se dividió el tiempo acumulado en 50, para tener un promedio del tiempo.

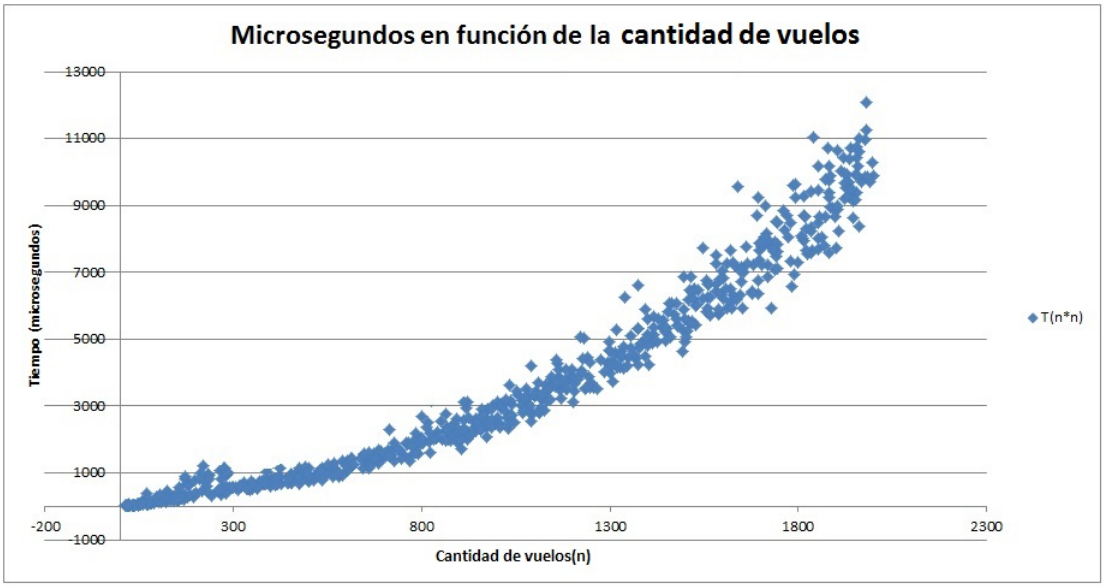

Figure 3: T(n^2) = tiempos n vuelos. Estos resultados son para el conjunto de datos $k = n/2$. La curva tiene un forma cuadrática en función de la cantidad de vuelos. Para poder confirmar esto, en el siguiente gráfico dividiremos por n^2.

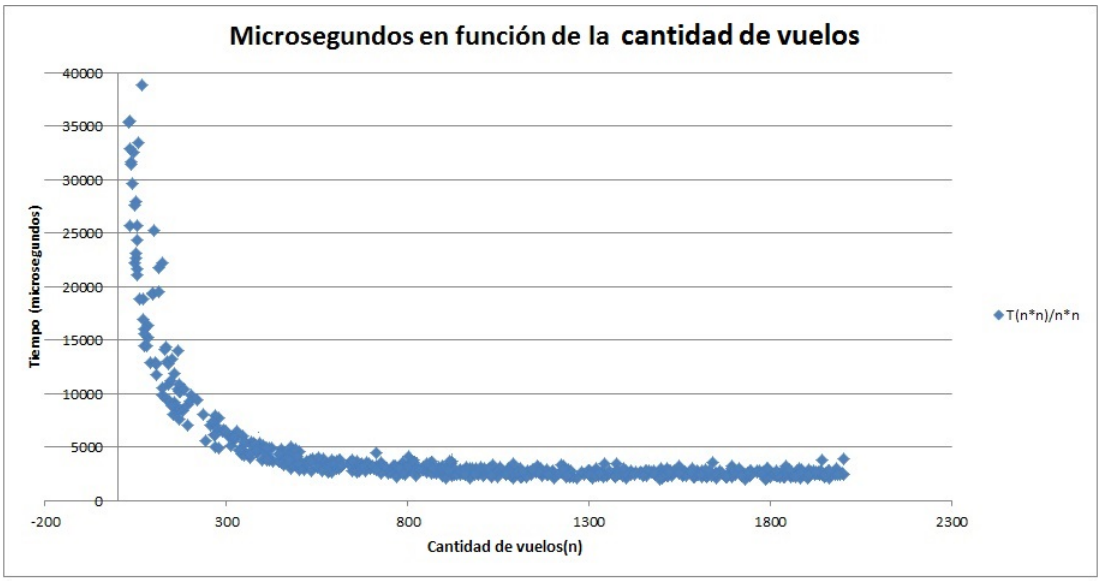

Figure 4: T(n^2) = tiempo para n vuelos. Esta figura es el resultado de dividir los tiempos de la gráfica anterior por n^2. Aquí se puede ver como estos resultados están acotados por una constante y parecería que continua decreciendo.

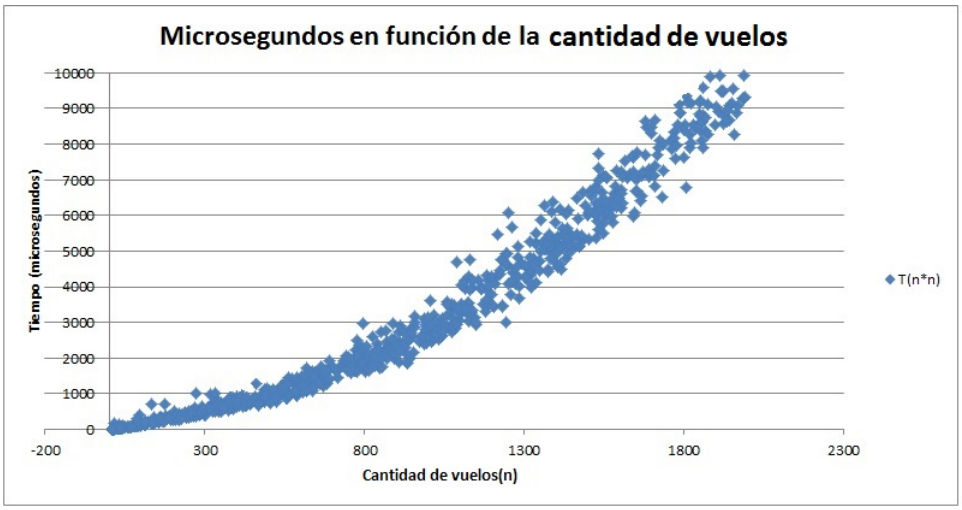

Figure 5: T(n^2) = tiempo para n vuelos. Estos resultados son para el conjunto de datos $k = n$. Para este caso, la curva no parece ser tan pronunciada como en el caso anterior, que se parecía a n^2. Una vez mas volveremos a dividir por n^2.

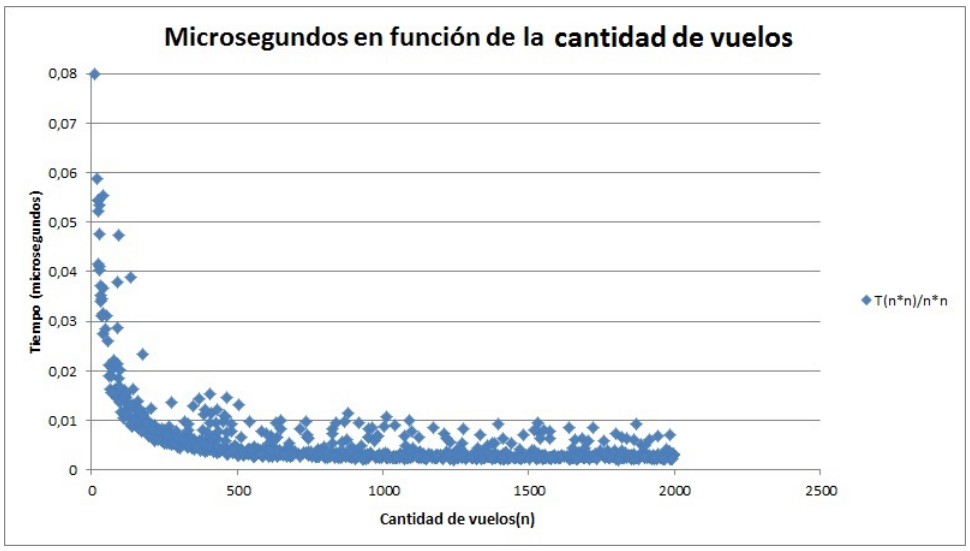

Figure 6: T(n^2) = tiempo para n vuelos. Esta figura es el resultado de dividir los tiempos de la gráfica anterior por n^2. Nuevamente podes apreciar que los resultados están acotados por una constante y otra vez parecería que continua decreciendo.

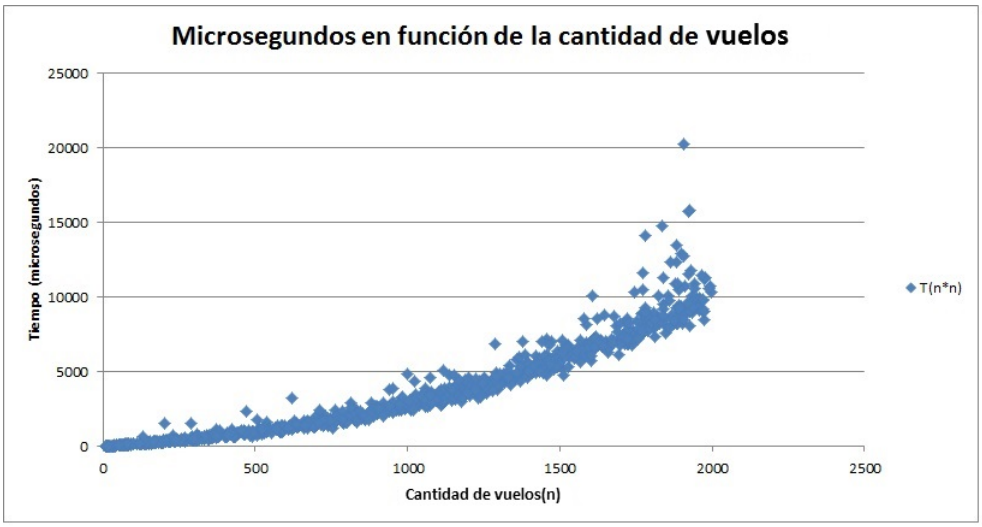

Figure 7: T(n^2) = tiempo para n vuelos. Estos resultados son para el último conjunto de datos $k = 2n$. En este caso, vemos que los resultados tienen una forma mas parecida a una recta con una leve inclinación. Bastante distinto a los 2 primeros conjuntos de datos propuestos. Salvo una leve alteración en n=200 (la cual no pudimos analizar), el gráfico es bastante ser uniforme. Por último, dividiremos por n^2 los resultados en busca de la comstante.

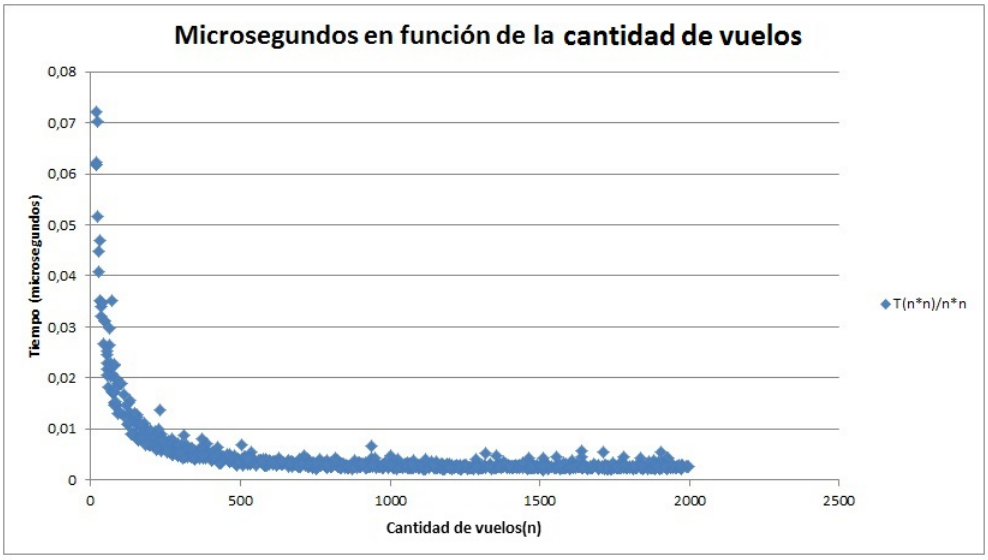

Figure 8: T(n^2) = tiempo para n vuelos. Esta figura es el resultado de dividir los tiempos de la gráfica anterior por n^2. En este último gráfico, podemos ver la constante con un leve decrecimiento.

1.6 Conclusión

Dado el problema del plan de vuelo, lo modelamos con un multigrafo (sin bucles) dirigido y mediante una modificación del algoritmo de camino mínimo de Dijsktra pudimos resolverlo. En el análisis teórico de complejidad temporal concluimos que el peor caso estaba en $O(n * (log\ n)$ pero durante la experimentación computacional, los datos empíricos nos mostraron que nuestro algoritmo que con ciertos conjuntos de datos tiene un comportamiento cuadrático. Cabe destacar que aún así, sigue respetando la exigencia del problema, que la solución debía ser no peor que $O(n^2)$.

2 Problema 2: Caballos salvajes

2.1 Introducción

La misma empresa nacional del anterior problema ahora quiere incursionar en el negocio de los juegos de mesa, por ello han sacado un innovador juego que utiliza un tablero cuadrado de ajedrez (hay en distintos tamaños) en donde se colocan en algunas casillas caballos blancos de ajedrez. El juego permite que haya mas de un caballo en la misma casilla, y los caballos pueden moverse por el tablero como lo harían en el ajedrez. Para ganar el juego hay que lograr ubicar a todos los caballos en una misma casilla en la menor cantidad total de movimientos posibles, es decir que la suma de los movimientos de todos los caballos en el tablero sea la menor. A continuación se muestra un ejemplo del juego con su respectiva solución:

(a) Inicio del juego **(b)** Solucion al juego

Figure 9: Aunque no es visible en (b) se encuentran los dos caballos en el mismo casillero

2.2 Desarrollo

Este problema lo resolvimos con una serie de pasos, primero comenzamos representando el tablero con el siguiente grafo:
Sea G(V,E) el grafo donde cada nodo v ϵ V representa una casilla del tablero y dos nodos v_1 y v_2 ϵ V están unidos por una arista e ϵ E $\Leftrightarrow$ es posible mover al caballo de la casilla v_1 a la casilla v_2.
Luego para cada caballo en su correspondiente casilla utilizamos el algoritmo de BFS sobre G, el cual para grafos con aristas no pesadas o de igual peso nos da el camino mínimo desde un nodo origen hasta cada uno del resto de los nodos del grafo [1]. Al algoritmo de BFS lo modificamos para que a medida que busca el camino mínimo a cada casilla calcule la cantidad de movimientos que le toma llegar a ésta casilla.
Después sumamos lo que le cuesta a cada caballo llegar a cada casilla obteniendo el total de movimientos que cuesta que todos los caballos lleguen a cada casilla. Finalmente nos quedamos con la casilla que lleguen todos los caballos y que haya resultado con menor cantidad total de movimientos. En caso que no haya ninguna casilla tal que todos los caballos lleguen a esta entonces la solución al problema será "no".

Pseudocódigo:

```
1  creamos una matriz acumMovimientos inicializada en 0
2  creamos una matriz cantidad inicializada en 0
3
4  para cada caballo
5     mientras recorremos G usando BFS desde la casilla donde se encuentra ubicado
6        cantMov ← contamos la cantidad de movimientos a cada casilla
7        acumulamos en acumMovimientos lo que contamos con cantMov
8     fin mientras
9     para cada casilla
10       acumulamos en cantidad si el caballo paso por esa casilla
11    fin para
12 fin para
13
14 buscar en acumMovimientos la casilla que tenga menor cantidad de movimientos y que en cantidad
15 hayan pasado todos los caballos, retornar la casilla y la cantidad de movimientos
16
17 En caso que no existe tal casilla retornar "no"
```

[1] Cormen, T.,Leiserson, C.,Rivest,R.,Stein, C.,"Introduction to Algorithms", third edition, pagina 594

2.3 Demostración de correctitud

Antes que nada como nuestro algoritmo usa BFS, veamos que es correcto. Esto es cierto, ya que esta demostrado en el "Cormen" [2].

Ahora para demostrar la correctitud de nuestro algoritmo vamos a usar inducción en n = cantidad de caballos.

Caso Base: Queremos ver que nuestro algoritmo es correcto, es decir que nos da la solución óptima para 1 caballo, esto es fácil de ver pues el algoritmo aplicara BFS para calcular la cantidad de movimientos que le cuesta llegar al caballo a cada una de las casillas, y estos cálculos los guardara en la matriz $acumMovimientos$. Luego buscara en $acumMovimientos$ la casilla con menor cantidad de movimientos y se encontrara con que la casilla donde comenzó el caballo cuesta 0 (ya que no se movería) y devolvería esta casilla que costo 0 movimientos como resultado.

Paso inductivo: P(n) = nuestro algoritmo calcula correctamente en una matriz, para cada posición si llegan los n caballos y la cantidad de movimientos totales de los n caballos para llegar a tal posición. Queremos ver que vale P(n+1), esto quiere decir que queremos ver que para n+1 caballos el algoritmo nos calcula correctamente la matriz con la cantidad de movimientos totales de cada casilla y la cantidad de caballos que pasan por cada casilla.

Por hipótesis inductiva vale P(n), veamos qué pasa cuando agregamos el n+1 caballo, el algoritmo recorrerá G (definido en la sección 2.2) usando BFS y para cada casilla contara la cantidad de movimientos. Una vez terminado el recorrido de BFS, se recorrerá el tablero para saber a qué casillas llega y la cantidad de movimientos a la casilla del n+1 caballo acumulando esto en lo que ya se tenía para los n caballos anteriores. Con esto terminaremos teniendo en una matriz el costo total de movimientos para cada casilla de los n+1 caballos y la cantidad de caballos que pasan por cada casilla, que es lo queríamos pues solo falta buscar la casilla donde hayan llegado todos los caballos y el costo total sea mínimo.

[2] Cormen, T.,Leiserson, C.,Rivest,R.,Stein, C.,"Introduction to Algorithms", third edition, pagina 594

2.4 Complejidad

A continuación el extracto más importante del código para analizar la complejidad:

Caballos Salvajes()

```
 1  matriz_t acumMovimientos(n_, vector<int>(n_, 0));                          // O(n^2)
 2  matriz_t cantidad(n_, vector<int>(n_, 0));                                 // O(n^2)
 3
 4  for(int cab = 0; cab < k_; ++cab) {                                        // O(k*n^2)
 5    matriz_t movimientos(n_, vector<int>(n_, 0));                            // O(n^2)
 6    bfs(caballos_[cab], movimientos);                                        // O(n^2)
 7
 8    for (int i = 0; i < n_; ++i) {                                           // O(n^2)
 9      for (int j = 0; j < n_; ++j) {                                         // O(n)
10        acumMovimientos[i][j] += movimientos[i][j];                          // O(1)
11        if (movimientos[i][j] > 0) {                                         // O(1)
12          ++cantidad[i][j];                                                  // O(1)
13        } else {
14          if ( (caballos_[cab].first-1 == i) && (caballos_[cab].second-1 == j)) { // O(1)
15            ++cantidad[i][j];                                                // O(1)
16          }
17        }
18      }
19    }
20
21  m_ = numeric_limits<int>::max();                                           // O(1)
22  for (int i = 0; i < n_; ++i) {                                             // O(n^2)
23    for (int j = 0; j < n_; ++j) {                                           // O(n)
24      if (cantidad[i][j] == k_) {                                            // O(1)
25        m_ = acumMovimientos[i][j];                                          // O(1)
26        f_ = i+1;                                                            // O(1)
27        c_ = j+1;                                                            // O(1)
28      }
29    }
30  }
```

Podemos ver que comienza creando 2 matrices de dimensiones n*n (líneas 1 y 2), lo cual demandara esa cantidad de operaciones $O(n^2)$. Después hay un for (línea 4) que itera k veces y en cada iteración tiene que:

- crear la matriz *movimientos* (línea 5) de dimensión n*n $\rightarrow O(n^2)$
- BFS (línea 6) que cuesta $O(n^2)$
- un for (línea 8) que iteran n veces, y en cada iteración tiene otro for (línea 9) que también itera n veces, y este en cada iteración hace asignaciones y comparaciones (líneas 10 a 15) que cuestan O(1). Entonces el for de la línea 8 tiene costo $O(n*n*1) = O(n^2)$

Por lo tanto el costo del for de la línea 4 es $O(k*(n*n + n*n + n*n)) = O(k*3*n^2) = O(k*n^2)$.

Luego tenemos la asignación de la línea 21 que es O(1) y el for de la línea 22 que itera n veces, y en cada iteración tiene el for de la línea 23 que tambien itera n veces, éste en cada iteración hace asignaciones y comparaciones (líneas 24 a 27) que cuestan O(1), así que el for de la línea 22 cuesta $O(n*n*1) = O(n^2)$.

Finalmente el costo del algoritmo es de $O(k*n*n + 1 + n*n) = O(k*n^2)$. Ahora corroboremos que la complejidad de nuestra implementación de BFS es la mencionada:

bfs (coord_t origen, matriz_t &m)

```
1  queue<coord_t> cola;                                                    // O(1)
2  vector<bool> aux(n_, false);                                            // O(n)
3  vector<vector<bool > > visitados(n_, aux);                              // O(n^2)
4  cola.push(origen);                                                      // O(1)
5  visitados[origen.first-1][origen.second-1] = true;                      // O(1)
6
7  while(!cola.empty()) {                                                  // O(1)
8     coord_t cab = cola.front(); cola.pop();                              // O(1)
9     list<coord_t> adyacentes;                                            // O(1)
10    dameAdyacentes(cab, adyacentes);                                     // O(1)
11
12    for(auto &v: adyacentes) {                                           // O(1)
13       if (visitados[v.first-1][v.second-1] == false) {                  // O(1)
14          visitados[v.first-1][v.second-1] = true;                       // O(1)
15          m[v.first-1][v.second-1] = m[cab.first-1][cab.second-1] + 1;    // O(1)
16          cola.push(v);                                                  // O(1)
17       }
18    }
19 }
```

Comienza creando la cola que es O(1), también creando 2 vectores uno de dimensión n y otro n*n, que cuesta O(n) y $O(n^2)$ respectivamente.

Meter una coordenada en la cola y la asignación de la línea 5 cuestan O(1).

Luego tenemos un while (línea 7) que comprueba si la cola esta vacía lo que demanda tiempo constante y en cada iteración hace:

- saca la siguiente coordenada de la cola y crea una lista de coordenadas (líneas 8 y 9) que cuestan O(1)
- la función dameAdyacentes (línea 10) cuesta O(1) pues son comparaciones y asignaciones (el codigo se encuentra en "TP2/ej2/solucion.h"
- por ultimo un for (línea 12) que itera sobre la cantidad de coordenadas de *adyacentes*, que a lo sumo son 8 pues son las casillas a las que se pueden llegar con los movimientos de un caballo. En cada iteración hace comparaciones, asignaciones y meter una coordenada en la cola (líneas 13 a 16) que son O(1). Por lo tanto el for cuesta O(8*1) = O(1).

Entonces el while tiene costo de O(1). y el algoritmo cuesta $O(n^2)$.

2.5 Testing

Testeamos los siguientes casos de interés:

- Tablero de 4*4 con 5 caballos en distintas casillas
- Tablero cualquiera con todos los caballos en la misma casilla
- Instancia generada aleatoriamente

Los test se pueden ver en el archivo: "TP2/ej2/src/tester.cpp"

2.6 Experimentación

Una vez hecho el análisis de la complejidad teórica, realizamos experimentos con el fin de contrastar los resultados empíricos y comprobar que el algoritmo implementado efectivamente tendrá una complejidad temporal de $O(k*n^2)$.

Para los siguientes experimentos para cada caballo se toman dos distribuciones discretas uniforme entre 1 y 100 que se utilizan como valor para la posición de cada caballo.

Para el primer experimento vamos a variar la cantidad de caballos y dejar fijo el tamaño del tablero en n=100. Para reducir los posibles errores de medición y evitar que los resultados se vean alterados por entradas de peor o mejor caso sesgando los resultados, se decidió tomar una cantidad fija de 220 instancias generadas de la forma antes descrita para cada n y k. En el siguiente grafico se pueden observar los resultados:

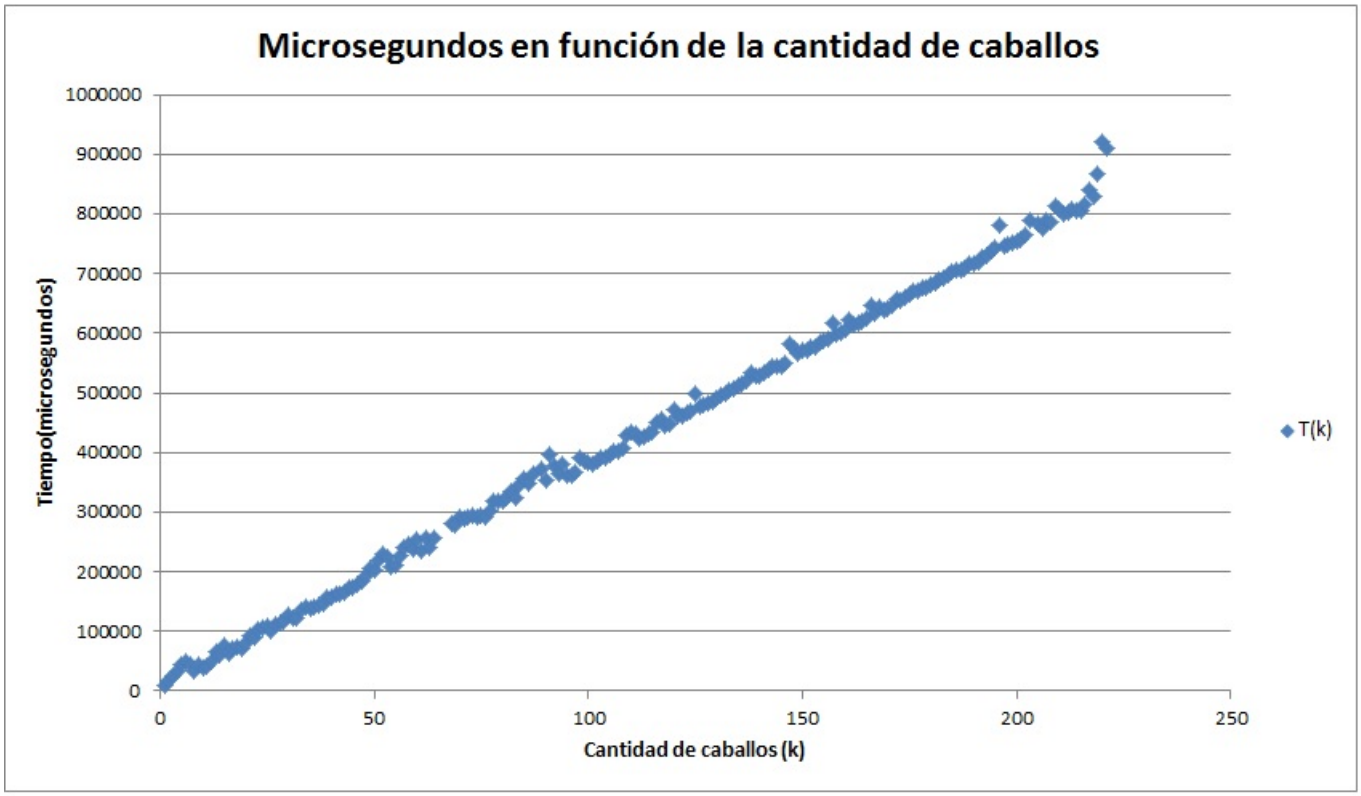

Figure 10: T(k)=tiempo para k caballos

Como podemos ver, obtuvimos una recta lo cual concluye al dejar constante n que la complejidad es O(k).

Veamos qué pasa cuando dejamos fija la cantidad de caballos en k=20 y variamos el tamaño del tablero (n) con una cantidad de 200 instancias:

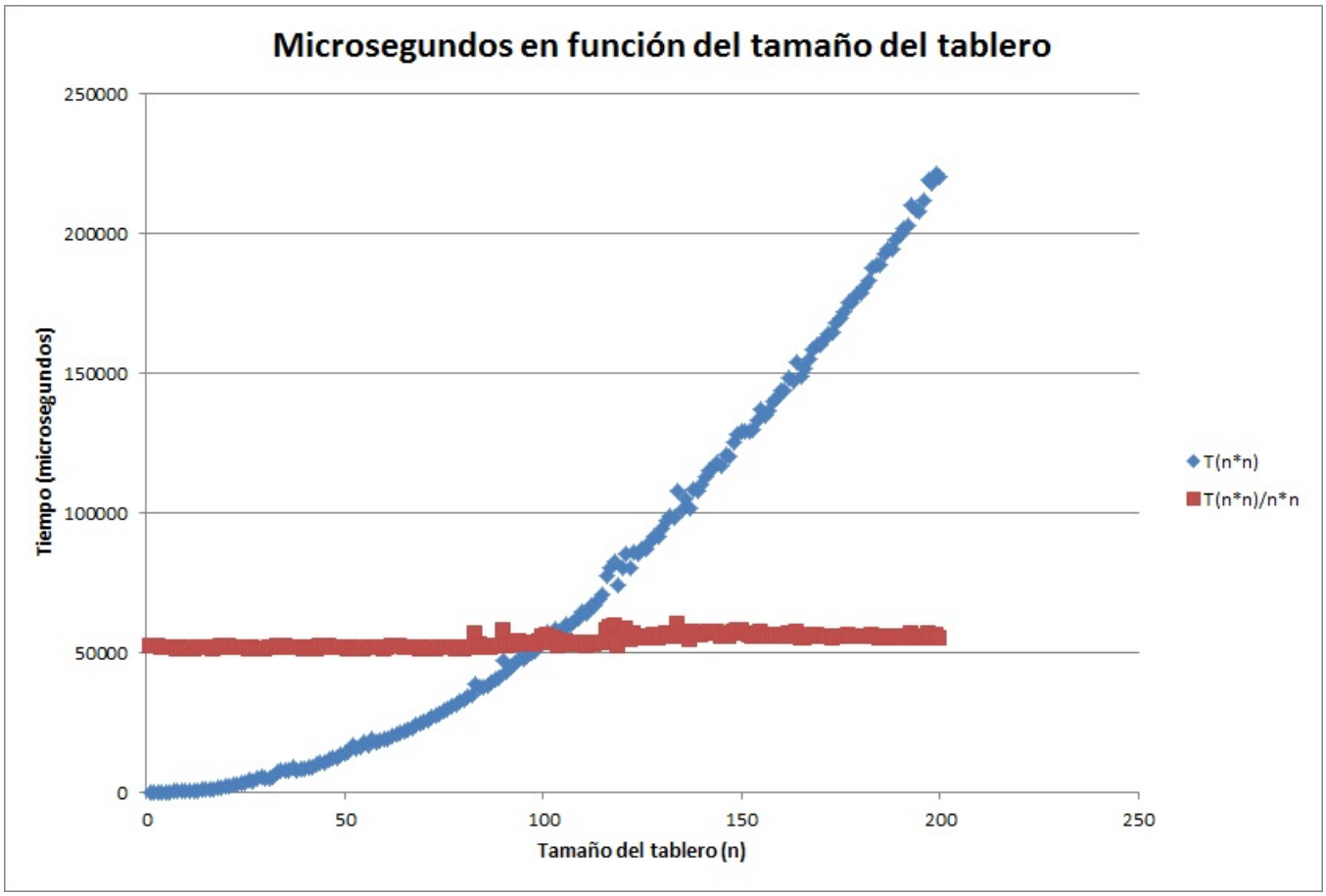

Figure 11: T(n^2)=tiempo para tablero de tamaño n

Como conclusión podemos ver que al dejar el k como una constante obtenemos una complejidad cuadrática del algoritmo contrastada en el gráfico.

Por ultimo una comparación cuando k y n varían de la misma manera con una cantidad de 300 instancias

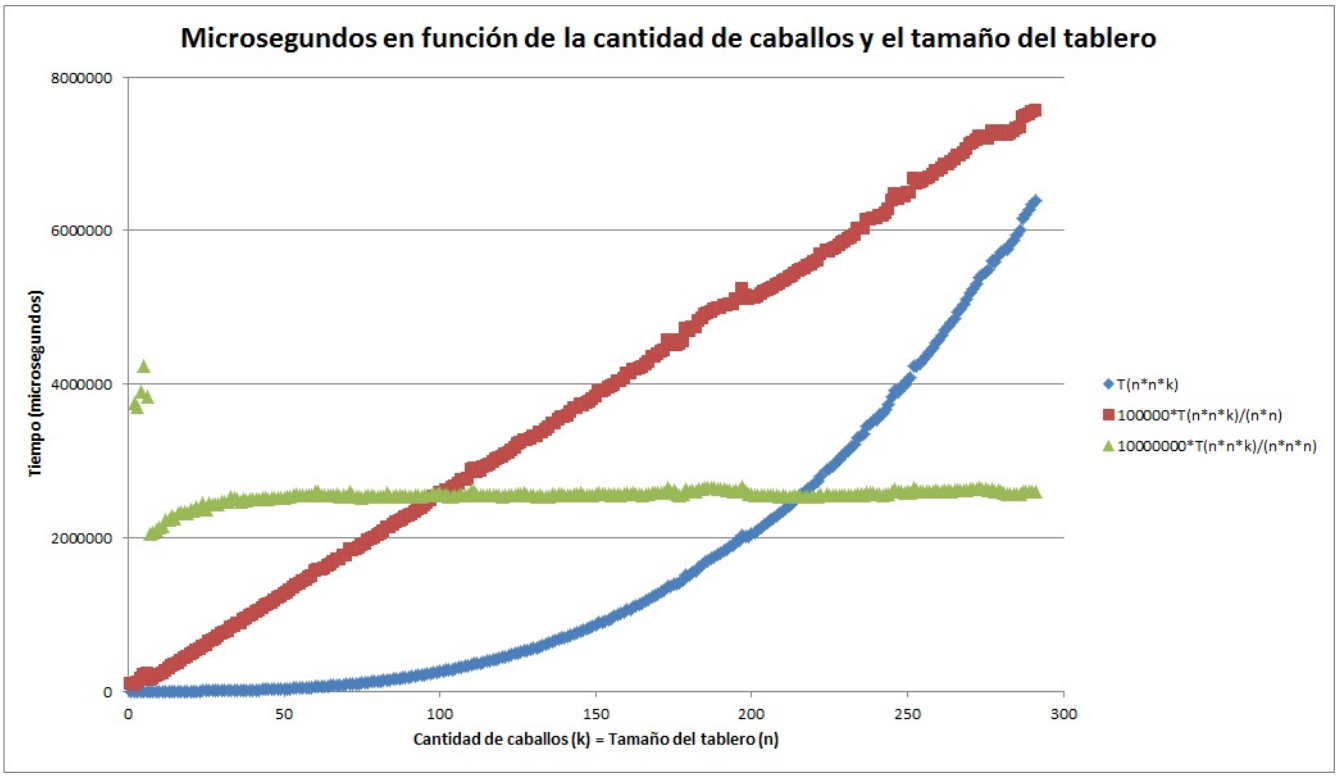

Figure 12: T(n^2*k)=tiempo para tablero de tamaño n = tiempo para k caballos

En este último caso, se puede verificar una complejidad de O(n^{*2}*k)

2.7 Conclusión

Este ejercicio nos enseñó la posibilidad de utilizar algoritmos que a comienzo sirven para el recorrido de un grafo (BFS) como una manera de calcular caminos mínimos de grafos no pesados, o con todas las aristas del mismo peso. Con respecto al desarrollo, a priori no se nos presentó una dificultad al resolverlo, contamos con varias estrategias para encararlo, una de ellas era resolverlo con k matrices distintas para tener en cada matriz el camino mínimo a todas las casillas de un caballo nada más. Además pudimos cumplir con la complejidad que nos delimitaban y corroboramos empíricamente que el tiempo del algoritmo cumplía con esta.

3 Problema 3: La comunidad del anillo

3.1 Introducción

Nuestra empresa, AlgoNET, se dedica principalmente a brindar soluciones algorítmicas para problemas de redes y en esta ocasión nos enfrentamos al siguiente desafío.

Nuestro cliente quiere ofrecer un servicio particular sobre de una red existente de computadoras. La red consta actualmente de conexiones entre algunos pares de equipos y cada enlace tiene un costo de utilización determinado.

Se pretende seleccionar algunas de estas computadoras para utilizar como servidores formando un backbone con topología de anillo (cada servidor tiene un enlace directo con exactamente otros 2, uno de entrada y otro de salida). Adicionalmente se requiere que todas las demás computadoras puedan tener acceso a los servidores del anillo, ya sea por un enlace directo o pasando a traves de otros equipos en el camino.

Se nos pidío implementar un algoritmo con complejidad estrictamente menor a $O((\#computadoras)^3)$ que nos permita saber que conexiones utilizar formar el anillo y cuales emplear para conectar el resto de los equipos, de manera de minimizar el costo incurrido por el uso de los enlaces.

3.1.1 Ejemplo práctico:

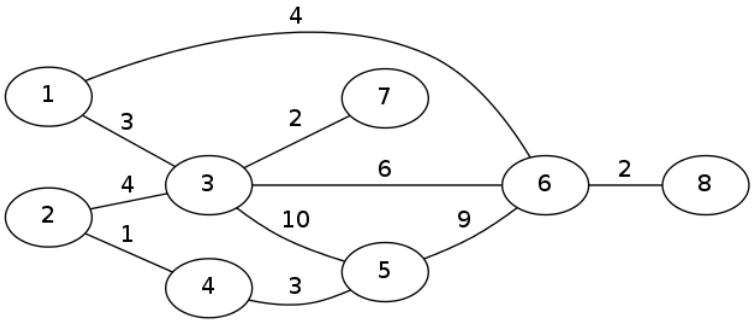

Figure 13: Ejemplo de red con 8 computadores y 10 enlaces con costos asociados

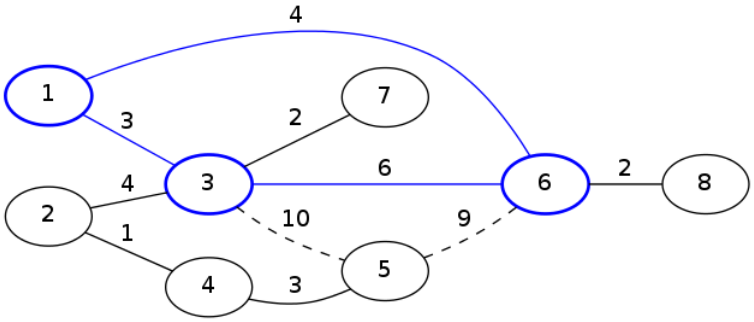

Figure 14: Solución del ejemplo

- Los equipos 1, 3 y 6 forman el anillo de servidores.
- Los enlaces (5,6) y (3,5) no son utilizados.
- El resto de las computadoras se encuentran conectadoa con el anillo ya sea directamente como el equipo 8 con el servidor 6 o indirectamente como el computador 4 que se conecta con el servidor 3 pasando primero por el equipo 2.
- Costo total de los enlaces utilizados = 25.

3.2 Idea General

Decidimos modelar el problema utilizando teoría de grafos.

- Cada nodo representará una computadora de la red.
- Dos nodos serán adyacentes si y solo si existe un enlace entre los 2 equipos.
- Será un grafo pesado, el peso de una arista representa el costo de utilización del enlace.

Dividiremos el problema en dos partes:

- Primero dejaremos de lado la parte de formar un anillo y conectaremos todas las computadoras minimizando el costo incurrido en los enlaces, es decir, buscaremos un arbol generador minimo sobre el grafo.
- Luego, buscaremos la arista de menor peso que no forme parte del AGM y se la agregaremos. De esta manera formaremos un ciclo y ese será nuestro "anillo".

3.3 Desarrollo

La implementación fue hecha en codigo $c++$.

Para buscar el AGM implementamos un algoritmo basado en PRIM

Algoritmo 4: buscar_AGM()

```
1  begin
2      if # Aristas < # Nodos then
3          No existe solución
4          return 1
5      cola = heap con punteros a todos los nodos del grafo
6      raiz→distancia = 0
7      while !esta_vacia(cola) do
8          u = extraer_minimo(cola)
9          if u→distancia == INF then
10             Sin Solución, no es conexo
11             return 1
12         costoTotal += u→distancia
13         Anotamos la arista que une u con u→Padre como parte del AGM
14         agendamos u como ya recorrido
15         for v ∈ adyacentes de u do
16             if !recorrido(v) && v→distancia > peso(u,v) then
17                 v→padre = u
18                 v→distancia = peso(u,v)
19     return 0
```

Donde

- *raiz* es el Nodo numero 1 del grafo.
- *distancia* se encuentra inicializada en INF para todos los nodos.
- El *padre* de cada nodo se encuentra inicializado en NULL.
- El heap esta ordenado por menor distancia.
- En *costoTotal* guardamos el costo total de la utilización de los enlaces incluidos en el AGM.

Una vez que tenemos generado en AGM, procedemos a buscar la arista de menor peso no incluida en el AGM para formar un ciclo, de la siguiente manera:

Algoritmo 5: formar_anillo()

```
1  begin
2      sort 'aristas' de menor a mayor peso
3      for e ∈ aristas do
4          if e ∉ AGM then
5              costoTotal += peso(e)
6              arista_extra = e
7              break
8      for x1, x2 ∈ nodos incidentes a arista_extra do
9          while x1 ≠ raiz && x1 ≠ x2 do
10             Anotamos x1 como parte del anillo
11             x1 = x1 →padre
12     for k ∈ nodos do
13         if k ≠ raiz then
14             if k esta anotado como parte del anillo then
15                 anillo.push_back(arista que une a k con su padre)
16             else
17                 resto.push_back(arista que une a k con su padre)
```

3.4 Demostración de correctitud

Primero notemos que la solución óptima posee un solo circuito o "anillo", ya que si posee más de uno puedo eliminar uno de ellos, pues sea C uno de ellos donde C = v_1, v_2, ..., v_i, v_1 con $2 \leq i < n$ puedo eliminar la arista que une v_i con v_1 cuyo peso es > 0 obteniendo una solución del problema de peso menor que la óptima por lo que es absurdo.

Queremos ver que la solucion que da nuestro algoritmo es optima, es decir que cualquier solucion optima es un AGM + la arista menos pesada fuera del AGM. Por lo que vamos a ver que:

- Dado A una solucion optima, sea "e" el eje más pesado del único circuito que posee, queremos ver que A-{e} es un AGM.

  Demostración:
  Sea T AGM $\Longrightarrow$ costo(T) $\leq$ costo(A-{e}) ya que A-{e} es un AG hasta el momento (es un grafo acíclico ya que sacamos una arista del único ciclo que poseía, y es conexo que contiene a todos los nodos porque la solución tiene que contener a todos los nodos, existiendo un camino entre todos ellos hacia el anillo, por lo tanto, es un árbol generador).
  Si e no está en T, ya está, ya que si A fuera de mayor peso que T rearmaría la solución como T+{e}, la cual posee un único ciclo obteniendo una solución de menor peso, lo que es absurdo porque A era óptima. Entonces costo(A-{e}) $\leq$ costo(T) y por lo tanto A-{e} es un AGM.
  Caso contrario tomo un eje e' que no pertenece a T del circuito que contiene a "e" en A (existe ya que como e pertenece a T, si todas las demás aristas también pertenecieran a T, entonces T tendría un ciclo, y esto es ABS!). Luego T+{e'} tiene un único circuito simple.
  Si el costo de A fuera > que costo(T+{e'}), utilizaría a T+{e'} como solución y sería mejor que la óptima, que es ABS! $\Rightarrow$ costo(A) $\leq$ costo(T+{e'}) por lo que podemos llegar a que:

  Costo(A) $\leq$ costo(T)+costo({e'}) $\leq$ costo(T)+costo(e) ya que "e" es la arista más pesada del ciclo en A y e' pertenecía a ese ciclo en A. Resultando en costo(A-{e}) $\leq$ costo(T) como queríamos demostrar.

- Ahora vamos a demostrar que si tomo un AGM T, entonces el eje e' de peso mínimo del grafo que no

esta en T, entonces peso(e') $\leq$ peso(e)

Para esto vamos a renombrar ciertas cosas:

- T = el AGM del óptimo = A-{e'}
- T' = un AGM
- e = eje mas pesado del ciclo en la solución óptima
- e' = eje más liviano fuera de T'

Demostración:
Sabemos que existe un eje e" de C que no está en T' ya que T' es a cíclico. Con esto podemos derivar que costo(e') $\leq$ costo(e") ya que e' y e" son ejes que están fuera de T' y e' era la más liviana. Y en particular costo(e") $\leq$ costo(e) ya que e era el eje más pesado del ciclo C. Con lo que queda que costo(e') $\leq$ costo(e) como queríamos demostrar

3.5 Complejidad

buscar_AGM es basicamente un algoritmo de PRIM, el mismo tiene complejidad temporal O(n^2).
Analizemos entonces la complejidad de formar el anillo:

formar_Anillo(){

```
1   sort(aristas.begin(), aristas.end());                                  // O(M*log M)
2   pair<Nodo*, Nodo*> arista_extra;                                       // O(1)
3   for (unsigned int i = 0; i < aristas.size(); i++)                      // O(M)
4   {
5     if(aristas[i].second.first→arista_en_AGM[aristas[i].second.second→numero] == 0) // O(1)
6     {
7       C += aristas[i].first;                                             // O(1)
8       arista_extra = aristas[i].second;                                  // O(1)
9       break;
10    }
11  }
12  Nodo * a = arista_extra.first;                                         // O(1)
13  Nodo * b = arista_extra.second;                                        // O(1)
14  for (int i = 0; i < 2; i++)                                            // O(1)
15  {
16    while(a ≠ NULL && a ≠ b)                                             // O(N)
17    {
18      a→en_anillo = true;                                                // O(1)
19      a = a→padre;                                                       // O(1)
20    }
21    a = arista_extra.second;                                           // O(1)
22    b = arista_extra.first;                                            // O(1)
23  }
24  anillo.push_back(arista_extra);                                        // O(1)
25  for (int i = 1; i ≤ N; i++)                                            // O(N)
26  {
27    Nodo* k = nodos[i];                                                  // O(1)
28    if (k→padre ≠ NULL)                                                  // O(1)
29    {
30      if (k→en_anillo)                                                   // O(1)
31      {
32        anillo.push_back(make_pair(k, k→padre));                         // O(1)
33      }
34      else
35        resto.push_back(make_pair(k, k→padre));                          // O(1)
36    }
37  }
38 }
```

- La función de sort ordena todas las aristas seǵun su peso, tiene complejidad O(M*log M) con M = # de aristas.
- El for de la linea 3, itera sobre todas las aristas (M) hasta encontrar una que no este usada en el AGM. Luego tiene complejidad O(M).
- El for de la linea 14, forma el anillo iterando los padres de cada uno de los nodos incidentes a la arista_extra. A lo sumo iterará todos los nodos del grafo (en el caso en que el grafo sea un C_n). Luego tiene complejidad O(N)
- El for de la linea 25 itera todos los nodos del grafo, chequeando en O(1) si perteneces al anillo o no, y los va separando. La complejidad nos queda O(N).

Luego, la complejidad total nos queda:

$$O(M * \log M) + O(M) + O(N) + O(N) = O(M * \log M) \leq O(N^2 * \log N^2) < N^3$$

3.6 Testing

3.6.1 Conexo sin ciclos

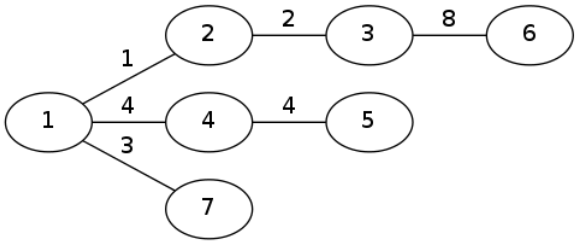

Figure 15: Grafo conexo sin ciclos

La entrada en sí es un arbol, al no tener ciclos no podremos encontrar un anillo de servidores. El algoritmo deja de ejecutar al darse cuenta que M < N y devuelve −1.

3.6.2 No conexo con ciclos

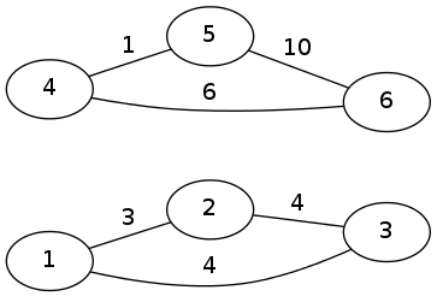

Figure 16: Grafo no conexo con ciclos

Si bien M ≥ N, al ser el grafo no conexo, no podremos encontrar un camino de enlaces de cada computadora a un servidor.
El algoritmo buscar_AGM devolverá −1 al extraer del heap un nodo con distancia == Infinito.

3.6.3 Cn

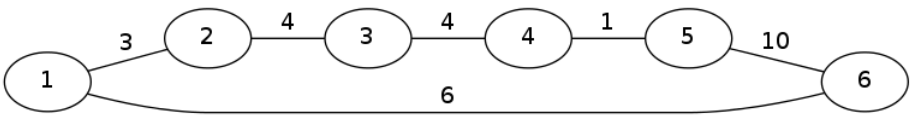

Figure 17: Grafo C_6

Caso borde donde todos equipos deben ser utilizados como servidores para que haya solución

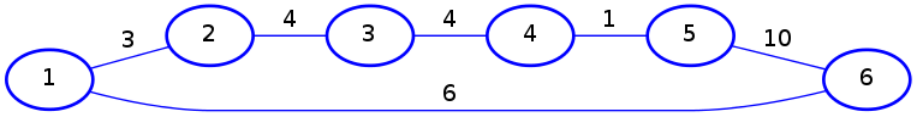

Figure 18: Salida grafo C_6

3.6.4 Todos los enlaces con el mismo coste

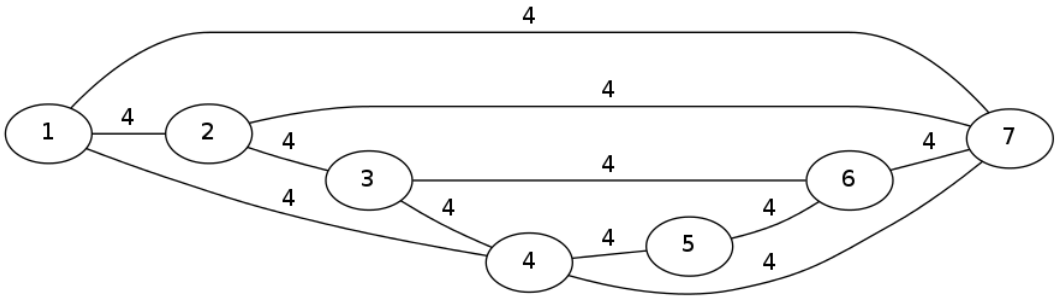

Figure 19: Todos los enlaces con el mismo coste

Caso borde donde todos los enlaces tienen el mismo costo de utilización.
Todas las combinaciones de N aristas son soluciones óptimas.

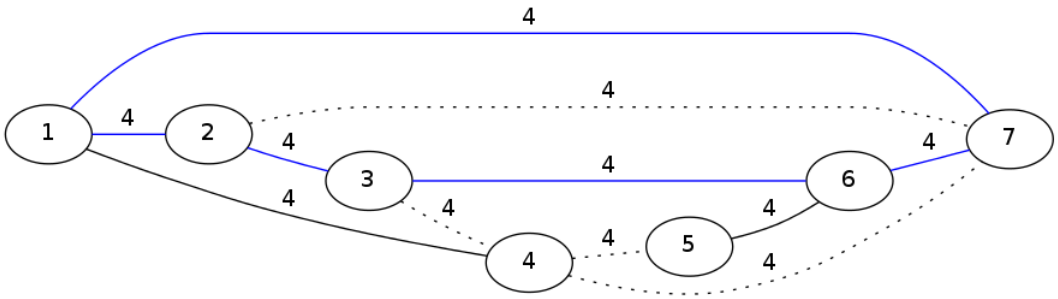

Figure 20: Salida todos los enlaces con el mismo coste

3.6.5 Caso general

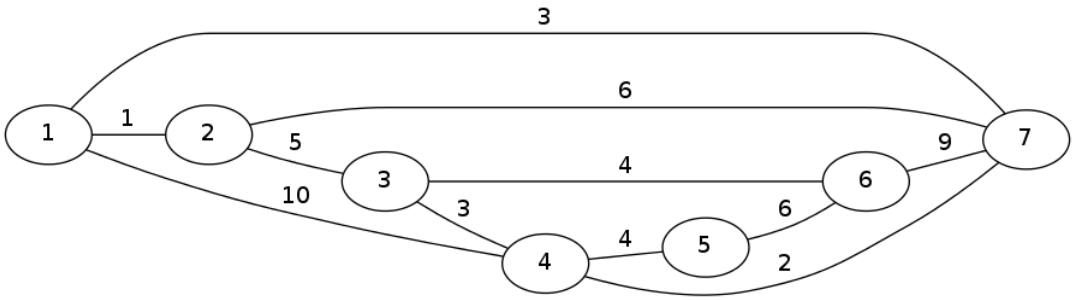

Figure 21: Ejemplo de red con 8 computadores y 10 enlaces con costos asociados

Un caso más general, el mismo grafo que en el punto anterior pero costos de enlace variables.

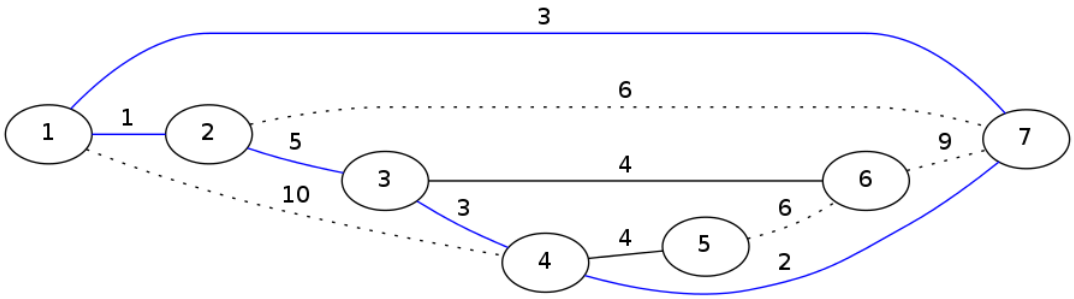

Figure 22: Ejemplo de red con 8 computadores y 10 enlaces con costos asociados

3.7 Experimentación

Una vez hecho el análisis de la complejidad teórica, realizamos experimentos con el fin de contrastar los resultados empíricos y comprobar que el algoritmo implementado efectivamente tendrá una complejidad temporal de $O(n^2)$.
Para los siguientes experimentos se toman una distribución discreta uniforme entre 1 y 100 que se utilizan como valor para el costo de un enlace.

El grueso de nuestra experimentación va a estar dado si el grafo es denso o poco denso para un n fijo. Para reducir los posibles errores de medición y evitar que los resultados se vean alterados por entradas de peor o mejor caso sesgando los resultados, se decidió tomar una cantidad fija de 220 instancias generadas y tomar el promedio. En el siguiente grafico se pueden observar los resultados:

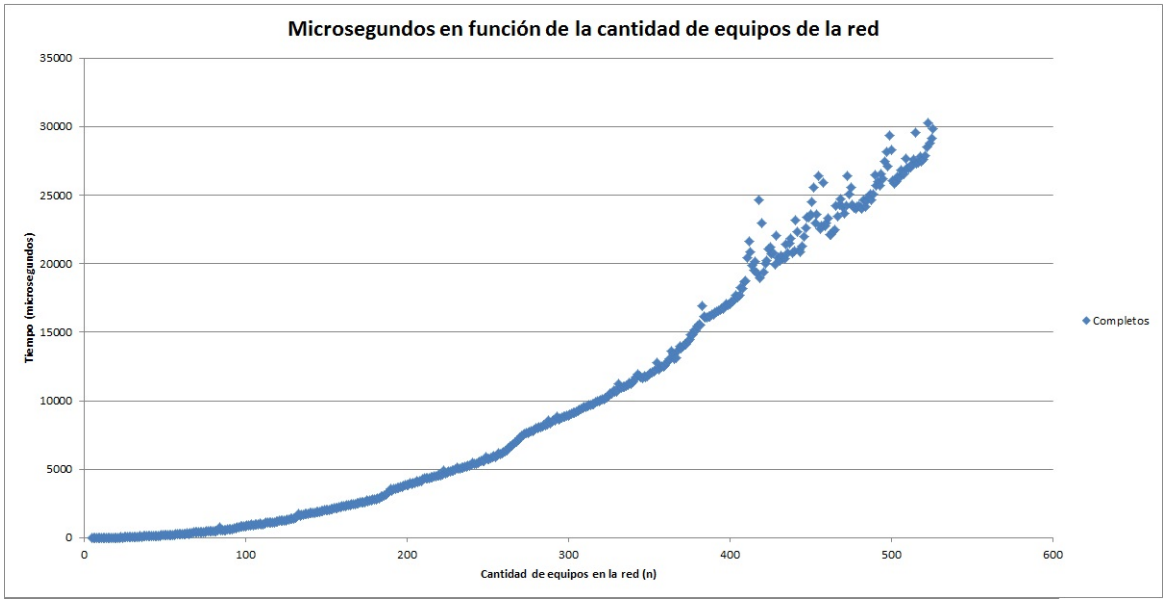

Figure 23

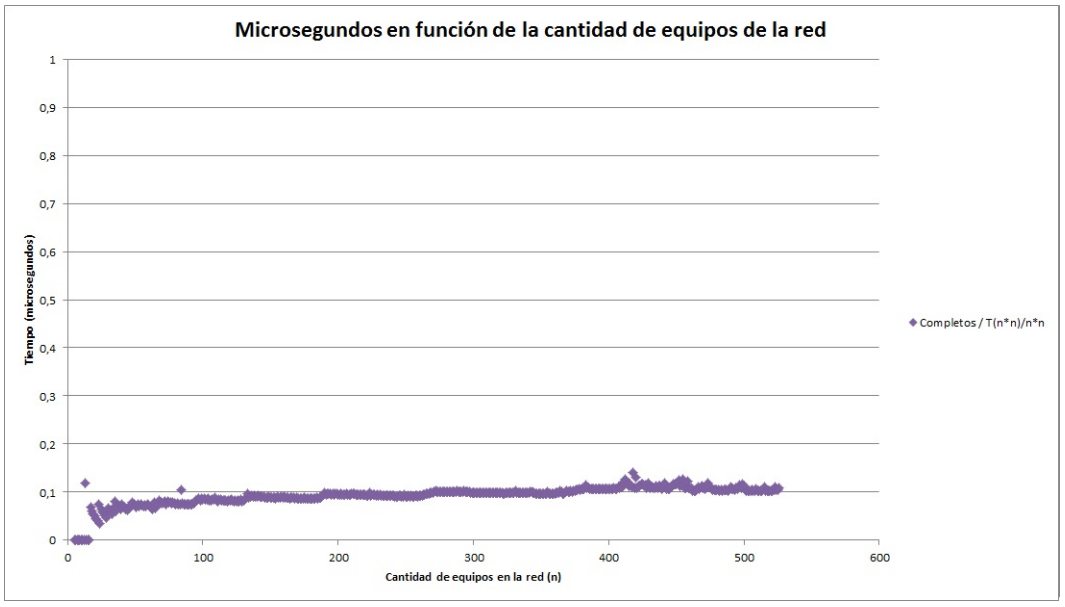

Figure 24

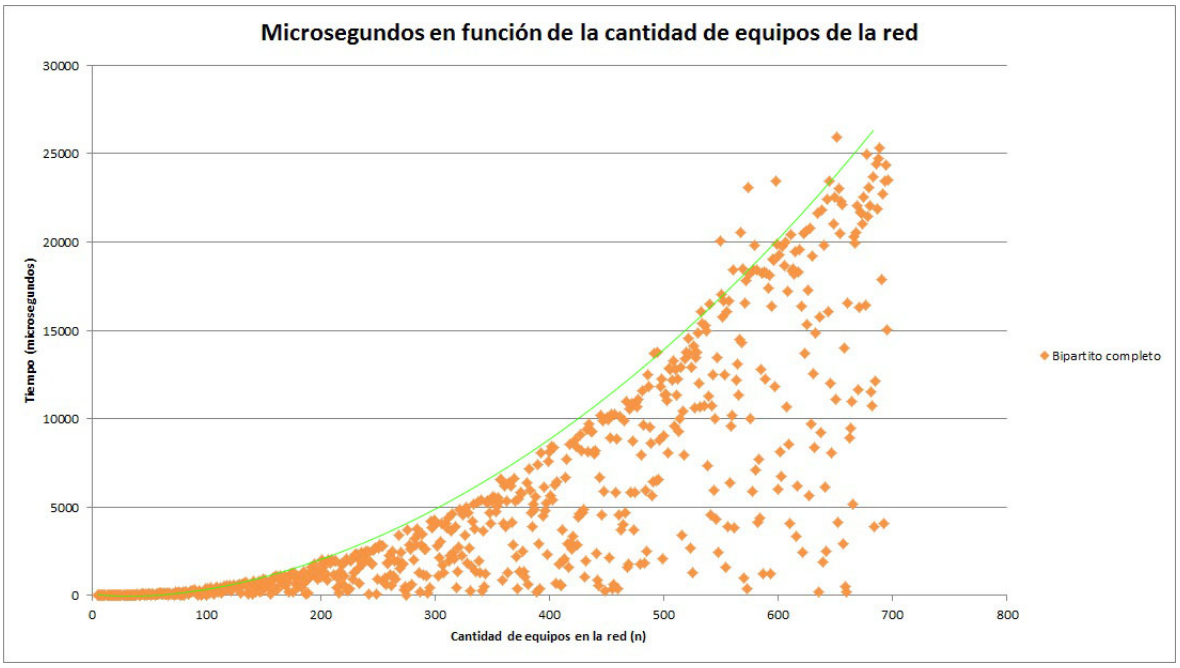

Figure 25

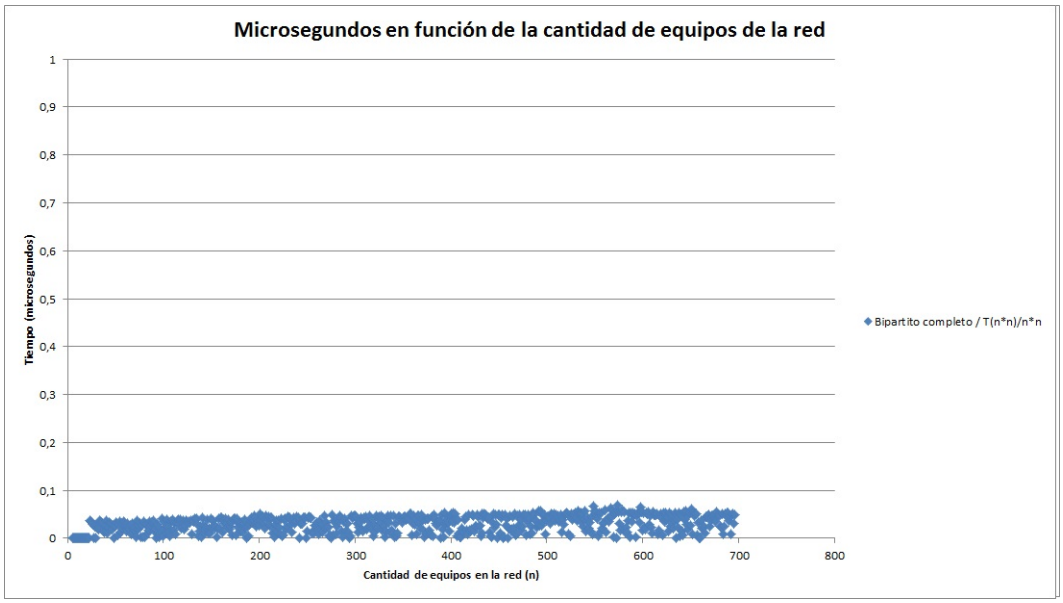

Figure 26

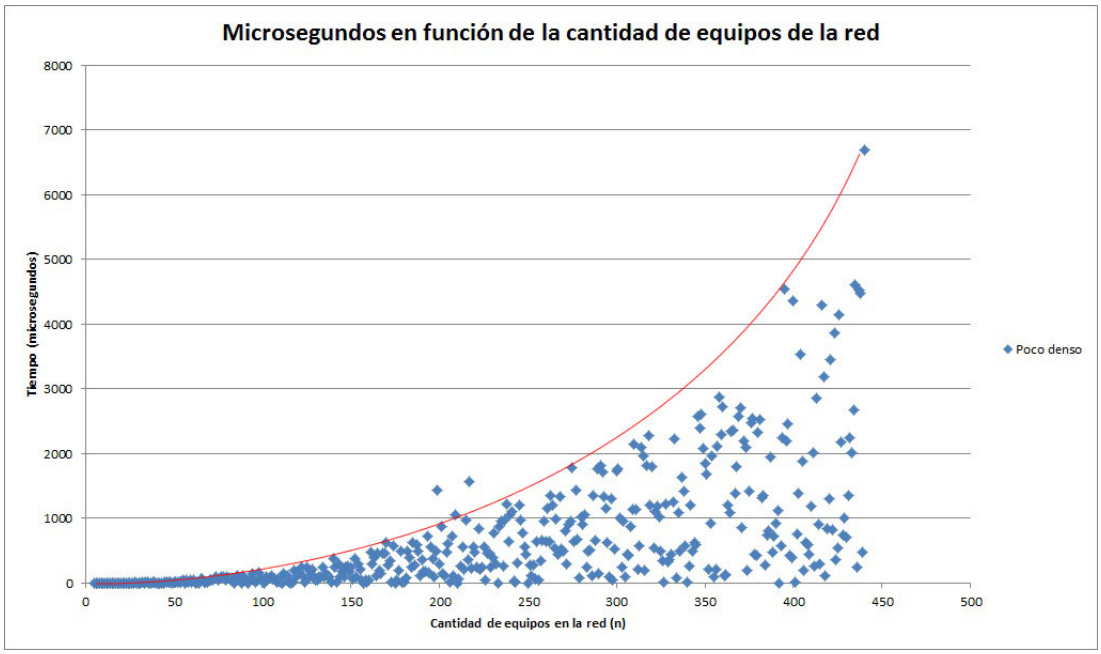

Figure 27

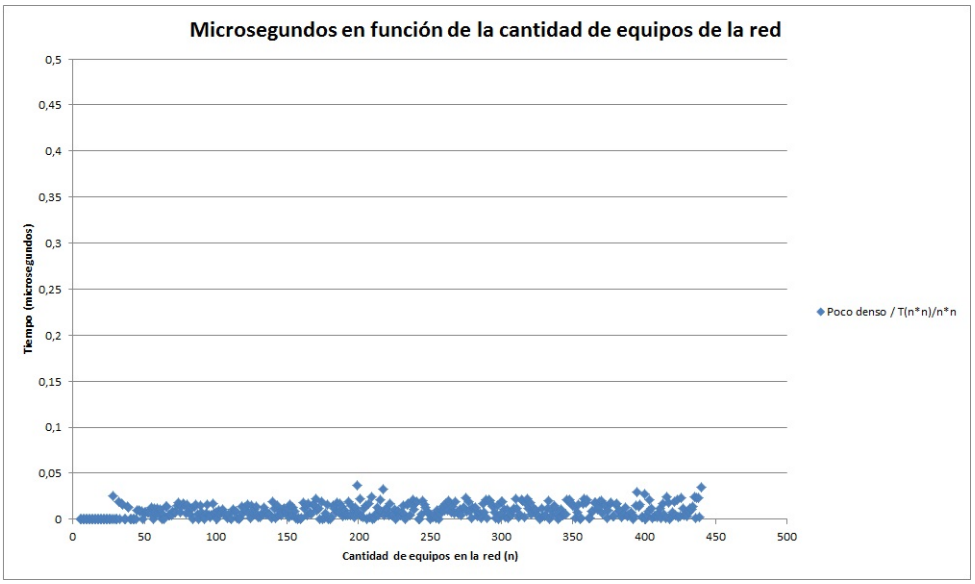

Figure 28

En estos graficos podemos corroborar que efectivamente la complejidad es $O(n^2)$. Ahora vamos a comparar dichas configuraciones de grafos para ver si efectivamente la densidad afecta en la complejidad, no a tal punto de hacerla crecer mas que n^2 pero de tener una constante mas alta.

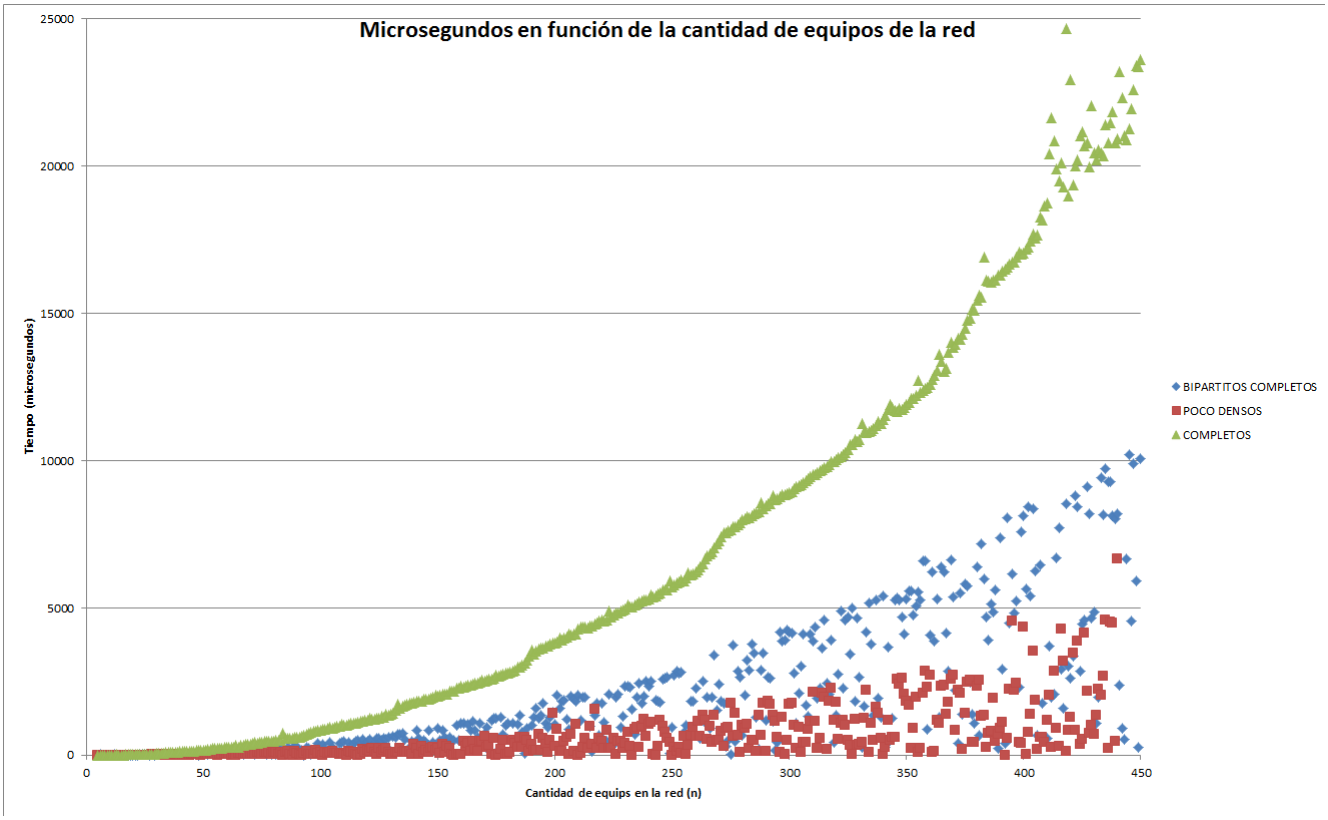

Figure 29

Como el grueso de nuestro algoritmo se basa en buscar un AGM entonces los grafos mas densos van a tener mas cantidad de aristas, haciendo que el algoritmo tarde mas en encontrarlo. Esto se corrobora con los gráficos, como en el gráfico 29 que es notable la diferencia de tiempos entre el mas denso y el menos denso. Podemos concluir que el algoritmo toma mas tiempo al procesar grafos densos.

Podemos concluir que el algoritmo toma mas tiempo al procesar grafos densos.

A Código fuente: Problema 1

.

```
1  size_t dijkstra(size_t origen, size_t destino, adyacencias_t& adyacencias, std::list<size_t>&
       itenerario) {
2
3      std::priority_queue< Tupla > cola;
4
5      std::vector<size_t> distancia(ciudades_.size(), INFINITO);
6      std::vector<size_t> padre(ciudades_.size(), INFINITO);
7      std::vector<size_t> vuelo(ciudades_.size(), INFINITO);
8
9      distancia[origen] = 0;
10     cola.push(Tupla(origen, 0));
11
12     while (!cola.empty()) {
13         Tupla u =cola.top(); cola.pop();
14         for(auto& v: adyacencias[u.first]) {
15             if (distancia[v.des_] > v.fin_) {
16                 if ((distancia[u.first] == 0) || ((int)distancia[u.first] <= (int)(v.ini_ - 2)
                       )) {
17                     distancia[v.des_] = v.fin_;
18                     padre[v.des_] = u.first;
19                     vuelo[v.des_] = v.id_;
20                     cola.push(Tupla(v.des_, distancia[v.des_]));
21                 }
22             }
23         }
24     }
25
26     if (distancia[destino] != INFINITO) {
27         size_t ciudad = padre[destino];
28         itenerario.push_front(vuelo[destino]);
29         while (ciudad != origen && ciudad != INFINITO) {
30             itenerario.push_front(vuelo[ciudad]);
31             ciudad = padre[ciudad];
32         }
33         // Si la ultima ciudad no es el origen, entonces no hay itinerario.
34         if (ciudad == INFINITO) {
35             itenerario.clear();
36             return INFINITO;
37         }
38     }
39     return distancia[destino];
40 }
```

B Código fuente: Problema 2

```
1  class Tablero {
2   /// Casillas
3   int n_;
4   /// Cantidad de caballos
5   int k_;
6   /// Casilla (f, c) elegida como destino final
7   int f_;
8   int c_;
9   /// Cantidad de movimientos totales necesarios para llevar a todos los caballos a la casilla
        (f, c)
10  int m_;
11
12  typedef pair<int, int> coord_t;
13  typedef vector<vector<int> > matriz_t;
14  vector<coord_t> caballos_;
15
16  inline bool esValido(const coord_t& c, int fila, int col) const {
17   return ((c.first+fila <= n_) && (c.first+fila > 0) && (c.second+col <= n_) && (c.second+col
         > 0));
18  }
19
20  inline coord_t sumar(const coord_t& c1, const coord_t& c2) const {
21   return coord_t(c1.first + c2.first,  c1.second + c2.second);
22  }
23
24  void dameAdyacentes(const coord_t& c, list<coord_t>& adyacentes) const {
25   // movimiento de caballo arriba derecha
26   if (esValido(c, 1, 2))
27    adyacentes.push_back(sumar(c, coord_t(1, 2)));
28   // movimiento caballo arriba izquierda
29   if (esValido(c, -1, 2))
30    adyacentes.push_back(sumar(c, coord_t(-1, 2)));
31   // movimiento de caballo derecha abajo
32   if (esValido(c, 2, -1))
33    adyacentes.push_back(sumar(c, coord_t(2, -1)));
34   // movimiento caballo derecha arriba
35   if (esValido(c, 2, 1))
36    adyacentes.push_back(sumar(c, coord_t(2, 1)));
37   // movimiento caballo abajo derecha
38   if (esValido(c, 1, -2))
39    adyacentes.push_back(sumar(c, coord_t(1, -2)));
40   // movimiento caballo abajo izquierda
41   if (esValido(c, -1, -2))
42    adyacentes.push_back(sumar(c, coord_t(-1, -2)));
43   // movimiento caballo izquierda arriba
44   if (esValido(c, -2, 1))
45    adyacentes.push_back(sumar(c, coord_t(-2, 1)));
46   // movimiento caballo izquierda abajo
47   if (esValido(c, -2, -1))
48    adyacentes.push_back(sumar(c, coord_t(-2, -1)));
49  }
50
51  void bfs(coord_t origen, matriz_t &m){
52   queue<coord_t> cola;
53
54   vector<bool> aux(n_, false);
55   vector<vector<bool > > visitados(n_, aux);
56   cola.push(origen);
57   visitados[origen.first-1][origen.second-1]  = true;
58
59   while(!cola.empty()){
60    coord_t cab = cola.front(); cola.pop();
61
62    list<coord_t> adyacentes;
63    dameAdyacentes(cab, adyacentes);
64
65    for(auto &v: adyacentes){
66     if (visitados[v.first-1][v.second-1] == false){
67      visitados[v.first-1][v.second-1] = true;
68      m[v.first-1][v.second-1] = m[cab.first-1][cab.second-1] + 1;
```

```
69
70        cola.push(v);
71       }
72      }
73     }
74    }
75
76
77    public:
78     Tablero(): n_(0), k_(0), f_(0), c_(0), m_(numeric_limits<int>::max()) {}
79
80     void resolver(){
81      // Para acumular la cantidad de movimientos
82      matriz_t acumMovimientos(n_, vector<int>(n_, 0));
83      // Para contar si cada caballo pas\'o por cada posici\'on (tienen que estar los k).
84      matriz_t cantidad(n_, vector<int>(n_, 0));
85
86      // Ejecuto k bfs (para cada caballo)
87      for(int cab = 0; cab < k_; ++cab) {
88       matriz_t movimientos(n_, vector<int>(n_, 0));
89
90       bfs(caballos_[cab], movimientos);
91
92       // Una vez que termino el BFS, acumulo los movimientos.
93       for (int i = 0; i < n_; ++i) {
94        for (int j = 0; j < n_; ++j) {
95         acumMovimientos[i][j] += movimientos[i][j];
96         // Ahora verifico que el caballo haya estado en esta posici\'on
97         if (movimientos[i][j] > 0)
98          ++cantidad[i][j];
99         else {
100         if ( (caballos_[cab].first-1 == i) && (caballos_[cab].second-1 == j))
101          ++cantidad[i][j];
102       }
103      }
104     }
105    }
106
107    // Busco la casilla que tenga la m\'inima cantidad de movimientos y todos los caballos
         lleguen a esa casilla.
108    m_ = numeric_limits<int>::max();
109    for (int i = 0; i < n_; ++i) {
110     for (int j = 0; j < n_; ++j) {
111      if (cantidad[i][j] == k_) {
112       if (acumMovimientos[i][j] < m_) {
113        m_ = acumMovimientos[i][j];
114        f_ = i+1;
115        c_ = j+1;
116       }
117      }
118     }
119    }
120
121   }
122
123   void clear() {
124     f_ = 0;
125     c_ = 0;
126     m_ = numeric_limits<int>::max();
127   }
128
129   int size_n() const {
130    return n_;
131   }
132
133   int size_k() const {
134    return k_;
135   }
136
137  };
```

C Código fuente: Problema 3

```
1  class Nodo{
2  public:
3      int numero, distancia;
4      bool visitado, en_anillo;
5      Nodo * padre;
6      vector <pair <Nodo*, int> > adyacentes;
7      vector <int> arista_en_AGM;
8
9      Nodo(int numero, int N): numero(numero){
10         padre = NULL;
11         distancia = numeric_limits<int>::max();
12         visitado = false;
13         en_anillo = false;
14         arista_en_AGM.resize(N+1);
15     }
16
17     void agregarAdyacente(Nodo* adyacente, int costo) {
18         adyacentes.push_back(make_pair(adyacente, costo));
19     }
20 };
21
22
23 struct Comp {
24     bool operator()(const Nodo* s1, const Nodo* s2) {
25         return (s1->distancia > s2->distancia);
26     }
27 };
28
29 class LCdA{
30 public:
31  int N, M, C;
32  vector<Nodo> nodos;
33     vector<pair < int , pair <Nodo*, Nodo* > > > aristas ;
34     vector<pair<Nodo*, Nodo*> > anillo;
35     vector<pair<Nodo*, Nodo*> > resto;
36
37  LCdA():C(0){}
38  int buscar_AGM() {
39
40         // Si M < N no puede ser conexo y tener un ciclo
41         if(M < N)
42             return 1;
43
44         vector <Nodo*>  heap;
45         for (int i = 1; i <= N; i++)
46             heap.push_back(&nodos[i]);
47
48         Nodo * u = heap[0];
49         u->distancia = 0;
50
51         while (!heap.empty()) {
52             Nodo * u = heap.front(); pop_heap(heap.begin(), heap.end(), Comp());
53             heap.pop_back();
54
55             // Distancia es infinito <==> es no conexo
56             if(u->distancia == numeric_limits<int>::max())
57                 return 1;
58
59             // Anoto la arista como parte del AGM
60             if (u->padre != NULL)        {
61                 u->arista_en_AGM[u->padre->numero] = 1;
62                 u->padre->arista_en_AGM[u->numero] = 1;
63             }
64
65             C+= u->distancia;
66             u->visitado = true;
67
68             for (unsigned int i = 0; i < u->adyacentes.size(); i++) {
69                 Nodo * v = u->adyacentes[i].first;
70                 if (!v->visitado && v->distancia > u->adyacentes[i].second) {
```

```
71                      v->padre = u;
72                      v->distancia = u->adyacentes[i].second;
73                  }
74              }
75              make_heap(heap.begin(), heap.end(), Comp());
76          }
77          return 0;
78      }
79
80    void formar_anillo() {
81          sort(aristas.begin(), aristas.end());
82          pair<Nodo*, Nodo*> arista_extra;
83
84          for (unsigned int i = 0; i < aristas.size(); i++) {
85              if(aristas[i].second.first->arista_en_AGM[aristas[i].second.second->numero] == 0)
                    {
86                  C += aristas[i].first;
87                  arista_extra = aristas[i].second;
88                  break;
89              }
90          }
91
92          Nodo * a = arista_extra.first;
93          Nodo * b = arista_extra.second;
94
95          // Formo el anillo llendo desde cada uno de los nodos de la arista extra hacia atras
                hasta llegar al otro nodo o a NULL
96          for (int i = 0; i < 2; i++) {
97              while(a != NULL && a != b) {
98                  a->en_anillo = true;
99                  a = a->padre;
100             }
101             a = arista_extra.second;
102             b = arista_extra.first;
103         }
104
105         anillo.push_back(arista_extra);
106
107         // Separo las aristas del resto de las del anillo
108         for (int i = 1; i <= N; i++) {
109             Nodo* k = &nodos[i];
110             if (k->padre != NULL) {
111                 if (k->en_anillo)
112                     anillo.push_back(make_pair(k, k->padre));
113                 else
114                     resto.push_back(make_pair(k, k->padre));
115             }
116         }
117     }
118
119 };
```