Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2015

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

Trabajo practico 1

Integrante	LU	Correo electrónico
Integrante Ignacio Rodriguez	797/13	igna_nacho286@hotmail.com
Integrante Matias Visgarra		matias_kingthor@hotmail.com
Integrante Nicolás Mastropasqua	828/13	me.nicolas7@gmail.com
Integrante Fermín Travi	234/13	fermintravi@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Problema 1: Telégrafo

1.1 Descripción del problema

El problema del telégrafo consiste en conectar mediante un cable de telégrafo a las estaciones de los ramales del sistema férreo que se encuentra en el país, sabiendo que cada ramal dispone de una cierta cantidad de cable para conectar sus estaciones, cuya longitud se mide en kilómetros.
Lo que buscamos conseguir es que dado un ramal con sus estaciones y los kilómetros de cable de los que dispone, saber cual es el mayor número de estaciones seguidas que se pueden conectar con dicho cable. Para esto contamos con los ramales y sus estaciones, la distancia en kilómetros de cada una de las estaciones a la capital, que se encuentra en el kilómetro 0, y la cantidad de cable de la que dispone cada ramal.
En los datos de entrada, cada ramal ocupa dos lineas. La primera representa los kilómetros de cable de los que dispone dicho ramal, y la segunda las distancias de las estaciones del ramal a la capital.
Como datos de salida, el algoritmo encargado de resolver el problema debe devolver una linea por cada ramal, donde cada linea indica la cantidad máxima de estaciones de dicho ramal que se pueden conectar con el cable de telégrafo.

Ejemplos:

1. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 6 8 12 15.
   En este ejemplo la máxima cantidad de estaciones que puedo conectar con 6 kilómetros de cable es 3, las cuales son las que se encuentran en el kilómetro 6, 8 y 12, ya que desde las estaciones que se encuentran en el kilómetro 0 y el 8 solo puedo conectar 2, las que se encuentran en el kilómetro 0 y 6 para la primera, y las que se encuentran en el kilómetro 8 y 12 para la segunda.
2. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 7 14 21 28.
   En este caso, la distancia entre cada una de las estaciones es mayor que la cantidad de kilómetros de cable del que se dispone, por lo que la solución de este caso es que hay 0 estaciones que se pueden conectar mediante el cable.
3. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 3 6 9 12 15 18.
   En este último caso las distancias entre todas las estaciones es la misma, y con los kilómetros de cable de los que disponemos podemos conectar como máximo hasta 3 estaciones.

1.2 Resolución

Lo que nuestro algoritmo hace primero es buscar, en un vector con las distancias de las estaciones del ramal a la capital llamado *distanciaDeLasEstaciones*, la máxima cantidad de estaciones que puedo conectar con la capital, lo cual me va a servir como cota mínima de la cantidad máxima de estaciones que puedo conectar.
Para esto recorro el vector linealmente hasta encontrar la última estación alcanzable desde la capital con el cable. El valor resultante lo guardo en *maximaCantidadDeEstaciones*, que va a servir para almacenar la máxima cantidad de estaciones conectables con el cable de telégrafo.
Luego, utilizando dos índices, *inicio* y fin, vuelvo a recorrer *distanciaDeLasEstaciones* para encontrar la máxima cantidad de estaciones que voy a poder conectar con el cable. Los índices *inicio* y fin van a representar la posición de las estaciones dentro del vector *distanciaDeLasEstaciones*, y van a comenzar inicializados con valores 0 y 1 respectivamente.
La manera en la que voy ir recorriendo dicho vector es ir aumentando el valor de fin hasta que la cantidad de estaciones entre *inicio* y fin supere el valor de *maximaCantidadDeEstaciones* o me quede sin estaciones para recorrer.
En caso de que suceda lo primero reemplazo el contenido de *maximaCantidadDeEstaciones* por la cantidad de estaciones entre *inicio* y fin(sin contar la última estación a la que el cable no llega), aumento en uno el valor de *inicio*, y vuelvo a comenzar a aumentar el valor de fin.
Por otro lado, como *inicio* no puede ser mayor que fin, si cuando tengo que aumentar el valor de *inicio* este iguala al valor de fin, en ves de aumentar únicamente *inicio*, aumento el valor de ambos

índices en uno.
Cuando me quede sin estaciones para recorrer devuelvo el valor de *maximaCantidadDeEstaciones* como resultado del problema.

Para finalizar, a continuación mostramos un pseudocódigo de lo explicado anteriormente.

Algorithm 1: Telégrafo

```
Input: int kilometrosDeCable, vector< int> distanciaDeLasEstaciones
int inicio ← 0
int fin ← 1
int cantidadDeEstaciones ← 0
int maximaCantidadDeEstaciones ← última estación alcanzable desde la capital con el cable
mientras fin < total de estaciones del ramal
    Si puedo unir con cable las estaciones entre inicio y fin
        cantidadDeEstaciones ← La cantidad de estaciones entre inicio y fin
        Si cantidadDeEstaciones > maximaCantidadDeEstaciones
            maximaCantidadDeEstaciones ← cantidadDeEstaciones
        Aumento el valor de fin en 1
    Si no puedo unir las estaciones entre inicio y fin con el cable
        Si fin es igual a inicio+1 aumento ambos índices en 1
        Si no, aumento únicamente inicio en 1
Devuelvo el valor almacenado en maximaCantidadDeEstaciones
```

1.3 Justificación

Como se menciona en el algoritmo explicado en el punto anterior, se van a recorrer las estaciones del ramal, las cuales se encuentran en el vector *distanciaDeLasEstaciones*, utilizando los índices *inicio* y fin.
Se va a ir aumentando el valor del índice fin mientras pueda seguir conectando las estaciones con el cable, y cuando no pueda conectarlas se va a ir aumentando el índice *inicio* hasta poder volver a hacerlo. Eso quiere decir que por cada valor que toma el índice *inicio* se va a buscar la última estación que se pueda conectar con el cable a partir de dicho índice, actualizando el valor de *maximaCantidadDeEstaciones* si la cantidad de estaciones supera su valor.
Por ende, como el valor de *maximaCantidadDeEstaciones* solo almacena la máxima cantidad de estaciones conectables por el cable de telégrafo, y desde cada estación marcada con el índice *inicio* calculo la máxima cantidad de estaciones que puedo conectar con el cable desde dicha estación, *maximaCantidadDeEstaciones* va a terminar almacenando el valor de la máxima cantidad de estaciones desde alguna estación, es decir, la respuesta al problema.

1.4 Complejidad

Llamemos n a la cantidad de estaciones. Como lo que hace el algoritmo es recorrer con dos índices el vector de estaciones de tamaño n, y ninguno de los dos índices decrece, al final lo que termina haciendo es recorrer el vector de estaciones hasta el final con cada índice. La complejidad de recorrer un vector dos veces es $O(2n)$, que es lo mismo que $O(n)$.
Por otro lado, a parte de recorrer el vector de estaciones, lo que hace el algoritmo es hacer acceder a dicho vector mientras lo recorre y hacer cuentas entre las estaciones ubicadas en los índices *inicio* y fin del vector para saber si se pueden conectar mediante el cable de telégrafo, actualizar el valor de *maximaCantidadDeEstaciones*, y aumentar el valor de los índices, todo lo cual tiene una complejidad $O(1)$.
Por todo lo anterior, la complejidad final del algoritmo es $O(n)$.

1.5 Casos de Test

Para corroborar el correcto funcionamiento del algoritmo, este mismo de probo con el siguiente conjunto de tests.

1. **Kilómetros de cable:** 5. **Distancia de las estaciones:** 6 12 18 24
   Este primer test se utilizó para corroborar que el algoritmo devolviera 0 en caso de que no se pudiera conectar ninguna estación mediante el cable, dado que la distancia entre cada una de las estaciones es mayor que la cantidad de kilómetros de cable disponible.
2. **Kilómetros de cable:** 0. **Distancia de las estaciones:** 3 8 12 17
   Este test se utilizó para corroborar que si la cantidad de cable disponible para conectar las estaciones es 0, el algoritmo devolviera 0, ya que sin cable las estaciones no se pueden conectar.
3. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 6 8 12 15
   Este test, dado por la cátedra, se utilizó para corroborar el correcto funcionamiento del algoritmo dado una instancia normal, es decir, una instancia sin ningún caso especial.
4. **Kilómetros de cable:** 34. **Distancia de las estaciones:** 6 10 22 34
   Este test sirvió para corroborar que si el cable alcanzara para conectar todas las estaciones, el algoritmo devolviera la cantidad total de estaciones, incluida la de la capital, como resultado.
5. **Kilómetros de cable:** 5. **Distancia de las estaciones:** 5 10 15 20 25
   Por último, este test se utilizó para corroborar que si la distancia entre todas las estaciones fuera igual a la cantidad de cable, el algoritmo devolviera 2 como resultado, ya que solo dos estaciones se pueden conectarse mediante el cable.

1.6 Tiempos de ejecución y gráficos

Para poder observar la performance en términos de tiempo del problema, se corrieron los siguientes casos de tests, variando en cada uno la cantidad de estaciones del ramal, la cantidad de cable de telégrafo disponible para conectar las estaciones, y la distancia entre dichas estaciones.

1. En el primer test utilizado para observar el tiempo de ejecución del problema, la cantidad de cable que se utilizó para poder conectar las estaciones fue de 200.
   El mismo test se corrió 1000 veces, en donde en cada iteración la cantidad total de estaciones iba aumentando en 100, es decir, comienza con 100 estaciones en la primera iteración del test, y para la última iteración termina con 100000 estaciones. Luego, para cada una de las iteraciones del test, la distancia entre cada una de las estaciones se mantuvo en 10.
   El figura 1 muestra el resultado de dicho test.

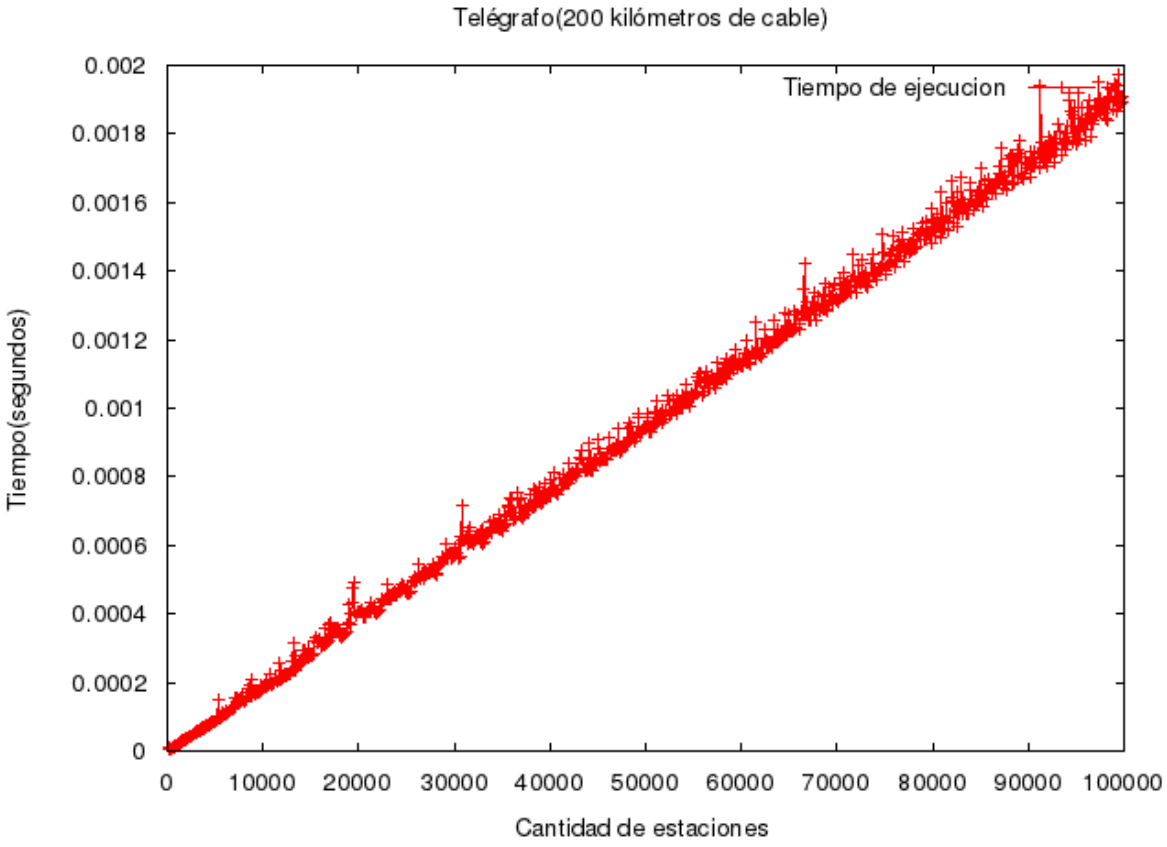

Figure 1: Primer test

2. Para el segundo test se mantuvieron los mismos datos que para el primero, a excepción de la cantidad de cable, que se disminuyo a 50.
   Como puede verse en la figura 2, el tiempo que tardo en resolver el mismo problema pero con una cantidad de cable menor es el mismo, con lo que podemos afirmar que la cantidad de cable utilizada para conectar las estaciones no influye en el tiempo, teniendo en cuenta que la distancia entre estaciones es menor que la cantidad de cable disponible.

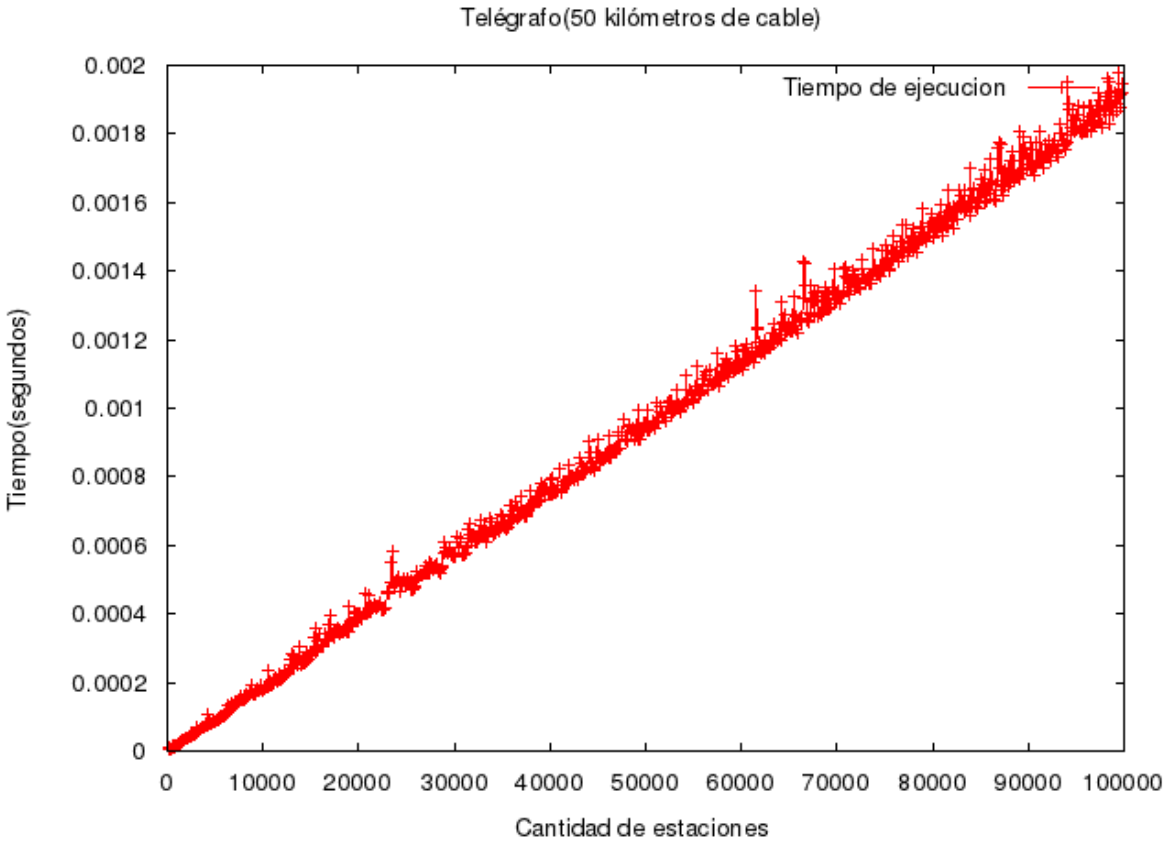

Figure 2: Segundo test

3. Ahora, en el tercer test, para poder observar si la distancia entre las estaciones influye en el tiempo de ejecución del problema, se corrieron las mismas instancias que en el primer test, salvo que ahora la distancia entre cada una de las estaciones se hizo de 220, de manera que la cantidad de cable no alcanzara para conectar ninguna estación.
   En la figura 3 puede observarse que el tiempo de ejecución de dicha instancia decreció respecto de lo que tardo en el primer test. Esto sucede porque cada vez que el algoritmo no puede conectar una estación con la siguiente mediante el cable pasa a la siguiente estación para corroborar lo mismo. Esto resulta en que el arreglo de estaciones se recorra una única vez, en vez de un total de 2 veces si siempre encuentra una estación para poder conectar mediante el cable.

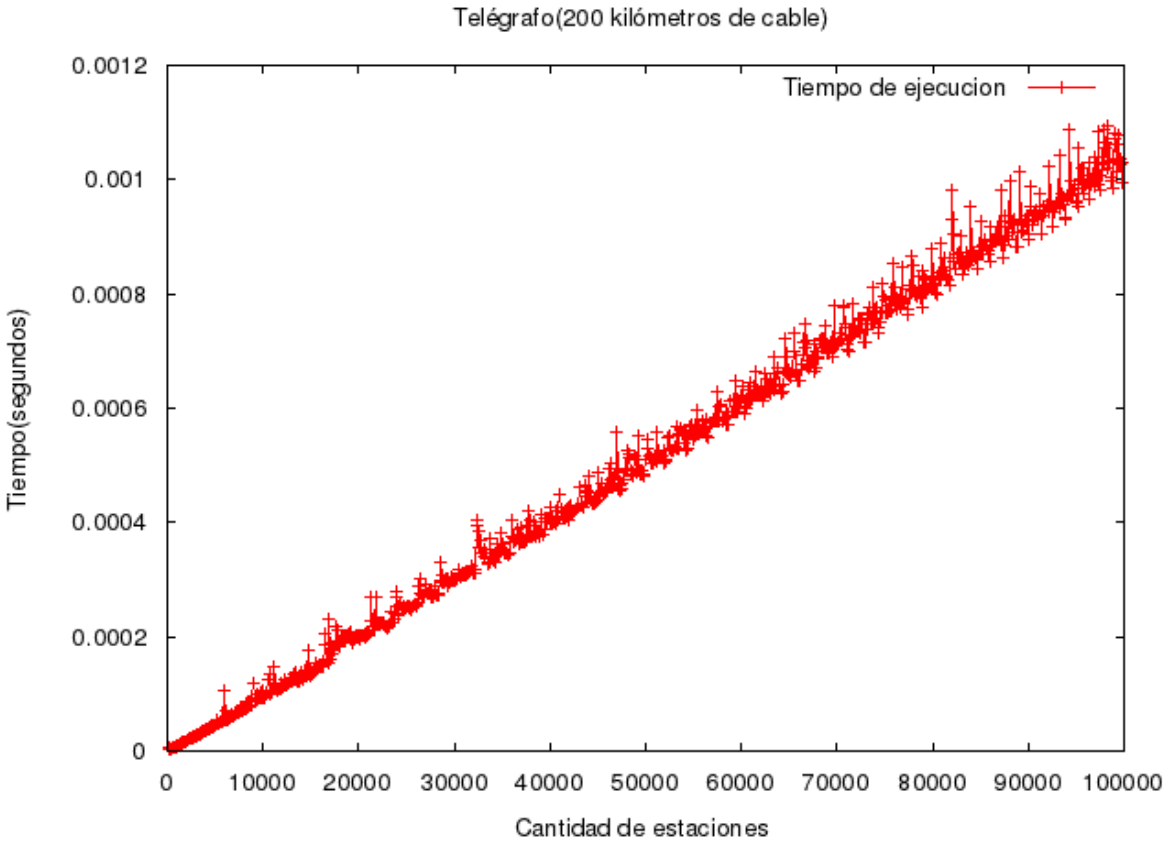

Figure 3: Tercer test

4. El siguiente test se utilizó para corroborar si la cantidad de estaciones efectivamente influye en el tiempo de ejecución. En este caso, se tomaron los mismos valores que en el primer test, pero se aumento la cantidad de iteraciones del test a 30000, es decir, que para la última iteración del test la cantidad de estaciones se hace de 300000.
   En la figura 4 se aprecia claramente que, como era de esperarse, mientras mas estaciones haya que revisar, mas tiempo le toma al algoritmo encontrar la solución al problema.

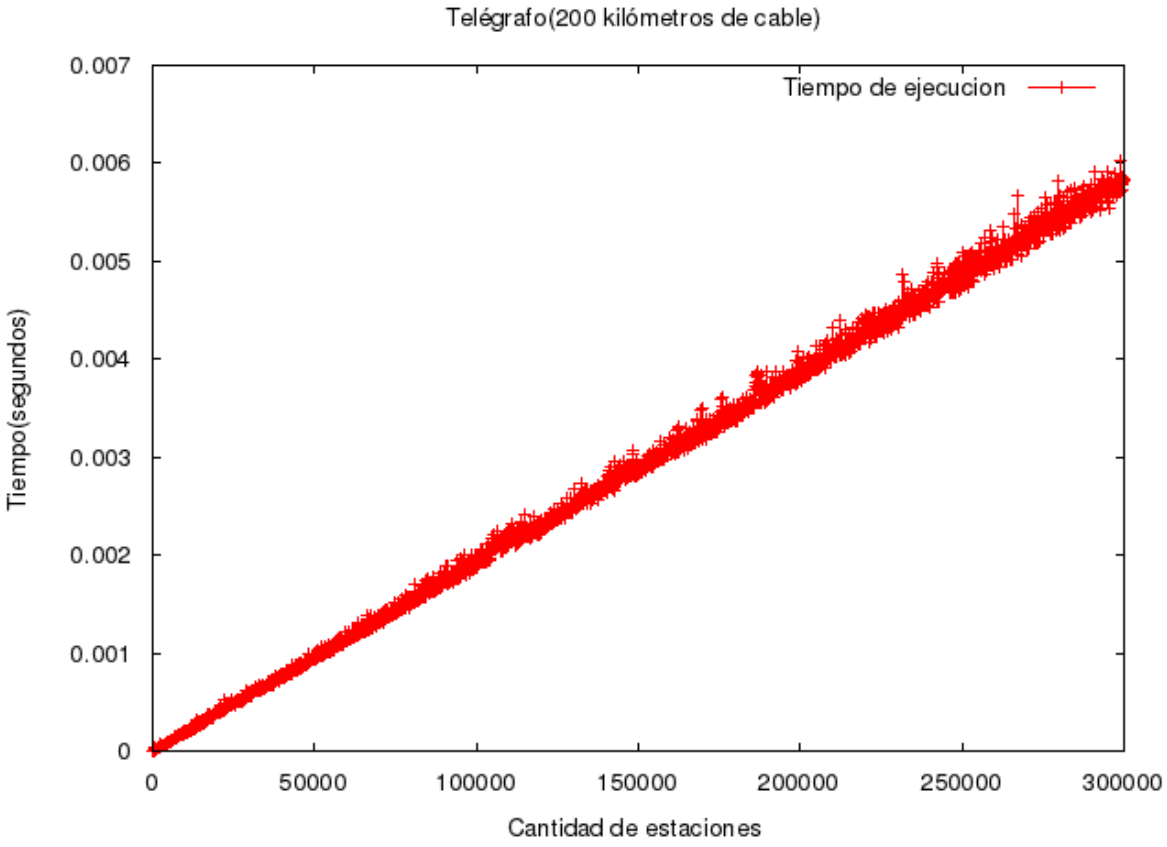

Figure 4: Cuarto test

5. Para terminar, el siguiente test fue utilizado para corroborar que la complejidad del algoritmo fuera la esperada. Para eso, se corrieron la misa cantidad de instancias que para el test 1 y con los mismos parámetros a excepción de la distancia entre cada una de las estaciones. En este caso cada estación esta separada aleatoriamente entre 100 y 200 kilómetros de la siguiente estación. Luego, a los tiempos resultantes de dicha experimentación se los dividió por el tamaño de la entrada, es decir, la cantidad de estaciones en cada iteración del test.
   Las figuras 5 y 6 muestran el tiempo de ejecución de correr dicho test, y el resultado de dividir los tiempos obtenidos por el tamaño de la entrada, y como puede observarse en la figura 6, la función de tiempo sobre tamaño de entrada tiende a una constante, por lo que podemos asumir que la complejidad del algoritmo es, según lo esperado, $O(n)$.

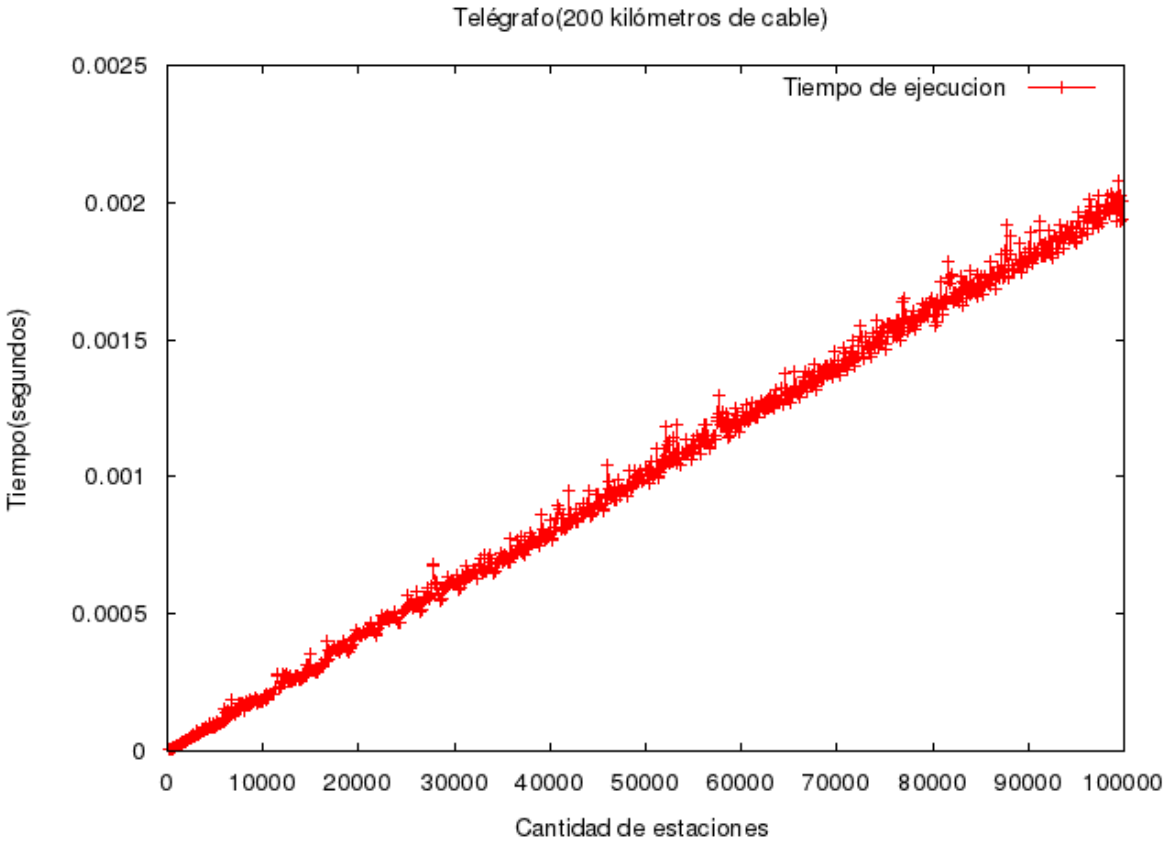

Figure 5: Quinto test

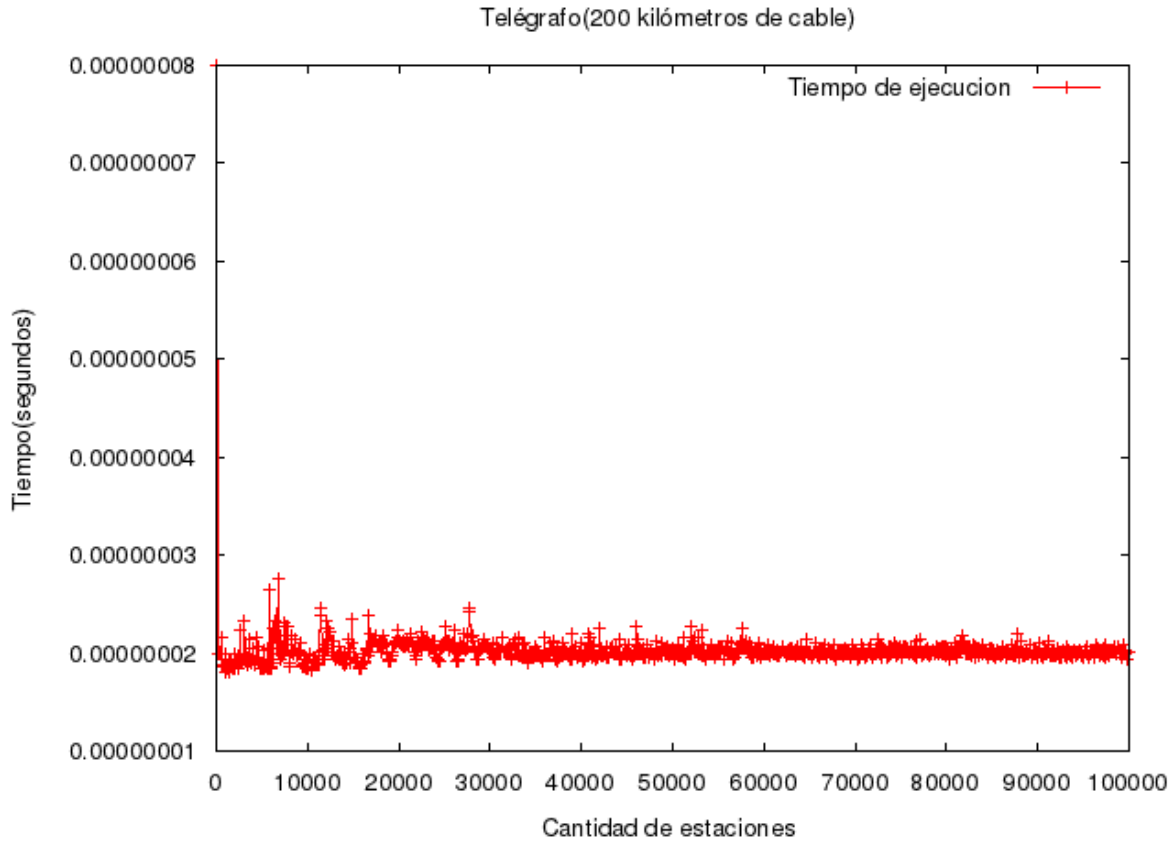

Figure 6: Quinto test comparando con tamaño de la entrada

1.7 Partes relevantes del Código

El siguiente algoritmo es el encargado de encontrar la máxima cantidad de estaciones que se pueden conectar mediante el cable de telégrafo proporcionado al ramal.

```
void calcularMaximaCantidadDeEstaciones(int kilometrosDeCable,vector<int> distanciaDeLasEstaciones

    int inicio=0;
    int fin=1;
    int cantidadDeEstaciones=0;
    int maximaCantidadDeEstaciones=0;
    int j=0;

    while(distanciaDeLasEstaciones[j]<=kilometrosDeCable)
    {
         maximaCantidadDeEstaciones=j+2;
         j++;
    }
    while(fin<distanciaDeLasEstaciones.size()){

         if((distanciaDeLasEstaciones[fin]-distanciaDeLasEstaciones[inicio])<=kilometrosDeCable){
             cantidadDeEstaciones=fin-inicio+1;

             if(cantidadDeEstaciones>maximaCantidadDeEstaciones)
                 maximaCantidadDeEstaciones=cantidadDeEstaciones;
```

```
            fin++;
        }
        else{
            if(fin==inicio+1){
                fin++;
                inicio++;
            }
            else{
                inicio++;
            }
        }
    }
    cout<<maximaCantidadDeEstaciones<<endl;
}
```

2 Problema 2: A Medias

2.1 Descripción del problema:

El problema que se nos presentó consistía en, mediante un arreglo ya formado, crear uno nuevo que en la posición i de este nuevo arreglo tuviese la mediana del subarreglo ordenado que se forma del arreglo original entre las posiciones 0 a i.
La mediana para un arreglo de tamaño n se calcula de la siguiente forma: si el arreglo tiene un tamaño impar la mediana es $x_{(n+1)/2}$; en cambio, si el tamaño es par, es $(x_{(n)/2}+x_{(n+1)/2})/2$.

2.2 Resolución del problema:

Para resolver el problema se decidió usar una estructura que consiste de cuatro observadores: dos heaps (un min heap y un max heap), un entero que contiene a la mediana y un entero que representa el tamaño del arreglo.
Para armar el arreglo que se va a devolver, se recorre el arreglo original, se toma el valor que está en la posición i del arreglo original y luego se analiza el tamaño del subarreglo (antes de agregar un nuevo número). En caso de que sea 0, quiere decir que esta vacío, entonces la mediana es el valor que tomamos.
Si el subarreglo tiene un tamaño impar, nos fijamos si el valor es mayor o menor que la mediana. En caso de que sea menor, se agrega el valor al max heap y la mediana al min heap, y si es mayor es al revés. Luego, la mediana es la suma de las raíces de los heaps dividido dos y se le suma 1 al tamaño.
En cambio, si el subarreglo tiene tamaño par, también nos fijamos si el valor es mayor o menor que la mediana. En caso de ser menor, se le agrega el valor al max heap y se desencola la raíz del max heap y ese se transforma en la mediana, pero si es mayor se agrega el valor al min heap y se desencola la raíz del min heap y ese se transforma en la mediana, y se le suma uno al tamaño.
Como dijimos, esto se va a hacer mientras se recorre el arreglo original, entonces cada vez que se agrega un nuevo valor a la estructura nos fijamos cual es la mediana y agregamos la mediana al arreglo que vamos a devolver.

2.3 Justificación del algoritmo:

La idea del algoritmo es que el tamaño de los heaps siempre sea igual y de esa forma poder siempre ver fácilmente la parte que nos importa del subarreglo rápidamente y de una forma ordenada. Como ya se explicó anteriormente, antes de agregar un nuevo valor al subarreglo nos fijamos el tamaño de este. Como caso base si el subarreglo tiene tamaño 0 antes de agregar un número, cuando lo agreguemos va a tener tamaño 1 y en ese caso ese mismo número va a ser la mediana, y los dos heaps tienen tamaño cero.

Ahora suponiendo que ambos heaps tienen el mismo tamaño:
En caso de que antes de agregar un número el tamaño sea impar, cuando lo agreguemos va a ser par, entonces por la fórmula para calcular la mediana tenemos que sumar los dos números que están en el medio y dividirlos por dos, pero antes de hacer eso hay que fijarse si el valor que se va a agregar es mayor o menor que la mediana y a qué heap tendría que ir.
Si el valor es menor que la mediana, se agrega el valor al max heap, que es donde están los valores menores que la mediana, y la mediana se agrega al min heap, que es donde están los valores mayores que la mediana, y de esta forma se mantiene el invariante de que los dos heaps tienen el mismo tamaño, y le mediana nueva se calcula con la fórmula previamente mencionada.
Si en cambio el tamaño es par, cuando lo agreguemos va a ser impar, entonces la mediana va a ser el elemento que está en el medio. Ahora nos fijamos si el valor a agregar es mayor o menor que la mediana, en caso de ser menor lo agregamos al max heap y desencolamos su raíz, y esa va a ser la nueva mediana. En caso de ser mayor se agrega al min heap y luego se desencola la raíz. La raíz desencolada va a ser la nueva mediana, y los tamaño de los heap se mantienen equivalentes.

2.4 Complejidad del algoritmo:

La complejidad de este algoritmo es de *nlog(n / 2)*, que asintóticamente es equivalente a *O(nlogn)*, que es mejor que la cota que nos pedían de n^2, siendo n la cantidad de números que tiene el arreglo original. Esta complejidad se debe a que en cada iteración se realiza uno o dos adiciones a un heap, dependiendo esto del tamaño del subarreglo, y se ven las raíces de los heaps, pero al ser de tiempo constante esta función no suma a la complejidad, haciendo que cada adición al subarreglo sea *log(n)*. Como esto se realiza para todos los números en el arreglo original, esto se realiza n veces, haciendo que la complejidad total sea la dicha.
Se sabe que los métodos de des/encolar son de *O(log n)* y ver la raíz de *O(1)* por la documentación de Java:
Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).
A su vez, el tiempo promedio de agregar un elemento al heap es de *O(1)*.

2.5 Tiempos de ejecución y gráfico

Para poder apreciar cómo varían los tiempos dependiendo del tamaño del input, se realizó el siguiente gráfico que incluye la ejecución del algoritmo con arreglos de distinto tamaño. Para los casos de prueba se utilizaron números pseudoaleatorios en el arreglo, generados mediante la librería *util.Random* de Java. Como se puede observar en el gráfico, se cumple la complejidad de caso promedio para la adición de elementos al heap (que es de $O(1)$), por lo que se obtiene una función casi lineal. El tamaño del input se fue incrementando inicialmente por 50 (de 50 a 1000), luego por 100 (de 1000 a 4000) y finalmente por 500 (de 4000 a 17000).

Para el peor caso, se supuso que Java representa internamente al heap con un árbol binario, por lo que se dispuso a diseñar arreglos que fuercen a producir *log n* intercambios cada vez que se realizaba una adición al heap.
Los arreglos que cumplían con esa propiedad eran aquellos que contenían números en orden creciente en sus posiciones pares y números en orden decreciente en sus posiciones impares (por ejemplo, (3, 89, 7, 62, 13, 50, 19, 43). De esta manera, por cómo se encuentra armado el algoritmo, siempre se agregaban elementos mayores que la raíz al max heap y siempre se agregaban elementos menores que la raíz al min heap (por lo que, teóricamente, se realizaban *log n* intercambios en cada encolamiento). Sin embargo, al realizar los tests, si bien en un principio los tiempos eran mayores, al aumentar el tamaño del input, la función crecía linealmente. Al desconocer cómo representa Java internamente a las colas de prioridad, no pudimos formar un caso que sea el peor.
El tamaño del input se fue aumentando en 100 desde el 0 al 1000 y en 500 del 1000 al 5000.

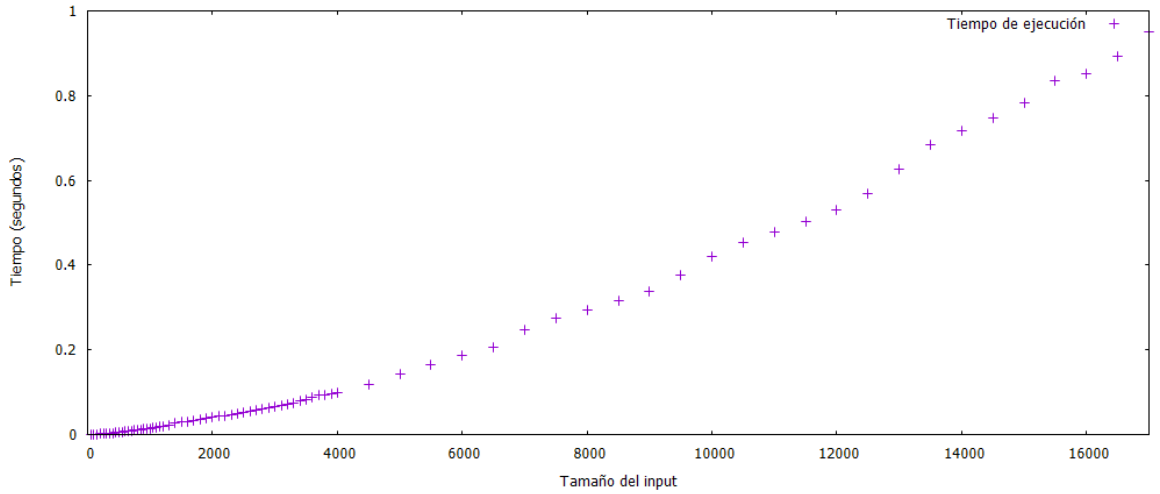

Figure 7: Tiempos variando el tamaño del input

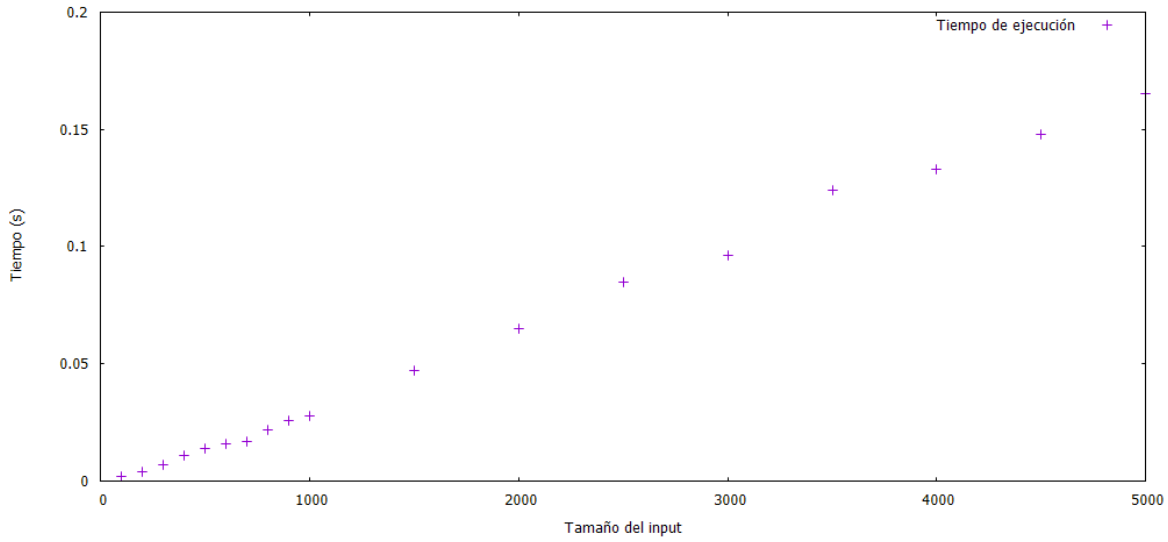

Figure 8: Tiempos del peor caso variando el tamaño del input

3 Problema 3: Girl Scouts

3.1 Descripción del problema:

El problema consiste en obtener a partir de una lista de exploradoras, con forma de ronda y un peso entre cada par de elementos determinado por la distancia entre estos, obtener una permutación de ella tal que se haya minimizado la suma de los pesos de todos los elementos. Se debe devolver dicha lista en el orden que cumpla con la condición y un numero correspondiente a el peso máximo entre dos elementos de la ronda obtenida. En caso de existir mas de una solución, se debe devolver la de menor orden lexicográfico.Notar que si dos elementos no están relacionados, no son amigas, entonces el peso es nulo.

Ejemplos:

1. Se tiene la siguiente lista, vista como una ronda: $l = [a, b, c, d]$ y la siguiente relación de amistad $a = (a, c)(c, d)$ La solución debe ser: $l' = [a, b, d, c]$ ya que $d(a, c)$ y $d(d, c)$ son uno, que es lo mínimo que se puede conseguir. Es importante ver que la distancia ,al ser una ronda, es siempre el $MINposicion(y) - posicion(x), longitudRonda - posicion(y) + posicion(x)$ con x e y dos elementos cualesquiera . Luego, esta solución es la mas chica ordenada de forma lexicográfica. Notar que, entonces, la suma de las distancias totales, que fue minimizada, es : d(a,c)+d(d,c)+0 = 2. Finalmente, el output debe ser: $1[a, b, d, c]$ pues 1 es la distancia máxima entre cualquier par de elementos.
2. Se tiene la siguiente lista, vista como una ronda: $l = [a, b, c]$ y la siguiente relación de amistad $a = (a, c)$ La solución debe ser: $l' = [a, b, c]$ ya que $d(a, c) = 1$. A Finalmente, el output debe ser:$1[a, b, c]$ pues 1 es la distancia máxima entre cualquier par de elementos y la lista es la menor de todas las que cumplen.
3. Se tiene la siguiente lista, vista como una ronda: $l = [a, b, c, d, e]$ y la siguiente relación de amistad $a = (a, b)(a, c)(a, d)(a, e)(c, e)$. La solución debe ser: $l' = 2[a, b, c, e, d]$. Notemos que si bien la lista $[a, b, d, c, e]$ cumple con ser una minimización, la misma es mayor lexicográficamente que l'. Podemos ver fácilmente que como a esta relacionada con todas, no importa el orden en el que se pongan, la suma de las distancias no cambia. Sin embargo, al pedir que (c, e) también sean amigas, obliga a tener que elegir un orden apropiado para estas dos ultimas letras. Básicamente, deben ir "pegadas". Luego, el mejor lugar para ponerlas es aquella que acerque la "e" hacia la "c", para obtener la menor de las posibles soluciones.

3.2 Resolución:

Estructura:

Para resolver el problema, se decidió dividirlo en un conjunto de módulos,clases, para poder abstraer mejor los detalles que no fuesen inherentes al problema principal en si, el de la minimización de las distancias. En este sentido, se implementaron tres clases: Amistad,Ronda y Fogón. Este es el orden de abstracción, es decir la ultima mencionada utiliza los servicios de las de abajo, y así sucesivamente. A saber, la clase Ronda es solo una extensión de la clase ArrayList. La clase Fogón tiene dos miembros: RondaOptima y un Conjunto de amistades(HashSet<Amistad>).

Ideas centrales:

Básicamente, implementaremos un algoritmo que haga backtracking sobre las posibles formas de ordenar una ronda de exploradoras y nos quedaremos con aquella que minimice la suma de las distancias. Esta misma sera la que quedara referenciada por el campo rondaOptima de la clase. Para esto, cuando el algoritmo de backtracking encuentre una primera permutación de la ronda, esta sera nuestra candidata a óptima. Luego,el algoritmo ira explorando nuevas soluciones posibles. Cuando llegue a una

hoja del árbol de posibilidades, es decir, cuando quede definida una ronda que es permutación de la original se preguntara si la nueva suma de distancias es mejor. Si es así, habremos obtenido una nueva rondaOptima. De esta forma continuaremos hasta haber inspeccionado todas las posibles soluciones que sean relevantes(que podrían no ser todas, por las posibles podas).

Mas en detalle, diremos que la clase fogón cuenta con dos funciones: **Solve** y **SentarExploradoras**, que hace el trabajo mencionado arriba. La primera recibirá una *lista(ArrayList)* de exploradoras(Character) y un conjunto de amistades(*HashSet<Amistad>*). A partir de esto:

1. Ordenara la lista alfabéticamente
2. Verificara si hay amistades definidas en el conjunto.
3. Si esto no fuese cierto, la lista de exploradoras es la solución óptima. Generamos una ronda igual a dicha lista y la instanciamos en rondaOptima.
4. Si fuese cierto, y la lista de exploradoras es no vacía, definimos una rondaAuxiliar a la cual le agregamos el primer elemento de la lista de exploradoras. Cambien se remueve dicho elemento de la lista. Finalmente, se llama a la función SentarExploradoras(lista exploradoras, rondaAuxiliar,Tam(exploradoras)) que instanciara el miembro rondaOptima con la solución
5. Devolverá una referencia a el miembro rondaOptima

De esta función podemos remarcar dos aspectos:

La función de **SentarExploradoras** usa, como precondicion, que la lista este ordenada. La razón se vera en detalle luego, pero ,por ahora, diremos que el algoritmo de backtracking mira, en cada nivel de la recursion, las posibles elecciones de letras(exploradoras) en orden. Además, como la solución óptima debe estar ordenada lexicográficamente, en principio, una buena decisión podría ser fijar como primer elemento de la ronda,vista como lista, a la primer exploradora(donde la primera es la menor alfabéticamente, pues ya esta ordenada). Esto evitara, en el backtracking, tener que mirar el resto de las posibles elecciones para la primer posición y comenzar desde la siguiente.

Veamos con detenimiento el algoritmo de SentarExploradoras:

Algorithm 2: SentarExploradoras

```
   Input: Lista exploradoras , Ronda rondaAux, int tam
   // Verificamos si rondaAux esta completa,i.e si ya se formo una rama
 1 if Tamaño(rondaAux) =tam + 1 then
   |  // El caso "base":  Si la solución es mejor que la óptima actual o es la
   |     primera ronda que se arma, pasa a ser la nueva optima:
 2 |  if ( Peso(rondaAux) < Peso(rondaOptima) or Vacia?(rondaOptima)) then
 3 |  |  rondaOptima ← rondaAux
 4 |  |  retornar
 5 |  end
 6 end
   // Si no esta completa, es decir todavía quedan decisiones por tomar, estamos
      en un nuevo paso de la recursion y tenemos que explorar todas las
      elecciones disponibles en exploradoras, para este nivel del árbol.
 7 for i < Tamaño(exploradoras) do
   |  // Elegimos el primero de la lista, que debe ser el elemento mas chico
   |     disponible en el nivel
 8 |  elección ← exploradoras[i]
 9 |  Agregar(elección,rondaAux)
   |  // PODA1:  Se aplica ya sobre el final.  Si vamos a comenzar a mirar las
   |     soluciones del tipo [a,ultima,...()] entonces,podamos esa rama(ver exp
   |     mas adelante)
10 |  if Tamano(rondaAux) = 2 AND i es el ultimo índice then
11 |  |  salir del bucle
12 |  end
   |  // PODA2:  Si la distancia de la ronda actual ya es peor que la óptima,
   |     podar rama
13 |  if Peso(rondaAux) >= Peso(rondaOptima) AND !Vacia?(rondaOptima) then
14 |  |  Remover(rondaAux,elección)
15 |  end
16 |  else
   |  |  // Si no se pudo podar, hay que explorar las posibles elecciones del
   |  |     nivel.
17 |  |  Remover(exploradoras,elección)
   |  |  // Sacamos al elemento elección de la lista de exploradoras, para
   |  |     reducir el espacio de elecciones del próximo nivel y garantizar que
   |  |     no se pueda volver a agarrar mas abajo en la recursion
   |  |  // Hacemos la llamada recursiva para continuar expandiendo la rondaAux y
   |  |     hallar una nueva solución
18 |  |  sentarExploradoras(exploradoras, rondaAux,tam);
   |  |  // Al regresar, habremos explorado el subárbol generado a partir de
   |  |     nuestra ultima decisión
19 |  |  ultima ← RemoverUltima(rondaAux) AgregarAdelante(exploradoras,elección)
   |  |  // /Sacamos la ultima elección de la rondaAux y mantenemos el invariante
   |  |     ,colocando el elemento al comienzo de la misma para mantenerla
   |  |     ordenada y para que quede disponible para las subsiguientes
   |  |     elecciones que se harán en el próximo ciclo(otra forma de verlo es
   |  |     pensar que tiene que estar lista para las exploraciones que van a
   |  |     hacer sus hermanos, i.e las demás posibles decisiones del mismo nivel
20 |  end
21 end
22 retornar
```

Es importante remarcar , ahora, algunos detalles sobre las clases de Ronda y Amistad. Sobre esta ultima, solamente basta aclarar que es una tupla que implementa un par no ordenado. Se hizo,entonces, un "override" del método equals para definir la igualdad como $(a, b) = (b, a)$.

En cuanto a la ronda, la misma es una clase que extiende a $ArrayList < Character >$. Solamente se le agregaron dos métodos relevantes: distancia(int a, int b) y sumaDistancias(HashSet¡Amistad¿ $amigas$)

La primera, simplemente calcula las distancias dados dos índices en la ronda. Para eso, tiene en cuenta que si la diferencia entre b y a es mayor que la mitad, la distancia debe ser: tamaño - b + a. Si no, es simplemente la resta b -a.

En cuanto a sumaDistancias,básicamente calcula las distancias todos contra todos de la siguiente forma:

1. Define un acumulador inicializado en cero
2. Itera la lista(ronda)

   Por cada elemento que recorre, itera el resto de la lista(i.e hay dos bucles anidados).

   Calcula la distancia entre el par de elementos y luego crea una instancia de amistad

   Se fija si dicha amistad esta contenida en el conjunto de amigas, que recibe por parámetro.

   Si es así, acumula la distancia

   Si no, no hace nada

   Continua el curso de los bucles

3.3 Justificación:

Vamos a asumir que las funciones de las demás clases son correctas. Razonaremos , luego, por inducción en la cantidad de exploradoras (e) y dejaremos fijo la cantidad de amistades($a \neq 0$).

Si $n = 1$, entonces la función **solve** agregara el primer, y único, elemento de la lista a la $rondaAuxiliar$. Luego, SentarExploradoras($vacia, rondaAux, 0$) caerá en el primer if(caso base del backtrack) pues $Tamano(vacia) = tam + 1 = 0 + 1$.Entonces, entrara al siguiente if, ya que la $rondaOptima$ se encuentra vacía. De esta forma se instancia rondaOptima con $rondaAux$ y se retornara a la función solve, la cual devolverá rondaOptima. Es fácil ver que, entonces, la solución es correcta en este caso.

Supongamos que $n > 1$. Es claro que no entrara en el primer if, pues $rondaAux$ tiene un solo elemento. Luego, supongamos que estamos en la iesima iteración del ciclo que explora las posibles elecciones. Siguiendo el algoritmo:

Se agrega el iesimo elemento de la lista de exploradoras a la $rondaAux$, que tiene n-1 elementos(pues inicialmente, el backtrack se llama sin el primer elemento de la lista). El invariante del ciclo garantiza que la lista este en orden y que , dentro del mismo nivel del backtrack, el tamaño va a ser $n-1$. Luego, asumamos que no hay ninguna poda para hacer. Entonces, se remueve la iesima exploradora de la lista(ahora con $n-2$ elementos). Luego, por hipótesis inductiva, SentarExploradoras($exploradoras$,$rondaAux$,tam) es correcta y por lo tanto explora todo el árbol de decisión posible con la elección que acabamos de hacer y se tiene en $rondaOptima$ la mejor ronda hasta el momento. Luego, hay que restablecer el invariante. Para eso, quitamos de la $rondaAux$ a la ultima elección, y la volvemos a agregar en la lista adelante de todo, para mantener el orden. Luego, la lista vuelve a tener $n - 1$ elementos. Cuando lleguemos a la iteración final($n - 1$) habremos inspeccionado correctamente todo el árbol de decisión para cada elección de la lista de $n - 1$ elementos. Luego, como miramos todas las posibles combinaciones, habremos encontrado la $rondaOptima$.

¿Que hubiese pasado en caso de haberse efectuado una poda? Si en alguna iteración hubiésemos caído en la primer poda, se habrían perdido posibles soluciones. Sin embargo, se puede ver que aquellas que se descartan ,en este caso, son rondas que se obtienen de mirar "al revés" las que ya se vieron. Por lo tanto no aportan un nuevo espacio de soluciones. Si en alguna iteración hubiésemos caído en la segunda poda, significaría que aquella ronda no era mejor que la óptima. Luego, como agregando nuevas elecciones no podría mejorar, es seguro podar.

3.4 Complejidad:

Como hay mas de una clase, podemos comenzar por analizar la complejidad de las mas pequeñas. En el caso de la clase Ronda, la complejidad de los métodos es la misma que los de ArrayList, pues es una extensión. Para los definididos por nosotros, se puede observar que la complejidad de la función distancia(int a, int b) es $O(1)$, pues solo realiza comparaciones de enteros, operaciones aritméticas básicas y asignaciones. El método int sumaDistancias(HashSet<Amistad> amistad) recorre la lista subyacente por cada elemento en ella. Es decir:

$$\sum_{i=0}^{n} O(i)O(a) = \sum_{i=0}^{n} O(i*a) = O(\sum_{i=0}^{n} i*a) = O(n^2*a)$$

Notar que en cada iteraciones del ciclo interior se llama a la función *contains*() del HashSet de Java. Como la misma esta implementada sobre una tabla de Hash, aunque , en promedio, la operación sea $O(1)$, el peor caso es $O(a)$. La función *contains*() utiliza el método hashcode y equals de la clase amistad para poder determinar si el elemento en cuestión esta en el conjunto. Ambas, se puede ver, son $O(1)$.

Determinemos ahora, la complejidad de SentarExploradoras. Primero, notemos que se utilizan algunos métodos propios de la clase ArrayList(y Ronda). A saber, los mismos son: $size(), add(), remove()$ y $get()$. Según la documentación de Java:

"The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking)." Para el analisis, se tendrá en cuenta el peor caso de $add()$, que es $O(n)$.

Centrándonos en el algoritmo, veamos cuantas "pasos" toma realizarlo. Dada una lista de exploradoras e y un conjunto de amistades a, el algoritmo trabaja con $e-1$ elementos. A partir de esto, podemos pensar el algoritmo de backtracking como un árbol de decisión, donde luego de colocar el primer nodo raíz, elegimos entre $e-1$ posibilidades(hijos),por cada uno de ellos, restan $(e-2)$ elecciones, y así siguiendo. De esta forma, la cantidad de hojas del árbol es $(e-1)!$. Cada hoja representa la cantidad de posibles rondas que se armaron. Sin embargo, esta no es la cantidad de pasos que realiza el algoritmo. Todavía falta agregarle los nodos internos, que fueron pasos previos para obtener las soluciones. Luego, la cantidad de nodos en un árbol completo exactamente e-ario es:

$$N = \sum_{i=0}^{h} e^i = \frac{e^{(h+1)}-1}{e-1} \ .$$

Si t es la cantidad de hojas en el árbol, $e^h = t$, por lo tanto

$$N = \frac{e.t-1}{(e-1)}.$$

Como nuestro árbol esta "acotado" por un árbol exactamente $e-ario$, diremos que:

$$N = \text{cantidad de pasos} = O(\frac{e.t-1}{e-1}) = O(\frac{e.(e-1)!-1}{e-1}) = O(\frac{e!-1}{e-1}) = O(e!)$$

Luego, falta ver cual es el "peso" de estos pasos. En cada iteración del ciclo se hacen:

$add()$ y $remove()$ sobre la lista, que es $O(2e) = O(e)$

$add()$ y $remove()$ sobre la ronda, que es $O(2e) = O(e)$

Se pregunta si hay que hacer la $PODA1$, $O(1)$ pues $size()$ lo es.

Se pregunta si hay que hacer la $PODA2$, llamando a **sumaDistancia** de la *RondaAux* y **sumaDistancia** de la *RondaOptima*, ambas con el conjunto a.

Luego, esto es, por lo visto, $O(2e^2a) = O(e^2a)$

Finalmente la cota de complejidad, teórica, resulta $O(e!e^2a)$. Veamos que esto es$O(e^ea^2)$

Probamos por inducción que $e!e^2a < c(e^e)a \;\; \forall e >= 1, c = 4$ y un a fijo.

$Sie = 1, \quad e!ea = a < 4(e^e) = 4a$

$Sie > 1, \quad (e+1)!(e+1)^2a = e!(e+1)^2(e+1)a = (e!(e^2) + 2e(e!) + e!)(e+1)a$

$< 3e!(e^2)(e+1)(a) < \;\; (por\ H.i)\ 3e^e(e+1)(a) = (3e^{(e+1)} + 3 * e^e)a < 4(e+1)^{(e+1)}a.$

3.5 Tiempo de ejecución y gráficos:

Para ver el desempeño del algoritmo, se pensaron algunos test de peor y mejor caso, al igual que casos sin intencionalidad. En el peor caso, se busco algún escenario en el que las podas se efectúen lo menos posible. Por ejemplo, dado una lista de exploradoras,si el segundo elemento(e1) esta relacionado con el ante ultimo(e2),la primer rondaAux la va a ubicar en su lugar actual. Sin embargo, como e2 tiene que quedar pegada a e1, cada vez que el backtrack la posicione en una posición mas cercana al origen, la poda2 fallará, ya que de ahí en mas ,cada vez que se la mueva, la ronda obtenida sera parcialmente mejor que la actual, por lo que será necesario no podar esa rama. El mejor caso, en cambio, estaría relacionado con una situación en la que una vez que hayamos encontrado la solución(que seria en la primer pasada) y estemos analizando la lista [a,other,b], habremos cortado esa rama. Se verá que el algoritmo respeta la cota calculada teóricamente y que, además, es $\theta(e!e^2a)$

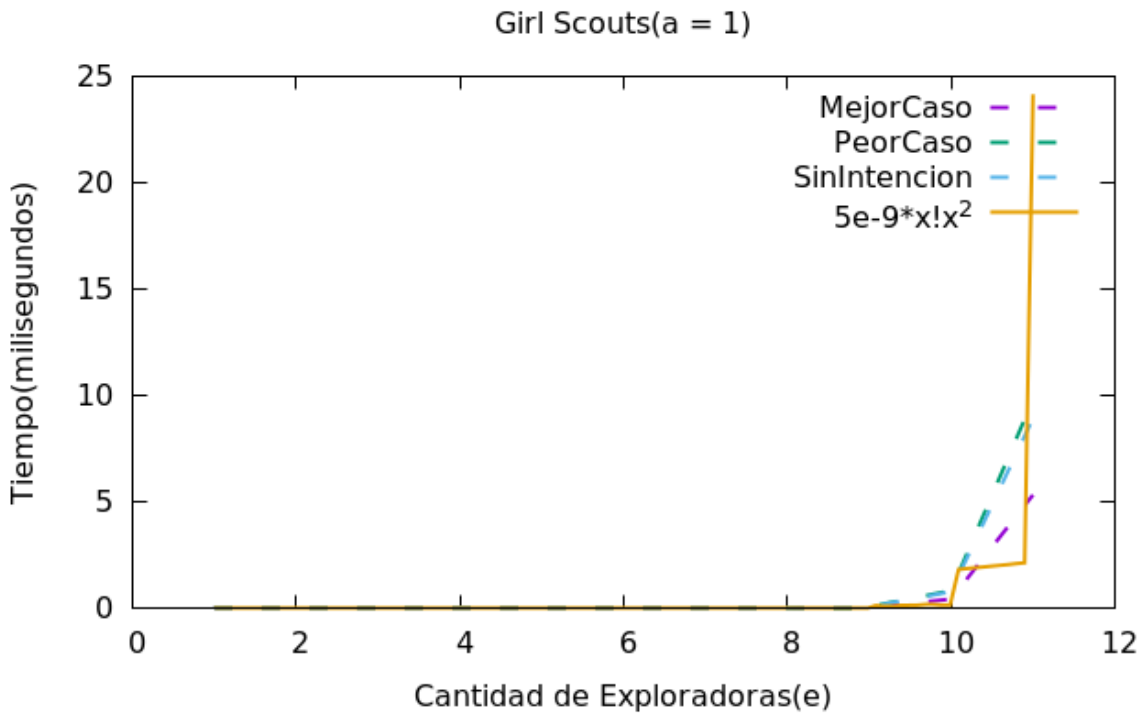

Figure 9: Test de casos

3.6 Partes relevantes del código

La parte de testing se encuentra en la clase FogonTest y Benchmark, donde además también hay breves comentarios describiendo el caso.

```
public void sentarExploradoras(ArrayList<Character> exploradoras, Ronda rondaAux, int tam){

//Veo si es una ronda completa:
  if(rondaAux.size() == tam + 1 ){  //O(1)
//Si la solucion es mejor que la optima actual, pasa a ser la nueva optima:
if((rondaAux.sumaDistancias(amigas) < rondaOptima.sumaDistancias(amigas)) || (rondaOptima.isEmpty(
  rondaOptima = new Ronda(rondaAux);
  return;
}

//Si no esta completa, exploramos las proximas elecciones:
for(int i = 0; i < exploradoras.size(); i++){
//Elegimos el primero de la lista
char e = exploradoras.get(i);
rondaAux.add(e);

//PODA1:Recortamos la rama que mira la ronda "al reves"
if(rondaAux.size() == 2 && i == exploradoras.size()-1 ){
break;
}

//PODA2: Si la distancia de la ronda actual ya es peor que la optima,
if((rondaAux.sumaDistancias(amigas) >= rondaOptima.sumaDistancias(amigas)) && (rondaOptima.size()
rondaAux.remove(rondaAux.size() - 1);

else{

  //Lo sacamos, para que en el proximo nivel de recursion, no este disponible.
 //Esto garantiza que no se pueda volver a agarrar
exploradoras.remove(i);

//Hacemos Backtrack
sentarExploradoras(exploradoras, rondaAux,tam);

//Sacamos de la ronda la ultima eleccion y la recordamo

char ult = rondaAux.remove(rondaAux.size() - 1);

//La volvemos a agregar adelante de la lista
//Esto permite mantener invariable la lista a los ojos del nivel de arriba del backtrack
exploradoras.add(i,e);
 }
}
return;
}
```