DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

31 de agosto de 2015 — Algoritmos y Estructuras de Datos 3

Integrante	LU	Correo electrónico
Zabaleta, Agustín	449/13	mfosco2005@yahoo.com.ar
Jessica, Fernandez de La Vega	449/13	mfosco2005@yahoo.com.ar
chiqui, El	449/13	mfosco2005@yahoo.com.ar
Fosco, Martín Esteban	449/13	mfosco2005@yahoo.com.ar

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

Índice

Resumen

En este trabajo se implementaron algoritmos que proporcionen soluciones a los tres problemas recibidos. A continuación se describen los problemas, dando ejemplos, y luego se describen en detalle los algoritmos que los resuelven.

1. El Telégrafo

1.1. El Problema

1.1.1. Introducción

Se plantea en este caso el problema de, dados:

- Una cantidad de kilómetros de cable para un ramal.
- Una cierta cantidad n de ciudades en un orden determinado, identificadas por la cantidad de kilómetros que las separan de la capital (además de la capital, que se encuentra implícita en todas las sucesiones de ciudades propuestas).

Hallar la mayor cantidad posible de ciudades que se puedan conectar con dicho cable sin cortarlo, en otras palabras: optimizar el uso del cable para conectar la mayor cantidad de ciudades consecutivas. De este resultado óptimo se quiere obtener como resultado cuántas ciudades se pudieron conectar.

Es necesario además que el algoritmo funcione con orden de complejidad $O(n)$.

1.1.2. Ejemplos

A modo de ejemplo de qué plantea el problema, y sus soluciones, se observan casos:

Dados 3 km de cable y las siguientes ciudades: (identificadas por la cantidad de km que las separan de la capital).

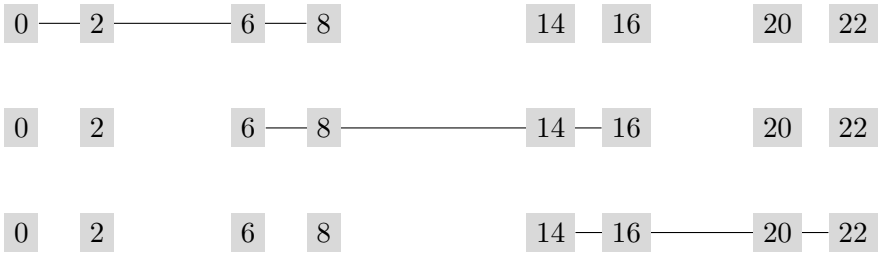
En este caso, lo ideal sería conectar las ciudades 0, 2, 3 entre sí, ya que la cantidad de cable es suficiente y en total conseguiría conectar 3 ciudades, en otros lugares conseguiría conectar menos ciudades:

Nota: **NO** se pueden conectar las ciudades 0-3 y las 6-7 de la siguiente manera:

Si bien se pueden conseguir 5 ciudades de esta manera, los intervalos no son consecutivos, sino que están separados, es necesario que sea distribuido en sólo un intervalo.

Dados 10 km de cable y las siguientes ciudades:

En este caso existen varias respuestas posibles, todas con el mismo resultado, se pueden conseguir 4 ciudades como máximo de las siguientes maneras:



Como el algoritmo debe retornar sólo el máximo de ciudades, cualquiera de los intervalos que tomemos para conectar con nuestros cables servirán a nuestro propósito.

En definitiva, la respuesta será 4 ciudades.

1.2. El Algoritmo

2. A Medias

2.1. El Problema

2.2. El Algoritmo

3. Girls Scouts

3.1. El Problema

3.1.1. Introducción

Se plantea en este caso el problema de, dados:

- Un conjunto de niñas exploradoras.
- Las relaciones de amistad entre ellas, es decir, para cada niña, sus amigas dentro del grupo de exploradoras.

Hallar la manera de sentarlas en una ronda, de manera tal que las niñas que son amigas, esten a una distancia mínima. Formalmente dicho, se pide que la sumatoria de las distancias entre todos los pares de amigas, sea la mínima. Como resultado, se debe devolver la distancia máxima entre los pares de amigas de la ronda, junto con la misma.

El algoritmo debe funcionar con orden de complejidad O($e^e a^2$), donde **e** es la cantidad de exploradoras, y **a** es la cantidad de amistades.

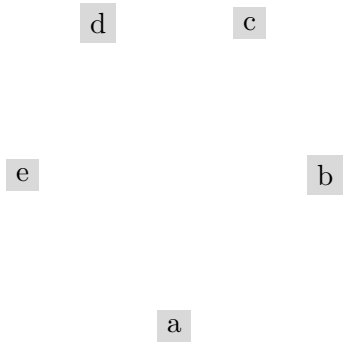
3.1.2. Ejemplos

A modo de ejemplo de qué plantea el problema, y sus soluciones, se observan casos:

Dado el grupo de exploradoras conformado por {a, b, c, d, e}, y sus respectivas amistades

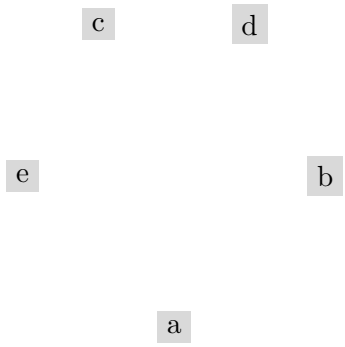
{{a,b}{a,c}{a,d}{a,e}{b,c}{b,d}{b,e}{c,d}{c,e}}

Notar que utilizamos un conjunto para ejemplificar las amistades porque estan son simetricas, lo que quiere decir que si **a** es amiga de **b** entonces **b** es amiga de **a**. Cada una de las amistades tiene su correspondiente distancia en la ronda. Por ejemplo, una posible manera de sentarlas sería:



Donde la sumatoria de las distancias entre todos los pares de amigas en esta ronda es 14

Como el problema nos pedía la ronda cuya distancia entre los pares de amigas sea la mínima, podemos ver intuitivamente como esta ronda no es óptima ya que **d** y **e** no son amigas, sin embargo estan sentadas juntas (a distancia 1). Como en este caso la cantidad mínima de amistades que tiene una exploradora es 3, no puede haber una solución óptima donde sentemos juntas a dos que no son amigas. Podemos ver como al separarlas obtenemos una ronda más óptima (cuya distancia total entre todos los pares de amigas es menor):



La sumatoria de las distancias entre todos los pares de amigas en esta ronda es 13

3.2. Estructuras

En esta sección discutiremos las posibles estructuras de representación que podrían ser utilizadas para la resolución de este problema, y justificaremos nuestra elección.

Para el conjunto de exploradoras, lo importante a remarcar es que queremos modelar una ronda. Sin importar que utilicemos efectivamente para representarla, consideramos de suma importancia (por el uso de abstracción, declaratividad, y modularización del problema) utilizar una clase hecha por nosotros que nos modele el comportamiento de la ronda.

Posibles estructuras que se pueden utilizar para implementar las rondas son:

- Lista simple o doblemente enlazada
- Lista circular
- Arreglo
- Vector

La estructura que utilizamos para resolver el problema de representación de la Ronda fue un Vector que modela el comportamiento de una lista circular a través de las operaciones provistas por la clase Ronda. Por ejemplo [a,b,c,d,e] es una posible representación de la ronda donde, al modelar el hecho de que es circular, la distancia entre **a** y **e** es 1 en vez de 4 porque del último se puede llegar al primero.

Por el otro lado sus respectivas amistades, se pueden representar, por ejemplo, con:

- Matriz de booleanos (1 indica amistad, 0 no).
- Diccionario con claves exploradoras y significado sus amigas.
- Lista de listas.
- Conjunto de conjuntos.

La estructura que utilizamos para resolver el problema de representación de las amistades fue un conjunto de conjuntos, ya que nos interesó modelar el hecho de la simetría entre las amistades, y que la amistad {a,b} es igual a {b,a}.

Además, nuestro contexto de uso (del cual se desprende la complejidad) no nos impuso ninguna restricción que no permita el uso de alguna de las estructuras mencionadas previamente.

3.3. El Algoritmo

3.3.1. Finalización

En esta sección demostraremos que dada cualquier entrada que cumpla con la especificación del problema el algoritmo termina. Que una entrada de datos cumpla con la especificación del problema significa que no haya datos sin sentido. En nuestro caso sería que todas las exploradoras en el conjunto de amistades esten en la ronda, y que todas las exploradoras de la ronda esten en el conjunto de amistades (la doble inclusión).

Para demostrar que el algoritmo termina, hay que analizar cuatro partes del mismo:

- La llamada a la función **permutaciones**.
- La llamada a la función **ordenar**.
- La llamada a la función **maxDistAmistades**.
- El ciclo **for** que itera sobre la ronda.

La función **permutaciones** consiste en, de manera recursiva, calcular todas las permutaciones posibles de una ronda, es decir, todas las posibles maneras de sentar al grupo de exploradoras. La función es llamada con un parámetro **pos** que indica la posición sobre la cual se esta realizando la permutación actual. Esta variable se inicializa en cero, apuntando así al comienzo de la ronda.

El caso base de la función es cuando ya se termino de calcular una permutación posible, que equivaldría a preguntar si el parámetro **pos** llegó al final del arreglo. En ese caso se compara la permutación calculada con la obtenida previamente en los otros pasos recursivos y, en caso de ser menor, se la utiliza como nueva ronda óptima hasta el momento.

El caso recursivo consiste en iterar sobre la ronda de exploradoras desde la posición indicada por **pos** hasta el final de la ronda y, por cada iteración, permutar el elemento indicado por esa iteración con la indicada por **pos** en ese caso recursivo. Como en cada iteración se llama recursivamente a la función con **pos** aumentado en 1, sucede que:

- Cuando pos = 0 el algoritmo realiza e llamadas recursivas con pos = 1.
- Cuando pos = 1 el algoritmo realiza $e - 1$ llamadas recursivas con pos = 2.
- Cuando pos = n el algoritmo realiza $e - n$ llamadas recursivas con pos = n+1.

Por ende, como podemos ver que en cada paso recursivo se reduce el tamaño del problema al aumentar la variables **pos** en 1, podemos afirmar que eventualmente para todas las ramas del arbol de recursión se llega al caso base.

La función ordenar consiste en ordenar la ronda de menor a mayor utilizando el algoritmo **insertion sort**.

La función **maxDistAmistades** consiste en iterar sobre el conjunto de amistades y, para cada amistad, calcular utilizando una busqueda secuencial en la ronda de exploradoras la posición de cada una, y su distancia. Luego, en cada iteración, me quedo con la máxima de las distancias calculadas. Como el algoritmo instancia un iterador que recorre desde el principio hasta el final el conjunto de amistades una única vez, podemos asegurar que finaliza.

El **for** del final del algoritmo finaliza ya que recorre una única vez de manera secuencial la ronda de exploradoras resultado.

3.3.2. Correctitud

En esta sección mostraremos que la solución obtenida por el algoritmo es la pedida por el problema.

El problema pide encontrar la ronda cuya sumatoria de las distancias entre los pares de amigas sea la menor. Nuesta solución al problema fue calcular todas las posibles permutaciones de la

ronda de exploradoras, por lo cual exploramos todas las posibles soluciones a la manera de sentar las exploradoras. Para cada solución nos fijamos si es la que provee en ese momento de ejecución del algoritmo la menor sumatoria de distancias. Como nuestro algoritmo evalua todo el conjunto de posibles soluciones del problema, y se queda en cada caso base con la mejor, podemos asegurar que el algoritmo es una solución correcta al problema, que cumple con las postcondiciones.

Existen dos casos especiales en los cuales no es necesario calcular todas las permutaciones del conjunto de exploradoras.

- Cuando la cantidad de amistades es nula, es decir, cuando ninguna exploradora es amiga de otra.
- Cuando todas las exploradoras son amigas de todas.

Para estos dos casos lo único que hay que hacer es ordenar lexicográficamente la ronda de menor a mayor. Resultan ser los mejores casos del algoritmo ya que cualquier manera de sentar a las exploradoras es una solución posible.

En el primer caso se puede notar que no hay ninguna restricción en cuanto a como sentarlas ya que no hay necesidad de sentar a ninguna al lado de la otra.

En el segundo caso, como todas las exploradoras son amigas entre sí no hay manera de diferir en el resultado. En cualquier permutación posible toda exploradora va a tener al lado suyo a dos de sus amigas y como máximo a distancia $\lfloor \frac{e}{2} \rfloor$ de una amiga si e es par, o de dos amigas si e es impar.

Luego, el resultado provisto es la ronda de exploradoras ordenada lexicográficamente, la cual es la solución al problema en estos casos.

Podemos saber cual es el tamaño máximo del conjunto de amistades ya que se podría pensar al peor caso de las amistades como un grafo completo, donde todos los vertices (exploradoras) estan relacionados con todos los otros. Como sabemos que la cantidad total de aristas de un grafo G(V,X) es:

$$\sum_{i \in V} d(i) = 2m$$

Donde V es la cantidad de vertices, $d(i)$ es el grado del vertice i, siendo el grado de un vertice la cantidad de aristas incidentes a ese vertice, y m la cantidad de aristas total del grafo. Entonces, como m sería la cantidad total de amistades y toda exploradora tiene $e - 1$ amigas tenemos que:

$$m = \frac{\sum_{i=1}^{e} (e-1)}{2} = \frac{e*(e-1)}{2}$$

Con lo cual sabemos que cuando el conjunto de amistades tiene longitud cero o m podemos afirmar que estamos en uno de estos dos casos.

3.3.3. Complejidad

En esta sección mostraremos un pseudocódigo de nuestra implementación, junto con su explicación y justificación de complejidad.

La idea del algoritmo utilizado para resolver el problema fue, dado que tenemos que encontrar la ronda óptima que mínimiza la sumatoria de la distancias, generar todas las posibles permutaciones de la ronda de exploradoras, y luego nos quedamos con la cual era más óptima. Para comparar entre dos rondas cual es mejor a la solución del problema, recorremos el conjunto de amistades y para cada una de ellas sumamos su distancia en un acumulador. Luego, nos quedamos con la que la sumatoria de las distancias sea menor.

Sólo por un tema de notación, al conjunto de conjuntos de char lo renombramos como bigSet, para hacer menos engorroso el código.

RONDAOPTIMA(**in/out** *rondaRes* : `tuple<Ronda int>`, **in/out** *exploradoras* : `Ronda`, **in** *amistades* : `bigSet`) $\longrightarrow$ *res* : `string`

var n : `int`	$O(1)$
$n \leftarrow exploradoras.cantidad()$	$O(1)$
if (amistades.size() $\neq 0 \wedge$ amistades.size() $\neq \frac{n*(n-1)}{2}$) **then**	$O(1)$
$permutaciones(rondaRes, exploradoras, 0, amistades)$	$O(e! * e * a)$
else	
$\Pi_0(rondaRes).ordenar()$	$O(e^2)$
end if	
var $dist$: `int`	$O(1)$
$dist \leftarrow \Pi_0(rondaRes).maxDistAmistades(amistades)$	$O(e * a)$
$res.append(dist)$	$O(1)$
$res.append("")$	$O(1)$
var i : `int`	$O(1)$
for i $= 0$ **to** $\Pi_0(rondaRes).cantidad()$ **do**	$O(e^2)$
$res.push\_back(\Pi_0(rondaRes).exploradoraEnPos(i))$	$O(e)$
end for	
	$O(e! * e * a + e * a + e^2)$

Nota: Π_0 y Π_1 son utilizados para referirse al primer y segundo elemento de las tuplas.

Explicación y justificación de complejidad

La complejidad del algoritmo es $O(e! * e * a + e * a + e^2)$.

- Como $e! = (e * (e-1) * (e-2) * ... * 1) > e$
  $\Rightarrow\ e \in O(e!)$.
  Luego, podemos escribir a $(e! * e * a + e^2) = (e * (e! * a + e)) \in O(e * (e! * a + e))$
  $\Rightarrow\ O(e * max(e! * a, e))$
  lo cual, por como demostramos antes es equivalente a $O(e! * a * e)$
- Como $(e! * e * a + e * a) = (e * a(e! + 1)) \in O((e! * e * a))$

Luego, la complejidad total del algoritmo es $O(e! * e * a))$.

Nuestro algoritmo cumple con la cota de complejidad provista por la catedra ya que:

- Como $e! = (e * (e-1) * (e-2) * ... * 3 * 2 * 1) < e^{e-1}$
  $\Rightarrow\ e! < e^{e-1}$
  $\Rightarrow\ e! * e < e^{e-1} * e = e^e$
  $\Rightarrow\ e! \in O(e^e)$
  Luego, como $e! < e^e$
  $\Rightarrow\ e! * a < e^e * a < e^e * a^2$

Por lo cual, demostramos que la complejidad del algoritmo es $O(e! * e * a) \in O(e^e * a^2)$. Como la complejidad de las funciones **permutaciones** y **maxDistAmistades** las justificaremos con su debido código, en esta parte explicaremos el resto del algoritmo.

Utilizamos las operaciones **append** y **push_back** que cumplen la misma función en cuanto a que concatenan al final del string el string o caracter pasado por parámetro, pero por problema de tipos, utilizamos las dos en los distintos casos. Ambas operaciones son lineales en la cantidad de elementos del nuevo string concatenado. Las primeras dos concatenaciones mediante la función

append al resultado cuestan O(1) ya que, al tener solo dos elementos, esas operaciones son de costo constante.

Luego, el ciclo **for** que itera sobre el conjunto de exploradoras resultante cicla e veces. En cada iteración se llama a la función **push_back** que le concatena al string el nuevo caracer pasado por parámetro. En total el ciclo cuesta:

$$\sum_{i=2}^{e} i = \frac{e*(e-1)}{2} - 1 \in O(e^2)$$

Por último, la complejidad de **exploradoraEnPos** y **cantidad** son constantes ya que estan directamente implementadas con el operator [] y la función *size*() de la clase Vector de c++.

Ahora pasaremos a demostrar las complejidades de los algoritmos auxiliares.

PERMUTACIONES(**in/out** *rondaRes* : `tuple<Ronda int>`, **in/out** *exploradoras* : `Ronda`, **in** *pos* : `int`, **in** *amistades* : `bigSet`)

if (pos == exploradoras.cantidad()) **then**	O(1)
var *sumaDists* : `int`	O(1)
$sumaDists \leftarrow exploradoras.sumaDistancias(amistades)$	O($e * a$)
if (sumaDists $< \Pi_0(rondaRes)$) **then**	O(1)
$\Pi_0(rondaRes) \leftarrow exploradoras$	O(e)
$\Pi_1(rondaRes) \leftarrow sumaDists$	O(1)
end if	
if ((sumaDists == $\Pi_0(rondaRes)$) $\wedge$ ($exploradoras < \Pi_0(rondaRes)$)) **then**	O(e)
$\Pi_0(rondaRes) \leftarrow exploradoras$	O(e)
end if	
else	
var i : `int`	O(1)
for (i = 0 **to** exploradoras.cantidad()) **do**	
$exploradoras.swap(pos, i)$	O(1)
$permutaciones(rondaRes, exploradoras, pos + 1, amistades)$	O(1)
$exploradoras.swap(pos, i)$	O(1)
end for	
end if	O($e * a$)

Explicación y justificación de complejidad

La idea del algoritmo es ir generando todas las permutaciones posibles del conjunto de exploradoras y, en el caso base, quedarse con la secuencia de exploradoras más óptima, cuya sumatoria de distancias entre amigas es menor. En caso de que sean iguales, se considera como menor a la cual es menor lexicográficamente.

Para analizar la complejidad del algoritmo es necesario ver cuantas llamadas recursivas realiza.

- Cuando pos = 0 (su valor inicial), el algoritmo realiza e llamadas recursivas.
- Cuando pos = 1 el algoritmo realiza $e - 1$ llamadas recursivas.
- Cuando pos = n el algoritmo realiza $e - n$ llamadas recursivas.

Lo cual, al generalizar nos queda la recurrencia de la manera:

$$T(e) = \begin{cases} e * T(e-1) & \text{si } e \neq 1 \\ 1 & \text{si } e = 1 \end{cases}$$

Entonces, como hago $e!$ llamadas a una función que tiene orden de complejidada $O(e * a)$ la complejidad total es $O(e! * e * a)$.

$$T(e) = e * T(e-1) = \prod_{i=1}^{e} i = e! \in O(e!)$$

.

SUMADISTANCIAS(**in/out** *exploradoras* : `Ronda`, **in** *amistades* : `bigSet`) $\longrightarrow$ *res* : `int`

var *exp1* : `char`	$O(1)$
var *exp2* : `char`	$O(1)$
var *acum* : `int`	$O(1)$
var *itA* : `amistades.CrearIterador()`	$O(1)$
while itA.siguiente() != NULL **do**	$O(a)$
$exp1 \leftarrow primerElem(itA)$	$O(1)$
$exp2 \leftarrow segundoElem(itA)$	$O(1)$
$acum \leftarrow acum + distancia(exploradoras, exp1, exp2)$	$O(e)$
end while	
$res \leftarrow acum$	$O(1)$
	$O(e * a)$

Explicación y justificación de complejidad

La idea del algoritmo es ir recorriendo el conjunto de amistades y, por cada amistad, almacenar el valor de su distancia en la ronda en un acumulador.

El ciclo itera a veces ya que cicla sobre todo el conjunto de amistades. Las operaciones que obtiene el primer y segundo elemento del conjunto amistad son $O(1)$ ya que el tamaño de ese conjunto esta acotado por 2 (las amistades son de a pares). Luego obtiene la distancia entre las dos exploradoras llamando a la función **distancia** la cual es lineal en la cantidad de exploradoras. Por ende la complejidad total del ciclo, y de la función, es de orden $O(e * a)$.

.

DISTANCIA(**in** *exploradoras* : `Ronda`, **in** *exp1* : `char`, **in** *exp2* : `char`) $\longrightarrow$ *res* : `int`

var *posExp1* : `char`	$O(1)$
var *posExp2* : `char`	$O(1)$
var *dist1* : `int`	$O(1)$
var *dist2* : `int`	$O(1)$
$posExp1 \leftarrow encontrarPos(exp1)$	$O(e)$
$posExp2 \leftarrow encontrarPos(exp2)$	$O(e)$
$dist1 \leftarrow abs(posExp1 - posExp2)$	$O(1)$
$dist2 \leftarrow exploras.size() - dist1$	$O(1)$
$res \leftarrow min(dist1, dist2)$	$O(1)$
	$O(e)$

Explicación y justificación de complejidad

La idea del algoritmo es encontrar las posiciones de las exploradoras en la ronda y luego, como modelamos una lista circular, tener en cuenta que del primero se puede ir al último. Por eso se busca el mínimo entre la distania estandar en cualquier arreglo, y la distancia obtenida cuando se puede ir del primero al último. Como buscar las posiciones dentro de la ronda de las exploradoras es lineal en la cantidad de exploradoras el algoritmo tiene orden de complejidad O(e).

.

encontrarPos(**in** *exploradoras* : `Ronda`, **in** *exp* : `char`) $\longrightarrow$ *res* : `int`

var i : `int`	O(1)
$i \leftarrow 0$	O(1)
while i < exploradoras.size() **do**	O(e)
if exp == exploradoras[i] **then**	O(1)
$res \leftarrow i$	O(1)
end if	
$i \leftarrow i + 1$	O(1)
end while	
	O(e)

Explicación y justificación de complejidad

La idea del algoritmo es realizar una busqueda secuencial en la ronda, la cual es lineal en la cantidad de exploradoras, y por ende, el algoritmo tiene orden de complejidad O(e).

.

maxDistAmistades(**in/out** *exploradoras* : `Ronda`, **in** *amistades* : `bigSet`) $\longrightarrow$ *res* : `int`

var *exp1* : `char`	O(1)
var *exp2* : `char`	O(1)
var *acum* : `int`	O(1)
var *itA* : `amistades.CrearIterador()`	O(1)
while itA.siguiente() != NULL **do**	O(a)
$exp1 \leftarrow primerElem(itA)$	O(1)
$exp2 \leftarrow segundoElem(itA)$	O(1)
$acum \leftarrow max(acum, distancia(exploradoras, exp1, exp2))$	O(e)
end while	
$res \leftarrow acum$	O(1)
	O($e * a$)

Explicación y justificación de complejidad

La idea del algoritmo es ir recorriendo el conjunto de amistades y en cada iteración elegir la amistad que esta a una distancia máxima en la ronda.

El ciclo itera a veces ya que cicla sobre todo el conjunto de amistades. Las operaciones que obtienen el primer y segundo elemento del conjunto amistad son O(1) ya que el tamaño de ese conjunto esta acotado por 2 (las amistades son de a pares). Luego obtiene la distancia entre las dos exploradoras llamando a la función **distancia** la cual es lineal en la cantidad de exploradoras. Por ende la complejidad total del ciclo, y de la función, es O($e * a$).

3.4. Análisis temporal

En esta sección mostraremos la eficiencia del algoritmo, y como se comporta dependiendo del input. Así distinguiremos entre los mejores y peores casos del algoritmo.

El tiempo que tarda en finalizar el algoritmo esta estrechamente relacionado con la cantidad de exploradoras, con la variable e de nuestro problema. Como nuestro algoritmo esta basado en la técnica de backtracking, calcula todas las posibles permutaciones de la ronda de exploradoras, lo cual deriva en **e!** posibles soluciones. Es por esto que la eficiencia del algoritmo no puede diferir en distintos casos teniendo en cuenta únicamente a la variable e.

Por el otro lado, la variable a representa la cantidad de amistades dentro del conjunto de exploradoras la cual, aunque no es completamente independiente de la cantidad de exploradoras, puede hacer variar el tiempo que tarda en finalizar el algoritmo. Consideramos entonces dos casos particulares para el mejor caso del algoritmo:

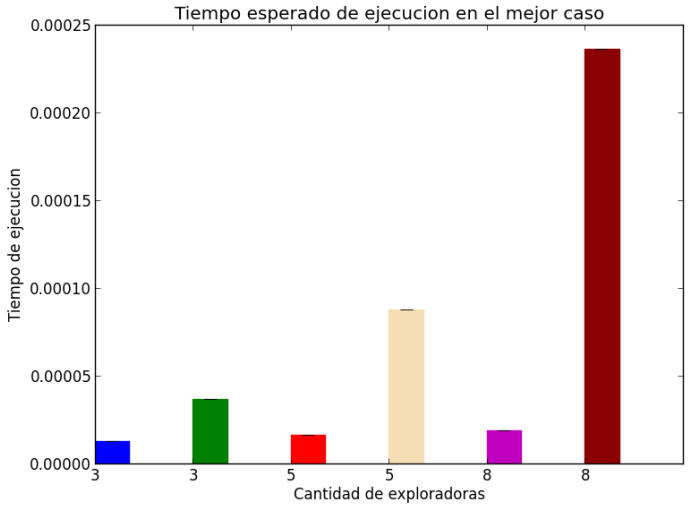
- Cuando la cantidad de amistades es nula, es decir, cuando ninguna exploradora es amiga de otra.
- Cuando todas las exploradoras son amigas de todas.

Estos dos casos resultan ser los mejores casos del algoritmo ya que cualquier manera de sentar a las exploradoras es una solución posible. Por ende, lo único que hay que hacer es ordenar el arreglo de menor a mayor.

Luego, el peor caso del algoritmo se da cuando el conjunto de amistades tiene $\frac{e*(e-1)}{2} - 1$ elementos, lo cual representa el caso en el que todas son amigas entre sí, excepto por un par.

Para mostrar de manera más clara las diferencias de rendimiento entre los distintos casos, corrimos varias veces el algoritmo calculando el tiempo que tardaba en ejecutarse. Luego tomamos el promedio y lo graficamos, de acuerdo a distintos tamaños de inputs. Para todos los casos, utilizamos grupos de exploradoras de tamaño 3, 5, y 8:

Caso mejor:

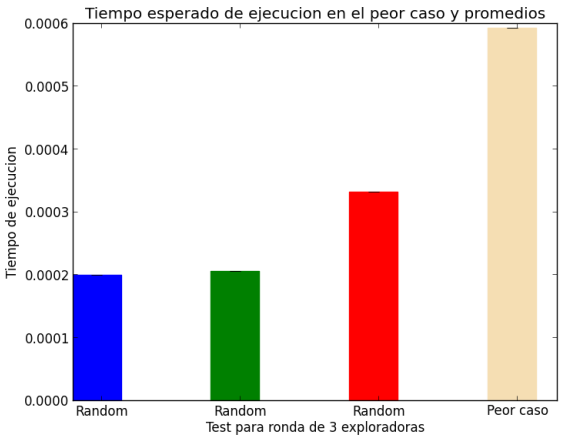


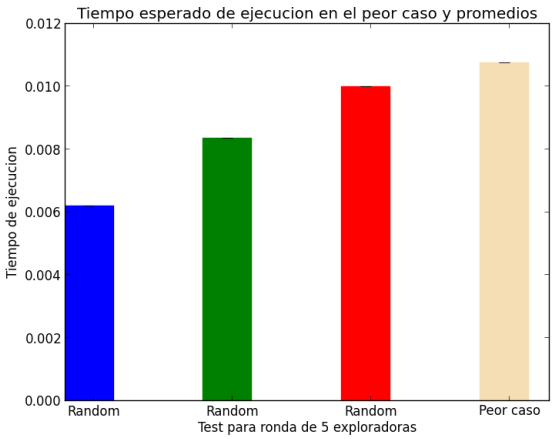
Por cada tamaño calculamos el tiempo para los dos mejores casos, a la izquierda cuando el conjunto de amistades es nulo, y a la derecha cuando el conjunto de amistades es el más grande posible. Podemos notar como los casos donde el conjunto de amistades es el máximo el algoritmo tarda más. Esto se debe a que (aca debería dar una explicación de porque almacenar más datos en memoria hace que vaya más lento).

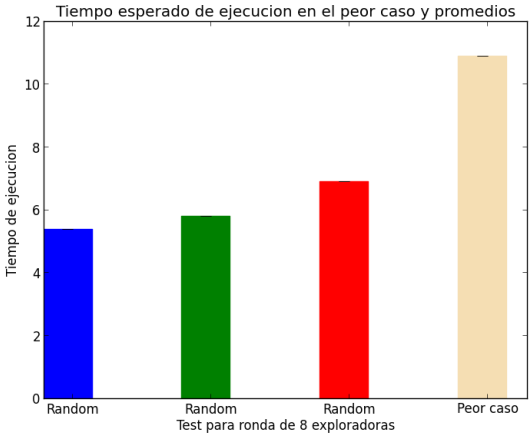
Caso peor y promedio

Para evaluar el peor caso hicimos ejemplos donde todas las exploradoras eran amigas entre sí, excepto por un par. Para el resto de las muestras, realizamos tests sin ninguna intencionalidad. Para generarlos establecimos un conjunto de exploradoras y luego generamos de manera pseudoaleatoria el conjunto de amistades. Tomamos un numero random usando la función **rand_r** y

luego le aplicamos módulo tamaño de la ronda para elegir una exploradora al azar. Repetimos el procedimiento con la segunda elección hasta que sea distinta de la primera. Iteramos $\frac{e*(e-1)}{2}$ ya que es el la cantidad máxima de posibles amistades.







Podemos observar como el algoritmo difiere temporalmente de manera abrupta al elegir un conjunto de 8 exploradoras . Esto sucede por la complejidad propia del problema, la cual es factorial en la cantidad de exploradoras. Por ende, al aumentar levemente la cantidad de exploradoras, el tiempo requerido para finalizar es ampliamente mayor.

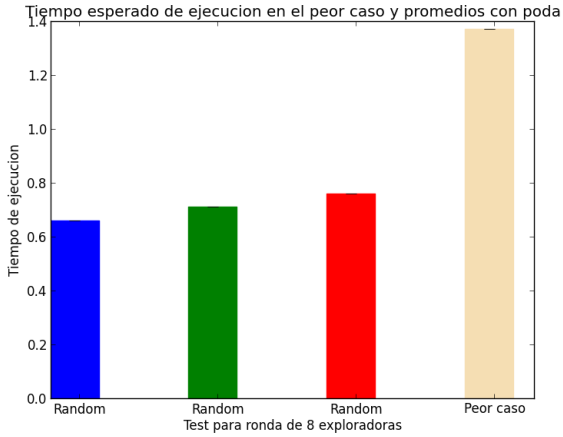
Como aclaramos previamente que una vez fijada la cantidad de exploradoras la complejidad temporal del algoritmo depende exclusivamente del cardinal del conjunto de amistades, mostramos los siguientes datos:

- Para el test con 3 exploradoras el cardinal de amistades es en todos los casos de 1, excepto en el peor caso que es 2.
- Para el test con 5 exploradoras el cardinal de amistades es:
  - Azul : 5 Verde : 4
  - Rojo : 8 Rosa : 9
- Para el test con 8 exploradoras el cardinal de amistades es:
  - Azul : 13 Verde : 14
  - Rojo : 15 Rosa : 28

Por ende, una vez fijada la cantidad de exploradoras, podemos observar como el tiempo de ejecución del algoritmo difiere en base a la cantidad de amistades.

Como nuestro algoritmo utiliza la técnica de backtracking, la cual consiste en explorar todas las posibles soluciones, consideramos utilizar una poda para mejorar el rendimiento del algoritmo. Como el algoritmo pide encontrar la ronda que, ademas de que cumpla la condición de las distancias, también sea la menor lexicográficamente se puede utilizar esta información para acotar el espacio de posibles soluciones. En el ejercicio modelamos una ronda y lo que nos interesa como resultado es la distancia relativa entre cada amiga, no su posición en la ronda. Con solo explorar las posibles permutaciones que comienzan con la exploradora que es la menor lexicográficamente de todo el conjunto es suficiente. Esto sucede porque dada cualquier ronda que no empiece con la menor exploradora, se la puede reacomodar a través de rotaciones simples entre las exploradoras contiguas, hasta llegar a una ronda cuya distancia entre todas es la misma (ya que no modificamos sus distancias relativas) y es menor lexicográficamente.

Al utilizar esta poda podemos observar como el algoritmo mejora su eficiencia en aproximadamente un 85 porciento:



Falta un poco de explicación aca, o conclusiones