DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Técnicas algorítmicas (Grupo 11)

30 de agosto de 2014

Algoritmos y Estructuras de Datos III
Trabajo Práctico Nro. 1

Integrante	LU	Correo electrónico
Iván Matías Badgen	259/10	ivanbadgen@gmail.com
Pablo Diego Rago	239/10	p4blo.r@gmail.com
Alejandro Grinberg	232/10	alegrinberg90@gmail.com
Uriel Libster	364/10	urilib@gmail.com

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

Índice

1. Ejercicio 1

1.1. Introducción

Hola Mago.

El problema a resolver consiste en encontrar la longitud de la meseta más extensa luego de combinar dos vectores de entrada, ordenados. Una meseta es una sucesión de elementos de igual valor. Para resolver esto, utilizamos una variante del algoritmo Merge**[1]** como base, agregándole la funcionalidad de contar repeticiones y poder comparar esta cantidad con la máxima longitud encontrada hasta el momento. La complejidad pedida es $O(|X| + |Y|)$ donde X e Y son los vectores de entrada.

1.2. Desarrollo

La solución planteada toma el algoritmo Merge**[1]** y lo modifica, para ir recorriendo en forma ordenada los valores de los vectores. En lugar de realizar la mezcla, se compara el valor actual con el último leído y se cuenta la cantidad de repeticiones. En caso de que los valores coincidan, el contador aumenta; en caso contrario, el algoritmo compara el conteo actual con el máximo alcanzado hasta el momento y se queda con el máximo entre ambos, reiniciando el contador y modificando la variable que contiene el último valor leído. De esta forma se logra obtener la longitud de la meseta más extensa con la complejidad temporal requerida.

```
Sean X, Y los vectores de entrada
Sean iX, iY los indices correspondientes a los vectores

Para i de 0 a (|X| + |Y| - 1)
  Si tengo elementos en X y (X[iX] <= Y[iY] o no tengo elementos en Y)
    Si X[iX] == ultimo_valor_leido
      aumento contador de repeticiones
    Sino
      me quedo con el maximo entre el contador y el maximo hasta el momento
      reinicio el contador
      modifico el ultimo_valor_leido
    aumento iX
  Sino
    realizo el proceso analogo para Y
```

Como se puede apreciar en el pseudocódigo, estamos iterando $|X| + |Y|$ veces un bloque de instrucciones que toman costo constante. Estas son asignaciones o comparaciones de variables de tipo entero o elementos de un array, cuyo acceso indexado es constante. Para el caso de encontrar el máximo entre dos elementos de tipo entero, puede realizarse también con costo constante. Concluímos que la complejidad de nuestro algoritmo es $\Theta(|X| + |Y|)$.

El algoritmo simula el merge y recorre en forma ordenada ambos vectores. Mientras se conserve el mismo número, el contador aumenta. Cuando se encuentra un número diferente al anterior, se evalúa si la cantidad de repeticiones es mayor al máximo acumulado, se toma una decisión según esto y se modifican las variables necesarias.

La correctitud del algoritmo está relacionada con la precondición de que los vectores están ordenados. Es decir, dado V alguno de los dos vectores, si la mayor meseta pertenece exclusivamente a uno de ellos, sabemos que $V_i \leq V_{i+1}\ \forall\, 0 \leq i < |V|$ por estar ordenado. En particular, mientras se cumple que $V_i = V_{i+1}$ el algoritmo cuenta esas repeticiones y va actualizando el valor máximo hasta el momento, en caso de ser superado. Si la meseta es compuesta por ambos, viendo el algoritmo se ve que si o si estamos en el caso en que $X[i_x] \leq Y[i_y]$ pues, de otra manera, significaría que alguno de los vectores ya está vacío o que $X[i_x] > Y[i_y]$. Utilizando el mismo argumento que antes, en el cual $V_i = V_{i+1}$, ahora se cuentan las repeticiones mientras que $X[i_x] \leq Y[i_y] \vee X[i_x] = X[i_x + 1]$. Teniendo en cuenta que los índices se aumentan apropiadamente, vemos que la cuenta de las repeticiones va a ser la esperada. Como al final del algoritmo nos fuimos quedando con la máxima, el resultado es el pedido.

1.3. Resultados y Discusión

En el proceso de experimentación se utilizaron 3 archivos de test: el propuesto por la cátedra, otro generado por el grupo con ciertos casos especiales y el tercero generado aleatoriamente (el tamaño y los valores de ambos vectores fueron generados aleatoriamente entre un rango definido). La experimentación sirvió para corroborar que el algoritmo propuesto funcionara correctamente y que la cota de complejidad encontrada y justificada en la sección anterior, en la práctica se cumpliera.

Para corroborar la correctitud, se crearon tests, con casos borde y disjuntos entre sí. Tanto en el test de la cátedra como en el test del grupo se muestra el caso de ambos vectores iguales y con una única repetición de cada número. En estos casos, la longitud de la meseta más extensa es 2 ya que, al mezclar los vectores, el orden se mantendría y todos los elementos quedarían repetidos dos veces. En nuestro test agregamos el caso de ambos vectores iguales, pero con repetición de valores. En este caso, la longitud de la meseta más extensa al combinar ambos es el doble de la longitud de la meseta más extensa de cualquier vector (recordando que son iguales). Al mezclar los vectores, los valores se duplican y por lo tanto todas las mesetas se mantienen pero duplican su extensión. Este caso es una generalización del anterior.

Otros casos testeados fueron vectores que no comparten ningún valor, en los que la longitud de la meseta más extensa es 1, vectores en donde la meseta más grande se encuentran al principio/final de la combinación de los vectores, vectores en donde la meseta esta formada por elementos de un único vector, vectores en donde la meseta esta formada por elementos de ambos vectores.

Un caso interesante es cuando la meseta más grande al combinar los vectores no coincide con la meseta más grande de ninguno de los dos . Esto sucede justamente porque al combinarlos, un elemento que aparecía en ambos vectores, pero no alcanzaba a formar la meseta mas grande en ninguno, consigue una cantidad de repeticiones igual a la suma de las repeticiones en ambos vectores. Esta supera a las mesetas de cada vector porque en el otro ese elemento no cuenta con tantas repeticiones.

La correcta ejecución de todas estas instancias de test, en donde los resultados obtenidos son iguales a los esperados, nos ayudaron a verificar que el algoritmo era correcto más alla de las explicaciones teóricas.

Para corroborar la cota de complejidad, en la implementación del algoritmo se agregó un conteo de operaciones. Para no ensuciar el código, y como todas las instrucciones realizadas toman tiempo constante (como fue explicado en la sección anterior), el conteo no se hizo por cada instrucción, sino que para este problema, se cuenta la cantidad de iteraciones del ciclo, lo que determina realmente la complejidad. Por cada instancia de test, se lista el tamaño de la entrada (en este caso, la suma del tamaño de los vectores) y la cantidad de operaciones. Como se puede ver en el pseudocódigo, la cantidad de iteraciones es justamente el tamaño de la entrada, por lo que se puede afirmar y observar en los gráficos presentados a continuación que la complejidad es lineal respecto al tamaño de la entrada. Se decidió utilizar este conteo de operaciones y no utilizar alguna otra herramienta (como medir el tiempo o la cantidad de ciclos de clocks transcurridos) para que los resultados fueran totalmente independientes del contexto en el cual se ejecutaran los test.

En los siguientes gráficos, cada cruz roja es un punto en el gráfico que representa el tamaño de la entrada y el costo en operaciones. Se grafican todas las instancias de cada test aleatorio. En color verde se grafica la función lineal $y(x) = x$. Se puede ver que todas las cruces rojas se encuentran sobre la función lineal, lo que empíricamente muestra lo explicado anteriormente.

Figura 1: 50 instancias con vectores de hasta 50 elementos

Figura 2: 50 instancias con vectores de hasta 20 elementos

Figura 3: 10 instancias con vectores de hasta 20 elementos

Los parámetros del generador de los test aleatorios son la cantidad de instancias, el tamaño máximo del vector X, el tamaño máximo del vector Y y el valor máximo que puede tomar un elemento en ambos vectores. Del gráfico 1 al 2 se modificó el tamaño máximo de los vectores. Como este valor determina la cantidad de operaciones, se puede ver que la diferencia entre los gráficos es el rango de valores que toman ambos ejes. Del gráfico 2 al 3 se modificó la cantidad de instancias, lo que determina que en el gráfico se vean menos cruces rojas. Más alla de estas variantes, se puede observar que la relación entre el tamaño de la entrada y la cantidad de operaciones se mantiene siempre lineal.

La modificación del último parámetro, el valor máximo que puede tomar cualquier elemento, no influye en estos aspectos de

complejidad, sino que cambia la longitud de las mesetas. Mientras más grande sea este número, cada elemento de cada vector tiene un rango mayor entre el cual el generador aleatorio puede elegir. Esto implica que probablemente haya menos repeticiones de números y por lo tanto que la extensión de las mesetas sea menor.

1.4. Conclusiones

Como se puede observar mediante el pseudocódigo y su posterior explicación, y los gráficos anteriormente expuestos, la relación entre el tamaño de la entrada y la cantidad de operaciones es lineal, tal cual requería el problema. Analizando el pseudocódigo también se puede ver que no existe un mejor o peor caso para nuestro algoritmo, y por lo tanto la cota de complejidad para cualquier caso es $\Theta(|X| + |Y|)$.

En cuanto a la correctitud, a partir de las explicaciones teóricas tomando como base el pseudocódigo y los experimentos realizados tanto de las funciones críticas específicamente, como de todo el algoritmo (mediante tests pensados y tests aleatorios), se prueba que el algoritmo propuesto resuelve el problema correctamente.

2. Ejercicio 2

2.1. Introducción

El objetivo de este ejercicio es encontrar la meseta de máximo tamaño en una matriz de tamaño $n \times m$. Una meseta es un conjunto de celdas adyacentes sobre sus lados que comparten un mismo valor.

La idea del algoritmo propuesto es ir recorriendo la matriz y calculando la meseta que se inicia en cada celda. Esto se realiza en forma recursiva en caso de encontrar una celda adyacente de igual valor, siempre y cuando todavía no se haya pasado por dicha celda. Esto es necesario para no generar un bucle infinito de llamadas. La complejidad requerida para el algoritmo es $O(n \times m)^2$ y la complejidad obtenida fue $O(n \times m)$.

2.2. Desarrollo

Como convención, de aquí en adelante llamaremos a las casillas adyacentes según los puntos cardinales correspondientes a la dirección en la cual se encuentran.

Para almacenar los datos, se utilizan estructuras de tipo matriz que permiten acceso a cada elemento con costo $\Theta(1)$.

```
Para cada casilla del tablero
  Si no está marcada en la matriz auxiliar
    Resetear el contador que lleva la cuenta del tamaño de la meseta actual
    Calcular meseta que comienza aquí
    mesetaMaxima = máx(máxima calculada hasta el momento; recientemente calculada)
```

Para ver si ya se pasó por una casilla se mantiene una estructura auxiliar que es una matriz de $n \times m$ en la cual se va marcando cada casilla al momento de comenzar a contar. Para calcular la meseta que comienza en una celda proponemos el siguiente algoritmo recursivo (en pseudocódigo):

```
Marcar la casilla en la matriz auxiliar
Incrementar el contador que lleva la cuenta del tamaño de la meseta
Si existe una casilla al oeste y no está marcada en la matriz auxiliar
  y tiene el mismo valor que la casilla actual
    llamar recursivamente con la casilla que está al oeste
Análogamente para el sur, este y norte
```

Este algoritmo impide que se genere un bucle infinito al preguntar si ya se había marcado esa casilla. Por eso es importante mantener la estructura auxiliar actualizada, lo que se realiza al comienzo del algoritmo.

Se puede ver que la cantidad de veces que *pasemos* por una casilla, es decir las veces que chequeamos si esta marcada en la matriz auxiliar, es a lo sumo 5: una vez en el ciclo principal y otras 4 en el algoritmo recursivo, cuando se llama desde una casilla adyacente.

La iteración se realiza tantas veces como casillas haya, que son $n \times m$. Revisar si ya se pasó por una casilla toma tiempo constante, porque se utiliza una estructura que permite acceso directo. Las comparaciones, asignaciones y cálculo de máximos tienen costo constante. La llamada al algoritmo recursivo se realiza como máximo una vez por cada casilla, con esto nos aseguramos que el algoritmo termina. Si se propaga la recursión a otra, esta se marca en la estructura auxiliar y no se volverá a calcular cuando el ciclo principal llegue allí o a otra casilla adyacente. Las comparaciones que se realizan en el algoritmo recursivo también toman tiempo constante pues son de tipos básicos -en particular, enteros- y el acceso a cada uno es constante por encontrarse en estructuras de tipo matriz.

Como mencionamos anteriormente, el algoritmo va recorriendo la matriz que representa el tablero fila por fila, de izquierda a derecha, de arriba a abajo. Al pasar por cada casilla calcularemos la meseta a la cual ésta corresponde. Esto se hace un número finito de veces ya que el tamaño tablero es finito. La meseta más larga que comienza en una casilla tendría tamaño 1, si sus adyacentes tienen distinto valor, y sino, 1 más el tamaño de las mesetas que comiencen en sus adyacentes de igual valor. Cabe aclarar que para el cálculo de las mesetas que comiencen en sus adyacentes de igual valor, dicha casilla no será tenida en cuenta.

Esto lo logramos manteniendo una matriz auxiliar donde marcamos si una casilla ya se utilizó para el cálculo de la meseta; una vez que esto ha sucedido la consideramos *marcada*. Notemos que de esta manera **ninguna** casilla será utilizada mas de una vez para el cálculo de la meseta. A partir de esto podemos ver que, sin importar desde qué casilla comencemos a calcular una meseta, siempre lograremos abarcarla toda y por ello el cálculo es correcto. Adicionalmente, en el bucle principal, la meseta a la que corresponde cada casilla se calcula sólo si está no esta marcada. De esta forma nos aseguramos de que estamos calculando cada meseta una sola vez. Al mismo tiempo, como al recorrer todas las casillas nos aseguramos de que todas las mesetas serán calculadas, podemos ver que si tomamos el máximo tamaño de entre todas las mesetas el resultado final será correcto.

2.3. Resultados y Discusión

En primer lugar, para testear correctitud empíricamente armamos casos de prueba concretos con instancias de entrada que consideramos relevantes. Estas fueron diseñadas especialmente para comprobar que el algoritmo se comporte correctamente con datos de entrada *borde*. Por ejemplo, armamos tableros donde cada casilla contiene un elemento distinto de manera tal que la meseta más grande tenga tamaño uno. Otro caso particular fue el de un tablero en donde todas sus casillas tienen el mismo elemento, con lo cual hay una sola meseta que tiene el tamaño del tablero mismo. Casos donde la meseta más grande contiene a la primera casilla del tablero, donde contiene a la última, donde hay dos mesetas distintas y ambas tienen el tamaño máximo. Pudimos comprobar que nuestro algoritmo se comporta correctamente bajos todas estas situaciones.

Para realizar las mediciones de tiempo, generamos instancias de entrada aleatoriamente. Para ello utilizamos la clase *RandomGenerator*. Dados los parámetros *cant, n, m, elem* se genera un archivo con *cant* instancias de prueba, cada una con un tablero de tamaño aleatorio de entre una y *n* filas y entre una y *m* columnas, donde en cada casilla hay un número elegido al azar entre uno y *elem*.

Consideramos como tamaño de la entrada al tamaño del tablero, *n* x *m*. Por ejemplo, si tomamos un tablero de dimensiones 3×4, el tamaño de la entrada será 12.

Para contar la cantidad de operaciones utilizamos contadores dentro de la implementación de los algoritmos. Estos son incrementados cada vez que se realiza un conjunto de operaciones de costo constante, dentro de un ciclo con una cantidad de iteraciones dependiente del tamaño de la entrada ó dentro de una llamada recursiva que se hace una cantidad de veces que depende del tamaño de la entrada.

Primero medimos 30 instancias para tableros de un máximo de 50 filas y 50 columnas y elementos con valor máximo de 6. Con estas instancias esperamos obtener mesetas grandes ya que el tamaño esperado de los tableros es relativamente grande, debido a que cada casilla puede tomar uno de 6 valores posibles. A continuación, en la figura 4 podemos ver los resultados de las mediciones. Se puede observar que la cantidad de operaciones es lineal respecto al tamaño de la entrada.

Figura 4: Gráfico de cantidad de operaciones en función del tamaño de la entrada para instancias generadas aleatoriamente con tableros de tamaño máximo de 50x50 y elementos al azar del 1 al 6

Luego medimos 30 instancias para tableros de un máximo de 10 filas y 10 columnas y elementos con valor máximo de 3. Para estas instancias también esperamos obtener mesetas grandes ya que cada casilla puede tomar uno de sólo 3 valores posibles. Sin embargo, pueden considerarse medianas en comparación a las primeras puesto que su tamaño máximo posible es mas reducido, 10x10. Los resultados de estas mediciones se observan en la figura 5. Nuevamente podemos ver que la cantidad de operaciones es lineal respecto al tamaño de la entrada.

Figura 5: Gráfico de cantidad de operaciones en función del tamaño de la entrada para instancias generadas aleatoriamente con tableros de tamaño máximo de 10x10 y elementos al azar del 1 al 3

Finalmente, medimos 30 instancias para tableros de un máximo de 5 filas y 5 columnas y elementos con valor máximo de 25. A partir de estas esperamos obtener mesetas muy chicas ya que cada casilla puede tomar uno de 25 valores posibles, lo cual es mucho en comparación a las dimensiones que puede llegar a tomar un tablero. Además del tamaño esperado de las mesetas, estas instancias contrastan con las anteriores en el hecho de que las dimensiones de sus tableros van a tender a ser bastante menores. Podemos ver los resultados de las mediciones en la figura 6. Al igual que en los casos anteriores se observa que el incremento de la cantidad de operaciones respecto al tamaño de la entrada es lineal.

Figura 6: Gráfico de cantidad de operaciones en función del tamaño de la entrada para instancias generadas aleatoriamente con tableros de tamaño máximo de 5x5 y elementos al azar del 1 al 25

En todos los casos, la constante que acompaña la linealidad es un 2. Esto es algo que depende de cómo se cuenten las operaciones en el algoritmo. En particular, decidimos incluir las constantes 1, 2, y 3 en los gráficos para poder comparar.

2.4. Conclusiones

Como se observa en las figuras 4, 5 y 6, la relación entre el tiempo, medido en cantidad de operaciones, y el tamaño de la entrada, es siempre lineal lo cual concuerda con el análisis teórico realizado, en particular con un 2 como constante. Podemos ver que esta linealidad se mantiene para muy diversos tamaños de la entrada, desde tableros de unas pocas casillas hasta un par de miles, y también para muy diversos tamaños de la meseta máxima obtenida. A partir de los resultados también vemos que no encontramos un mejor ni peor caso para nuestro algoritmo, si no que la cantidad de operaciones que demora no depende de los datos de entrada, ni en cuanto al tamaño del tablero ni en cuanto a la distribución de los valores de las casillas.

3. Ejercicio 3

3.1. Introducción

Se dispone de un tablero de $n \times m$ en el que se colocan t fichas, en t casillas distintas. El objetivo del juego es ir eliminando las fichas mediante saltos hasta dejar sólo una. Una ficha se saca del tablero cuando otra adyacente salta por sobre ella en sentido vertical u horizontal quedando en el casillero inmediato siguiente, que debe estar desocupado antes del salto. El algoritmo propuesto encuentra **una** sucesión de pasos a seguir para ganar el juego. Es importante notar que el juego no siempre va a tener solución y que en caso de tenerla, podría ser más de una. El ejercicio no tiene una restricción de complejidad y la calculada para nuestro algoritmo aplicando la técnica de backtracking fue de $O((4t)^{t+1})$.

3.2. Desarrollo

La idea del algoritmo propuesto será ir listando todos los posibles movimientos y para cada uno de ellos, crear una nueva instancia de los datos, ejecutar el movimiento e intentar resolver el nuevo juego, con una ficha menos. Puede verse que, mientras haya movimientos posibles, al tener una ficha menos cada instancia, el algoritmo va a terminar. Por otra parte, la naturaleza del algoritmo es recursiva, basándose en que un juego sin movimientos posibles con una o más fichas está terminado. En caso de llegar a una sola ficha, sabremos que se obtuvo una solución al problema y en este caso, el algoritmo termina (no continúa recorriendo ninguna otra posible solución).

La idea de la técnica de backtracking es ir recorriendo las posibles soluciones en forma de árbol, recursivamente, a diferencia de la técnica de fuerza bruta que consistiría en enumerar todos los casos e intentar resolverlos. En nuestro caso, cada rama del árbol se corresponde con cada movimiento posible que se va ejecutando y generando nuevos posibles caminos.

El pseudocódigo del algoritmo propuesto es

```
 1.Para cada ficha
 2. Buscar movimientos posibles (norte, sur, este u oeste)
 3. Para cada movimiento posible
 4.   Copiar las estructuras de datos del juego
 5.   Realizar el movimiento
 6.   Si la cantidad de fichas es mayor a 1
 7.     Resolver el nuevo juego recursivamente
 8.   Sino
 9.     Retornar la solución obtenida
10.Retornar solución vacía
```

Para una implementación particular en lenguaje Java, utilizamos una clase llamada Juego que contiene la lógica de su resolución. Sus estructuras internas son dos listas enlazadas. En una se almacenan las fichas (un par x e y que representan su posición en un tablero). En la otra, movimientos que están dados por una posición origen y una destino. Estos son los movimientos que condujeron desde el juego inicial al juego actual.
Las listas enlazadas aseguran costo de inserción $\Theta(1)$ y tanto para búsqueda como para eliminación $O(n)$, siendo n la cantidad de elementos en dicha lista**[2]**.

Copiar las estructuras del juego tiene un costo de peor caso $O(t)$ ya que se copian dos listas enlazadas, una con todas las fichas restantes (cada una es un par) y la otra con los movimientos realizados hasta el momento (un par de fichas). Es decir, la cantidad de operaciones básicas puede acotarse por $2t$ (t cantidad de fichas inicial), lo cual deja una complejidad de $O(t)$.

Buscar movimientos posibles tiene costo $\Theta(t)$ ya que, dadas las estructuras utilizadas, es necesario recorrer toda la lista de fichas para ir evaluando la realización de ese movimiento.

Para realizar un movimiento, se utiliza el siguiente algoritmo

```
Calcular la posición destino de la ficha que se mueve
Agregar <posición actual, destino> a la lista de movimientos
Actualizar la posición actual de la ficha con destino
Remover la ficha a eliminar de la lista de fichas
```

Este algoritmo tiene costo $O(t)$ siendo t la cantidad de fichas, ya que remover la ficha de la lista tiene este costo por lo dicho anteriormente. El resto de las operaciones toman tiempo constante.

La complejidad total del algoritmo va a estar condicionada mayormente por la cantidad de llamadas recursivas que se realizan; en particular por el costo de cada una. En un principio tenemos t fichas. La cantidad de movimientos que se pueden ejecutar está acotada por $4t$ ya que cada ficha puede realizar un máximo de 4 movimientos. La cota es bastante *grosera* ya que en la práctica esto difícilmente se cumpla para todas. Cada movimiento implica una llamada recursiva para la cual el nuevo juego tendrá una ficha menos. Si llamamos f al costo de ejecutar el algoritmo, en cada paso tenemos los siguientes costos:

- 1. Se ejecuta t veces
  - 2. $O(t)$
  - 3. Se ejecuta a lo sumo 4 veces
    - 4. $O(t)$
    - 5. $O(t)$
    - 6. $\Theta(1)$
    - 7. $f(t-1)$

Por lo tanto, obtenemos la siguiente ecuación

$$f(t) = \begin{cases} 1 & si\ x = 1 \\ t.(t + 4(t + t + 1 + f(t-1))) & si\ x \neq 1 \end{cases}$$

En caso de tener una sola ficha, puede verse en el pseudocódigo que el costo total es constante ya que hay una sola ficha y no tiene movimientos posibles.

Desarrollando la ecuación obtenemos para la ecuación recursiva:

$$f(t) = 9t^2 + 4t * f(t-1)$$
$$f(t) = 9t^2 + 9 * 4t(t-1)^2 + 4^2t * t * (t-1) * f(t-2)$$
$$f(t) = 9t^2 + 9 * 4t(t-1)^2 + 9 * 4^2t * (t-1) * (t-2)^2 + 4^3t * t * (t-1) * (t-2) * f(t-3)$$

Si seguimos expandiendo la ecuación obtenemos:

$$f(t) \leq \sum_{i=0}^{t-2} \left(9 * 4^i * \prod_{j=0}^{i-1} (t-j) * (t-i)^2 \right) + 4^t * t! * f(1)$$

y acotando $(\prod_{j=0}^{i-1} t - j) * (t-i)^2 \leq t^{i+2}$ obtenemos la siguiente expresión

$$f(t) \leq \sum_{i=0}^{t-2} \left(9 * 4^i * t^{i+2}\right) + 4^t * t! * f(1) \leq (t-1) * 9 * 4^{t-2} * t^t + 4^t * t! * f(1) \leq$$

$$(t-1) * 9 * 4^t * t^t + 4^t * t^t \leq (t-1) * 9 * 4^t * t^t + 9 * 4^t * t^t \leq t * 9 * 4^t * t^t \leq 9 * (4t)^{t+1}$$

Como $f(t)$ es una función de complejidad, el costo del algoritmo en el peor caso es $O(f(t)) = O((4t)^{t+1})$
Las acotaciones realizadas son en muchos casos bastante *groseras*, por lo que en la práctica podemos esperar que el algoritmo se comporte de forma más eficiente.

En un principio consideramos utilizar una estructura auxiliar adicional, de tipo matriz, de las dimensiones del tablero, en donde guardar las posiciones de las fichas. La intención era conseguir un acceso rápido a dichas posiciones a la hora de calcular los saltos posibles. Sin embargo, a la hora de copiar las estructuras esto tenia un costo excesivo y además nuestra complejidad pasaba a depender de las dimensiones del tablero. Por estos motivos decidimos mantener la complejidad independiente y usar únicamente una estructura de lista. De esta manera, para instancias del juego con tableros grandes y pocas fichas o bien para juegos en un estado avanzado, en los que quedan pocas fichas, logramos mayor eficiencia.

En cuanto a la correctitud del algoritmo, podemos comenzar pensando en el caso base de la recursión. Como ya dijimos, este se alcanza cuando no quedan movimientos por ejecutar. Esto puede deberse a que quede más de una ficha sin posibilidad de moverse (se perdió) o que quede sólo una (se ganó). Cada paso recursivo del algoritmo en la i-ésima recursión consiste en intentar ganar un juego con $t_i - 1$ fichas, donde t_i es a su vez $t - i$. La ficha que se elimina proviene de ejecutar un movimiento válido. Es decir, el algoritmo clona el juego, ejecuta un movimiento y se llama recursivamente con una ficha menos.

Cuando se llega a un caso base, se devuelve esa solución. El juego *padre* que generó esa instancia retornada, se fija si su sub-instancia ganó. En ese caso, termina. En caso contrario, ejecuta otro movimiento diferente e intenta seguir esa nueva rama. Al obtener una solución ganadora o al no tener más ramas que ejecutar, el algoritmo termina. Sólo en caso de poder ganar, lo que se obtiene es una sucesión de movimientos válidos que llevan desde el juego inicial provisto hasta su caso base. Es decir, se obtiene la solución al problema.

3.3. Resultados y Discusión

Para analizar el algoritmo desarrollado para este ejercicio, probamos en primer lugar algunas instancias simples a partir de las cuales pudiéramos seguir los pasos del resultado expuesto y asegurar que esta sucesión de pasos nos llevara a ganar el juego o a perder, en caso de no ser posible. Probamos algunos casos sin solución, por ejemplo por no poder ni siquiera mover una ficha, o casos en los que se ejecutaran casi todos los movimientos posibles hasta llegar a sólo dos fichas en lugares separados del tablero. De la misma manera, algunos pequeños casos en los que se llegara a ganar, con dos fichas adyacentes, o una tercera a dos posiciones.

Entendimos que por la naturaleza recursiva del algoritmo no era necesario hacer pruebas más grandes para ver empíricamente la correctitud. Teniendo casos base que se resuelven bien y suponiendo que sé resolver cualquier instancia con tamaño n, sabiendo que en un juego de $n + 1$ fichas, con algún movimiento posible, al ejecutarlo, obtenemos uno de tamaño n, ejecuto ese movimiento y obtengo una nueva instancia que sé resolver. En caso de no haber más movimientos posibles, significaría que llegué a un caso base.

Una vez hecho esto, procedimos a realizar mediciones de complejidad, para lo cual generamos instancias de entrada aleatoriamente. Con este fin utilizamos la clase *RandomGenerator*. Dados los parámetros *cant, n, m, chance* se genera un archivo con *cant* instancias de prueba, cada una con un tablero de tamaño aleatorio de entre una y *n* filas y entre una y *m* columnas, donde cada casilla tiene o no una ficha con una probabilidad $0 \leq$ *chance* ≤ 1.

Consideramos como tamaño de la entrada a la cantidad de fichas puestas sobre el tablero. Una primera impresión es que según esta decisión, no hay diferencia entre un tablero de 3×3 con uno de 30×30, siempre y cuando tengan la misma cantidad de fichas. Efectivamente la complejidad de nuestro algoritmo fue calculada a partir de la cantidad de fichas sobre el tablero, las cuales condicionan la cantidad de movimientos posibles y en consecuencia, la cantidad de pasos recursivos a ejecutar.

Para contar la cantidad de operaciones utilizamos contadores dentro de la implementación del algoritmo. Estos son incrementados cada vez que se realiza un conjunto de operaciones de costo constante dentro de un ciclo con una cantidad de iteraciones dependiente del tamaño de la entrada. Una observación importante es que a priori, por la naturaleza del algoritmo, sería razonable que la cantidad de operaciones efectivamente ejecutadas no siempre alcanzara la cota de complejidad, ya que en cuanto se encuentra una solución, el algoritmo se detiene.

Las mediciones realizadas fueron sobre 3 conjuntos de instancias:

1. 15 Tableros de un máximo de 7 filas y 7 columnas con una probabilidad de tener una ficha de 0.4.
2. 20 Tableros de un máximo de 6 filas y 6 columnas con una probabilidad de tener una ficha de 0.6.
3. 30 Tableros de un máximo de 5 filas y 5 columnas con una probabilidad de tener una ficha de 0.8.

Los tamaños de la entrada (cantidad de fichas) fueron pensados, en un comienzo, de mayor tamaño. La complejidad del algoritmo hace que su tiempo de ejecución pueda ser muy largo lo cual dificultó la realización de los experimentos. Por ello fuimos disminuyendo sus tamaños hasta llegar a los descritos más arriba. En cuanto a la calidad de los datos, se realizaron sucesivas ejecuciones de cada uno de los generadores aleatorios y se eligieron ciertos tableros que para nuestro criterio tenía sentido evaluar y que aportaban diversidad al experimento. Por ejemplo, algo importante fue que hubiera diferentes casos de igual cantidad de fichas. Más adelante analizaremos por qué.

Para graficar, utilizamos escala logarítmica debido a la gran dispersión de los datos en cuanto a escala, ya que la función de complejidad tiene una tasa de crecimiento muy grande. En todos ellos incluímos en azul la función de complejidad calculada $f(t) = (4t)^{t+1}$ y en verde $g(t) = t^{t+1}$.

Figura 7: Gráfico de cantidad de operaciones en función del tamaño de la entrada para instancias generadas aleatoriamente con tableros de tamaño máximo de 7×7 y elementos con probabilidad de aparición de 0.4 (Escala logarítmica).

Figura 8: Gráfico de cantidad de operaciones en función del tamaño de la entrada para instancias generadas aleatoriamente con tableros de tamaño máximo de 6×6 y elementos con probabilidad de aparición de 0.6 (Escala logarítmica).

Figura 9: Gráfico de cantidad de operaciones en función del tamaño de la entrada para instancias generadas aleatoriamente con tableros de tamaño máximo de 5×5 y elementos con probabilidad de aparición de 0.8 (Escala logarítmica).

Podemos ver en las figuras 7, 8 y 9 que en todos los casos, la cantidad de operaciones se mantuvo por debajo de la complejidad calculada pero con una forma que tiende a la cota. Como mencionamos anteriormente, esto se debe a que es una cota de complejidad de caso peor y en todos estos casos, el algoritmo pudo encontrar una solución válida antes de recorrer todas las posibles.

Por otra parte, podemos apreciar particularmente en las figuras 8 y 9 que tenemos varias instancias de igual tamaño de entrada con diferente costo. La explicación de este fenómeno es que el algoritmo se detiene al encontrar una solución, lo cual hace que según la naturaleza de la distribución de las fichas, por maś que la cantidad sea la misma, puede llevar más tiempo encontrar (si existe) una solución o toparse con su no existencia.

Analizando la diferencia entre las funciones $f(t) = (4t)^{t+1}$ y $g(t) = t^{t+1}$, vemos que inclusive esta última es, en la mayoría de los casos, cota suficiente. Recordemos que el número 4 era una cota de la cantidad de movimientos realizables por cada ficha. Como fue descrito en la sección **Desarrollo**, es una cota grande ya que por ejemplo, fichas en los bordes del tablero no pueden moverse para todas las direcciones, o que una ficha pueda moverse para las 4 imposibilita que alguna de sus adyacentes lo haga.

3.4. Conclusiones

En base a lo analizado, podemos afirmar que la cota de complejidad calculada ($O((4t)^{t+1})$) es satisfactoria para los casos generados. Si bien vimos que es una cota que puede considerarse *grosera*, no existe una forma de ser más precisos debido a la amplia cantidad de casos que pueden generarse, por la naturaleza del problema. Puede haber casos de muchas fichas que luego de pocos movimientos no tengan solución, otros de menos en los cuales existan pocas, las cuales puedan llegar a partir de comenzar por mover una de las últimas fichas.

Por otra parte, el hecho de ser una complejidad tan grande hace que la ejecución de entradas que bajo alguna definición puedan considerarse *grandes* genera problemas en el tiempo de ejecución, el cual crece supraexponencialmente.

4. Referencias

1. Apunte de Algoritmos Básicos, Cátedra de Algoritmos y Estructuras de Datos II, Depto. de Computación, Facultad de Cs. Exactas y Naturales, U.B.A. (**`http://dl.dropbox.com/u/31338007/apunte-algoritmos.pdf`**)

2. Oracle Java Official Documentation (**`http://docs.oracle.com/javase/1.4.2/docs/api/java/util/LinkedList.html`**)