DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Benitti, Raul	592/08	raulbenitti@gmail.com
Mengarda,Lucas	787/10	l.j.mengarda@gmail.com
Scarpino, Gino	392/08	gino.scarpino@gmail.com
Vallejo, Nicolás	500/10	nico_pr08@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

Índice

1. Problema 1

1.1. Introducción

El problema que se plantea es el siguiente: En una fábrica de quesos hay n máquinas que trabajan en forma independiente, cada una produciendo un determinado tipo de queso. Una vez encendida, la máquina i tarda p_i minutos en finalizar su producción. Por cuestiones de seguridad, las máquinas comienzan el día con sus tanques de combustible totalmente vacíos, y luego de que se llene su tanque, tarea en la que se demora c_i minutos, la máquina i comienda a producir sin ningun otro tipo de intervención. Como se cuenta con un único surtidor para todas las máquinas, la carga de los tanques debe realizarse de manera secuencial. Sin embargo, el tiempo que se tarda en mover el surtidor de una máquina a otra es despreciable.

Lo que se desea es encontrar un orden para la carga de combustible de manera tal que la producción completa de todas las máquinas se termine lo antes posible. El algoritmo que resuelva el problema debe tener una complejidad menor o igual a $O(n^2)$

P_i	10	4	6
c_i	1	2	4
i	1	2	3

Figura 1: Ejemplo de una instancia del problema. Se tiene tres máquinas, con sus respectivos tiempos de produccion y carga.

1.2. Desarrollo

Sea O un orden en que las máquinas son cargadas. Diremos que una máquina i demora en terminar lo que se tarda en esperar a que sea su turno de ser cargada, más su tiempo carga y de producción. El tiempo de espera equivale a la suma de los tiempos de cargas de las máquinas que son tratadas antes que ella. Entonces, definimos $T_O(i)$, el tiempo total que demora en terminar la i-esima máquina dentro del orden O, como

$$T_O(i) = \sum_{j=1}^{i} c_{O[j]} + P_{O[i]}$$

El tiempo que demora en terminar la producción completa es equivalente al tiempo que demora en finalizar su producción la ultima máquina. Definimos, entonces, T(O) como

$$T(O) = \max_{1 \leq i \leq n} T_O(i)$$

Buscamos encontrar un orden O tal que minimice T(O).

P_i	10	4	6
c_i	1	2	4
i	1	2	3
O[i]	1	2	3
$T_O(i)$	11	7	13

(a)

P_i	10	4	6
c_i	1	2	4
i	1	2	3
O[i]	1	3	2
$T_O(i)$	11	11	11

(b)

Figura 2: En estas figuras se pueden observar dos posibles órdenes de carga de máquinas. En **(a)** se sigue el orden O según la numeracion de las máquinas, con un resultado T(O) = 13. En cambio, en **(b)** se muestra un orden O óptimo, con resultado T(O) = 11.

La primera idea intuitiva que surge es tratar de tener la mayor cantidad de máquinas funcionando en simultáneo. Así, al momento de seleccionar cual es la siguiente máquina que debe ser cargada, tomamos aquella que tiene mayor tiempo de producción de manera que las máquinas restantes lleven a cabo sus trabajos mientras esta se encuentra produciendo.

Otra forma de verlo es la siguiente: como se quiere que todas las máquinas terminen lo antes posible (de manera que la ultima en hacerlo lo haga lo antes posible), y considerando que, para comenzar su producción, la i-esima máquina debe esperar su tiempo de carga c_i mas la suma de los tiempos de cargas de las máquinas anteriores, lo que buscamos es que la máquina con mayor tiempo de producción comience lo antes posible (que tenga el menor tiempo de espera posible); es decir, que sea la primera en cargarse dentro del conjunto de máquinas disponibles. Esta idea se ajusta claramente a la forma de un algoritmo goloso:

- decisión golosa: seleccionar y cargar la máquina con mayor P_i
- ordenar las máquinas todavia no cargadas de forma tal que la producción completa de estas termine lo antes posible

A partir de aquí se puede plantear la solución como una funcion recursiva

$$Orden(maquinas) = [seleccionGolosa(maquinas)] + Orden(maquinas - \{i\})$$

donde *Orden* devuelve un lista, *maquinas* es un conjunto de máquinas y *seleccionGolosa* devuelve la máquina cuyo tiempo de producción es el mayor que entre los las máquinas del conjunto. En la sección de correctitud veremos que no es necesario considerar los tiempos de carga al tomar la decisión golosa.

Ahora, el resultado de esta idea equivale simplemente a ordenar las máquinas de manera decreciente segun sus tiempos de producción. Por esto, en lugar de desarrollar un algoritmo en particular, utilizaremos alguno de los algoritmos de sorting ya conocidos (en este caso, dado la complejidad requerida, cualquiera de ellos es válido).

1.3. Algoritmo

El algoritmo *resolver* toma como argumentos de entrada los tiempos de carga (c) y procesamiento (P), y como argumentos de salida, el orden (O, un vector de n elementos, donde O[i] = i) y el tiempo total (T), y utiliza la implementación de sorting de la plataforma.

El algoritmo *calcTiempo*, que calcula el tiempo $T(O)$, solo recorre el orden O calculando los tiempos $T_O(i)$, y devuelve el máximo de todos ellos.

Algoritmo 1.1

RESOLVER(P, c, O, T)

1 Ordenar O segun el ordenamiento descendiente de los elementos en, de manera que P[O[i]] $\leq$ P[O[i+1]] para i = 1..n-1
2 $T =$ CALCTIEMPO(P, c, O)

Algoritmo 1.2

CALCTIEMPO(P, c, O)

```
1  resultado = 0
2  inicio = 0
3  for i in O
4        t_i = inicio + c[i] + p[i]
5        if (resultado < t_i)
6              resultado = t_i
7        inicio = inicio + c[i]
8  return resultado
```

1.3.1. Correctitud

Como mencionamos anteriormente, el algoritmo se basa en una idea golosa. Para demostrar la correctitud de ésta, observemos los siguientes puntos:

- EL problema presenta subestructura óptima:
  Como ya vimos, el problema de encontrar el orden óptimo puede plantearse como uno en el que se realiza una elección, y se resuelve un subproblema del

mismo tipo pero de tamano menor. Ahora, supongamos que el orden O es una solución óptima al problema original; es decir, el valor de $T(O)$ es el menor posible. Llamemos S = O[2..n] a la solución del subproblema que queda al elegir el primer elemento de O, de forma que $O = O[1] + S$. Si S no es un solución óptima para ese subproblema, podriamos encontrar algun otro orden S' que si lo fuera, tal que se cumpla $T(S') < T(S)$. Consiremeos entonces una nueva solución $O' = O[1] + S'$. Dado el caso en que $T_O(1) \leq T(S')$, al calcular $T(O')$ tendriamos

$$T(O) = max(O[1], T(S)) < max(O[1], T(S')) = T(O')$$

Sin embargo, esto no puede suceder pues supusimos O un orden óptimo. Por lo tanto, la subsolución S debe ser, tambien ella, óptima.

- Existe una solucion óptima que tiene, como primera elección, la elección golosa que propone el algoritmo:
  Supongamos que O es óptimo, y sea g la máquina con mayor tiempo de producción.
  Si $O[1] = g$, entonces se cumple lo que buscabamos.
  Si $O[1] \neq g$, entonces debe existir algun otro j, $1 < j \leq n$ tal que $O[j] = g$. Supongamos que modificamos O intercambiando su primer elemento con el elemento j-esimo, obteniendo asi un nuevo orden O' que cumple

  - $O'[1] = O[j] = g$,
  - $O'[j] = O[1]$ y
  - $O'[k] = O[k]$ para todo k distinto de 1 y j.

  Queremos ver que O' sigue siendo óptimo, o sea, que $T(O') \leq T(O)$. Notemos, en primer lugar, que para todo k tal que $j < k \leq n$ se cumple $T_{O'}(k) = T_O(k)$, pues, considerando que los primeros k elementos de O' son solo una permutacion de los primeros k elementos de O,

$$T_{O'}(k) = \sum_{i=1}^{k} c_{O'[i]} + P_{O'[k]} = \sum_{i=1}^{k} c_{O[i]} + P_{O[k]} = T_O(k)$$

  Por otro lado, observemos que en O[1..j] tenemos que $T_{O'}(j) = \text{máx}_{1 \leq i \leq j} T_O(i)$ ya que teniendo en cuenta que los c_i son naturales, que $k < j$ y que $P_{O[k]} \leq P_{O[j]} = P_g$,

$$T_O(k) = \sum_{i=1}^{k} c_{O[i]} + P_{O[k]} \leq \sum_{i=1}^{k} c_{O[i]} + \sum_{i=k}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

  Entonces, lo que necesitamos ver es que tambien suceda $T_O(j) \geq \underset{1 \leq i \leq j}{\text{máx}} T_{O'}(i)$.
  Para las máquinas intercambiadas tenemos

$$T_{O'}(1) = \sum_{i=1}^{1} c_{O'[i]} + P_{O'[1]} \leq \sum_{i=1}^{j} c_{O'[i]} + P_{O'[1]} = \sum_{i=1}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

$$T_{O'}(j) = \sum_{i=1}^{j} c_{O'[i]} + P_{O'[j]} \leq \sum_{i=1}^{j} c_{O'[i]} + P_{O'[1]} = \sum_{i=1}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

Para las demas máquinas (k en el rango 2..j-1),

$$T_{O'}(k) = \sum_{i=1}^{k} c_{O'[i]} + P_{O'[k]} \leq \sum_{i=1}^{k} c_{O'[i]} + \sum_{i=k}^{j} c_{O'[i]} + P_{O'[k]} \leq \sum_{i=1}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

Notemos que no hara falta considerar el orden segun los tiempo de carga: supongamos que tenemos una seguidilla $P_k...P_l$ donde $P_i = p$ para todo $k \leq i \leq l$. Sea $t = \sum_{i=1}^{k-1} c_i$

El tiempo que tarda en terminar la tarea P_l es $t + \sum_{i=k}^{l}(c_i) + p$.

El tiempo que tarda en terminar una cualquiera de las otras tareas en el grupo, sea P_j que cumple $i \leq j < l$, es $t + \sum_{i=k}^{j}(c_i) + p$.

De esto se tiene que

$$t + \sum_{i=k}^{j}(c_i) + p < t + \sum_{i=k}^{j}(c_i) + \sum_{i=j}^{l}(c_i) + p = t + \sum_{i=k}^{l}(c_i) + p$$

Es decir, no importa como se permuten los valores c_i, un grupo de máquinas con igual tiempo de producción siempre tardará lo mismo en terminar, y este valor será la suma de todos los tiempos de carga del grupo (y las máquinas anteriores, si es que hay) más el tiempo de producción p.

1.3.2. Análisis de complejidad

Sea n la cantidad de máquinas.

El algoritmo de sorting que se utiliza esta implementado en la STL de C++. Se asegura que el mismo tiene complejidad $O(nlog(n))$ [1]. El algoritmo *calcTiempo* solo tiene un loop que itera n veces, dentro del cual todas las operaciones son de orden O(1), lo que resultan en una complejidad final del orden de O(n).

Por lo tanto, el algoritmo *resolver* tiene complejidad $O(nlog(n)+n) = O(nlog(n))$, con lo que se cumple el requerimiento de encontrar una solucion en tiempo menor a $O(n^2)$

1.4. Pruebas

No distinguimos casos particulares que sean de interés específico en analizar, pues nuestra implementación depende de la implementación del algoritmo de sort de la STD, y se asegura que este no presenta casos patológicos. Por otra parte, a

[1] http://www.sgi.com/tech/stl/sort.html

priori podría parecer útil analizar casos como aquellos en los que se encuentran uno o más bloques de maquinas con los mismos tiempos de produccion, o probar distintas relaciones entre los tiempos de carga y producción. Sin embargo, estos detalles no son tenidos en cuenta en el algoritmo en sí, por lo que utilizarlos para separar familias de pruebas resulta superfluo. Dicho esto, presentamos un gráfico donde se muestra como en la práctica el algoritmo se ajusta a la la complejidad predicha.

Figura 3

1.5. Conclusiones

En este ejercicio observamos como un problema, luego de ser analizado con un poco de detenimiento, puede convertirse en una instancia de otro problema para el cual pueden ya existir algoritmos satisfactorios. En ese caso, todo el proceso psoterior de implementación y análisis se simplifica.

2. Problema 2: Sensores defectuosos

2.1. Introducción

Se tiene un conjunto de sensores con sus respectivos intervalos de tiempo de medición y se conoce el número de la medición que falla. Se desea conocer el id del sensor que falló.

2.2. Desarrollo

Encaramos este problema rearmando la secuencia de mediciones de los sensores. Consiguiendo ésto, la solución al problema sería fijarse en esta secuencia, el id que figura en la posición de medición que falló.

Para armar la secuencia de mediciones, decidimos ir guardando para cada sensor en que instante de tiempo tendría que volver a realizar una medición. A medida que determinamos el sensor que va a medir, calculamos su siguiente tiempo de medición.

Teniendo un diccionario donde para cada sensor se puede obtener en qué momento le toca realizar una medición, se puede conocer al próximo que le toca medir, ya que es el mínimo de los tiempos.

En el momento inicial todos los sensores miden, por lo que la secuencia inicial no es vacía, sino que contiene todos los números de sensores ordenados por su id ya que desempatan por éste.

En cada paso de nuestro algoritmo, agregamos a la secuencia el siguiente sensor que midió y calculamos su próxima medición actualizando la tabla de próximas mediciones

Cuando la secuencia alcanza el tamaño igual al número de medición que falló, terminamos y el resultado es la última posición de esa secuencia.

Aclaración: en la implementación del diccionario se usa directamente un minHeap. Más adelante se explica el motivo de tal decisión.

Pseudocódigo:

Algoritmo 2.1 Entero sensorDefectuoso(Conjunto sensores, Entero medDefectuosa)

```
1: Diccionario diccTiempos = sensores
2: Entero medicion = | sensores |
3: for all Sensor s in sensores do
4:    agregar(mediciones,id(s))
5: end for
6: while medicion ¡medDefectuosa do
7:    proximo = proxSensor(diccTiempos,sensores)
8:    agregar(mediciones,proximo)
9:    medicion = medicion + 1
10: end while
```

Donde

medDefectuosa es el número de medición que falla.

diccTiempos es el diccionario que para cada sensor guarda el tiempo de su próxima medición.

sensores es el conjunto de sensores.

medicion es el número de medición parcial. Como al inicio hubo la cantidad de sensores, se inicializa con esa cantidad.

mediciones es la secuencia de mediciones con los ids de los sensores.

proxSensor es la funcion que devuelve el id del mínimo de los tiempos que están presentes en el diccionario, y además actualiza el tiempo para la próxima medición.

2.2.1. Correctitud

En alguna instancia del problema, si tuviéramos la siguiente tabla con los momentos t_i para la siguiente medición de cada sensor s_i con $(1 \leq i \leq n)$ donde n es la cantidad total de sensores:

s_1	s_2	...	s_{n-1}	s_n
t_1	t_2	...	t_{n-1}	t_n

determinar el siguiente sensor que le toca realizar su medición, sería elegir el de menor t. Caso contrario, se estaría eligiendo un sensor con tiempo mayor, por lo que haría una medición anterior cuando en realidad le correspondería a un sensor con menos tiempo, generando una incongruencia en el orden de mediciones.

Por lo que alcanzaría con ver que nuestro algoritmo genera esa tabla/diccionario de forma correcta.

En el instante inicial, el diccionario contendrá los tiempos de intervalos de medición ya que todos midieron de entrada. Para el primero, elegimos el mínimo de esos tiempos. Una vez que sabemos el id, lo agregamos a la secuencia de mediciones, y redefinimos en el diccionario su tiempo de la siguiente forma:

$$t'_i = t_i + intervalo(s_1)$$

es decir, que al tiempo que ya figuraba en el diccionario, se le agrega el tiempo de intervalo para determinar el tiempo de su próxima medición. Por lo que, en cada

paso donde se elige el sensor que mide, se actualiza el diccionario. De esta forma, se mantiene la correctitud del significado del diccionario una vez elegido el próximo sensor a medir.

Se repite este proceso hasta llegar a la medición que falla inclusive, con lo que el id del sensor quedaría guardado en la secuencia de mediciones.

2.2.2. Análisis de complejidad

Se cargan los datos de los sensores en vectores. En uno, el tiempo de intervalo entre mediciones. En otro, una tupla que representaría la primer componente el id del sensor y la segunda componente el tiempo de próxima medición. A la vez que se cargan los datos, ya se colocan los ids por orden en el vector de mediciones ya que todos miden en el instante inicial.

La complejidad de lo anterior es O(n), donde n es la cantidad de sensores.

En el caso que el número de medición sea menor a la cantidad de sensores, el algoritmo termina devolviendo el valor del vector de mediciones según la posición correspondiente a la medición que falla.

Caso contrario, calculamos la secuencia de mediciones. Para ello necesitamos el ya mencionado diccionario. Decidimos utilizar un min-Heap como diccionario. Esto nos permitiría obtener el mínimo valor de tiempo de próxima medición en O(log n) siendo n la cantidad de sensores. Crear el minHeap tiene un costo de $O(3 * n)$.[2].

Una vez creado el minHeap, comenzamos un ciclo donde se busca el siguiente sensor que mide. Al ser un minHeap, tiene costo O(1) pero luego hay que actualizar el valor y el diccionario utilizando *push_heap* y *pop_heap*, ambas con costo $O(logn)$[3], [4]. Como ya dijimos, al usar un minHeap, ésto tiene un costo de O(log n). Luego, se agrega el id del sensor a la secuencia de mediciones con un costo constante.

La cantidad de iteraciones del ciclo es el número de medición que falla menos la cantidad de sensores existentes. Ésto se debe a que, en el momento inicial, todos los sensores realizan una medición. Por lo que de entrada se tienen n mediciones.

Por lo tanto, la complejidad total del algoritmo termina siendo:

$$\sum_{i=k}^{n} logn = (k - n) * logn \leq k * n$$

Termina siendo estrictamente menor a $O(k * n)$ como se pedía en la consigna.

2.3. Resultados

Para analizar el comportamiento del nuestro algoritmo, decidimos generar tests aleatorios donde en uno se va ir variando la cantidad de sensores mientras que el

[2]http://www.sgi.com/tech/stl/make_heap.html
[3]http://www.sgi.com/tech/stl/push_heap.html
[4]http://www.sgi.com/tech/stl/pop_heap.html

número de falla va a depender de esa cantidad. En el mismo nos queda una gráfica:

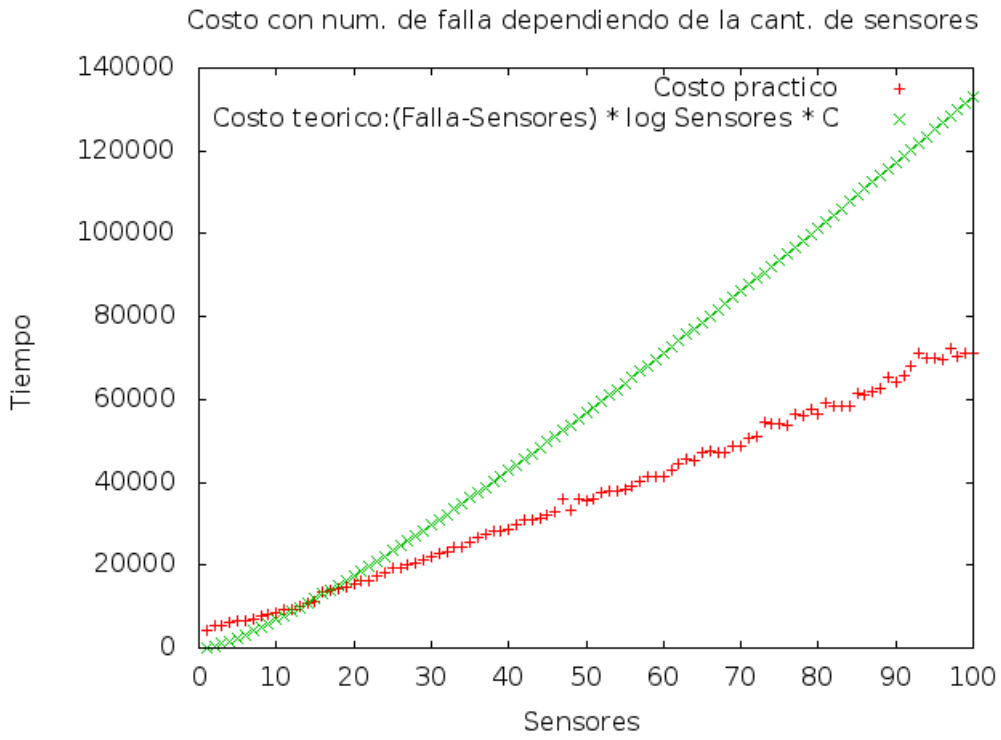

Figura 4: Se compara el tiempo que tardó en promedio en obtener un resultado contra el tiempo teórico del peor caso

Armamos el test de la siguiente manera. Vamos a hacer instancias que vayan de 1 sensores a 100 y por cada cantidad de sensores realizamos 1000 repeticiones. Decidimos que la falla se produzca en la medición: $4 * n + r$. Donde n es la cantidad de sensores y r un valor aleatorio entre 1 y n.

Para cada ejecución guardamos el tiempo que tardó usando la función *clock-gettime* y luego contrastamos con la cota teórica calculada anteriormente.

Para los dos siguientes test, fijamos la cantidad de sensores. En uno poca cantidad de sensores y en otro una cantidad considerable. El número de falla lo fuimos variando desde 1 hasta n * 50 donde n es la cantidad de sensores utilizados.

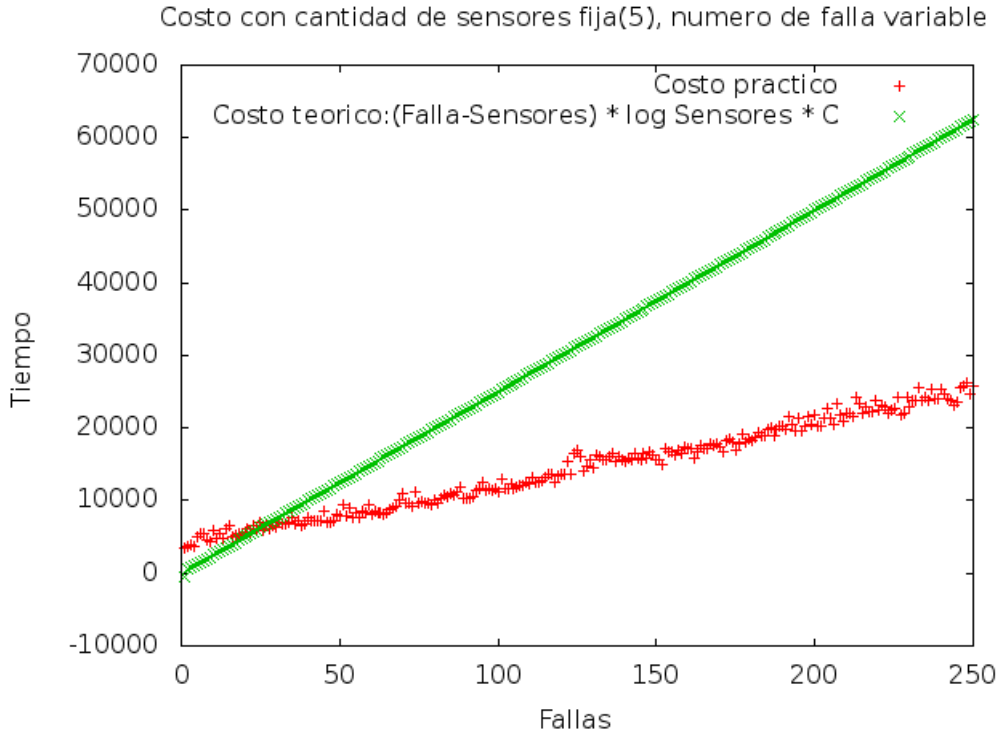

Figura 5

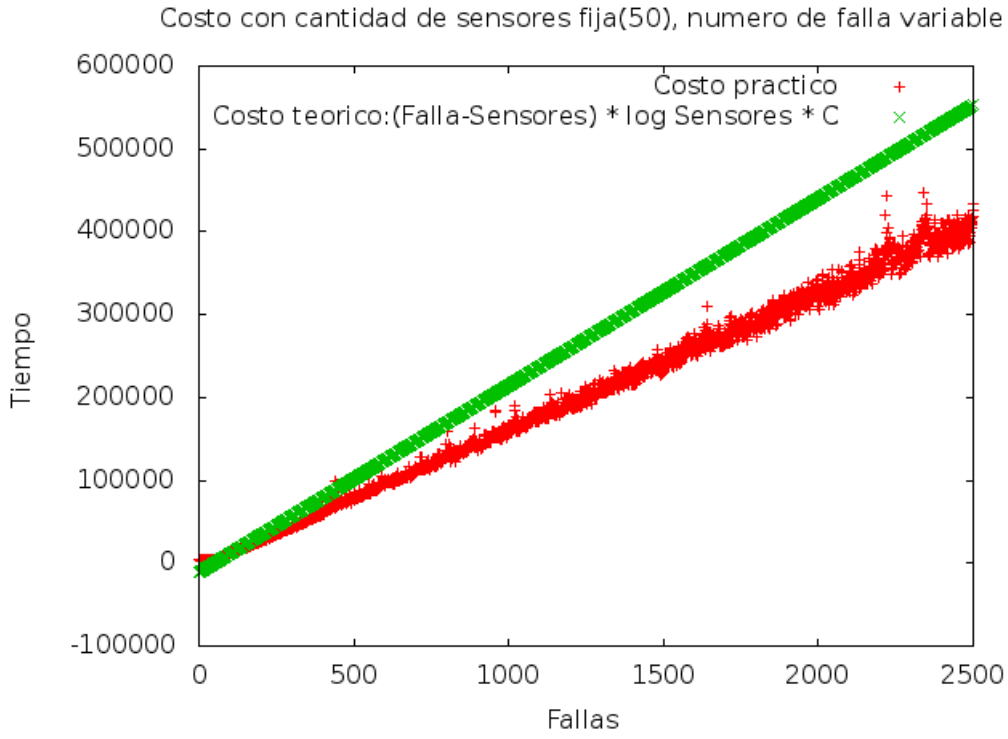

Figura 6

Se decidión arbitrariamente que los tiempos de los sensores entre 1 y 20 elegidos aleatoriamente.

2.4. Conclusiones

Como era de esperar, los resultados mostraron que el tiempo de ejecución es asintóticamente menor al tiempo teórico calculado para el peor caso. Siendo ese tiempo, menor al pedido por la consigna del trabajo práctico.

Analizar bien las posibles estructuras de datos a utilizar fue fundamental para volver más eficiente el algoritmo. El problema de obtener en cada instante el mínimo de un conjunto de valores utilizando un minHeap logró bajar la complejidad lineal a logarítmica.

Luego, también era necesario considerar la parte del problema en que todos los sensores realizan una medición en el momento inicial. Lo que permite realizar n iteraciones menos de nuestro algoritmo, donde n es la cantidad de sensores.

La implementación se podría mejorar evitando construir el vector de mediciones de los sensores. Basta con llevar un contador, que con cada medición se incremente. Determinando así el sensor que midió erróneamente cuando el contador alcance el número de medición fallida.

3. Problema 4: Puzzle de casilleros

3.1. Introducción

Para este tipo de problemas, se suele usar la técnica de *backtracking*. No se conocen algoritmos polinomiales para resolver el problema. También se podría resolver con fuerza bruta pero la ventaja de esta técnica es que se busca armar e ir probando posibles soluciones. A medida de que se van construyendo, se puede determinar ciertos casos que no son posible solución, por lo que no se intentaría resolver el problema en esos casos evitando ejecuciones innecesarias. En el peor caso, *backtracking* es igual a intentar resolverlo por fuerza bruta. Se nos pide encontrar, si existe, alguno de los menores subconjuntos de piezas de tal forma que cubran todo el tablero. Donde cada casillero del tablero es de color negro o blanco, y cada pieza de forma rectangular, posee casilleros de los mismos colores. La pieza para poder ubicarse en cierta posición del tablero tiene que coincidir exactamente en el tablero y se la puede rotar.

3.2. Desarrollo

Este ejercicio contiene varios problemas. Para empezar, nos encontramos con el requerimiento de dado un conjunto armar todos los subconjuntos posibles. Se conoce como *power set*. La complejidad de este problema es exponencial, no se conocen algoritmos polinomiales hasta el momento.

La primera implementación del problema de power set consistía en usar una máscara de bits para determinar si un elemento estaba presente o no en un conjunto. Se recorrían los números desde el 1 hasta $2^{|piezas|}$ formando así todas las posibles combinaciones de subconjuntos. Era simple pero tenía el problema de que podía producir overflow en arquitecturas donde $|piezas|$ sea mayor a la cantidad máxima de bits para algún registro que sea utilizado como contador del procesador. Ej: con 40 piezas y un procesador de 32 bits.

Por lo tanto, decidimos implementar este problema mediante la recursión...

Para resolver el problema principal, lo que hicimos fue ir probando cada uno de los subconjuntos con tamaño desde 1 hasta la cantidad de piezas. Como punto de corte razonable, es haber encontrado algún subconjunto como solución.

Para determinar si cierto subconjunto de piezas es solución, primero determinamos una poda. Esta consiste en que, al buscar una solución óptima, el subconjunto de piezas debía de cubrir exactamente el tablero. Si la superficie de las fichas era menor al del tablero, entonces no era solución. En cambio, si era mayor, podría ser solución, pero si lo era, no era óptima porque sobraba alguna pieza.

Luego implementamos una función recursiva en la cual para cada ficha se buscaba sus posibles posiciones en el tablero. Para cada posición, se creaba una instancia del tablero en la cual se la ubicaba ahí. Luego se llamaba devuelta a la función con ese tablero y sin esa pieza.

El caso base de la función es cuando no quedan más piezas para ubicar. Entonces se fija en el tablero si está completamente cubierto, en caso afirmativo, se encontró una solución al problema.

La función tiene en cuenta las rotaciones de las piezas. Por lo que si no se encontrón solución, se rota la pieza. Si es idéntica a la original, se corta esa rama. Caso contrario, se busca todas las posibles posiciones para esa pieza rotada, y se repite lo anterior.

Una vez que se encuentra solución, si existe, se devuelve el vector con la información de las piezas colocadas en el tablero. Éste vector, lo maneja la clase *Tablero* que hicimos, al cual agrega una pieza cada vez que es posible ubicarla en el tablero.

En todo momento del desarrollo tuvimos que tener en cuenta generar todas las podas posibles para evitar ejecuciones innecesarias debido al tipo de problema.

Creamos las clases *Tablero* y *Pieza* para que cada se encargue de las funciones que les competen propiamente a dicha abstracción.

En la clase *Tablero*, podemos crear un tablero con el tamaño deseado, obtener las posibles posiciones de una pieza, si encaja la pieza en cierta posición, si está completo o no. Si cierto conjutno de piezas puede llegar a cubrir todo el tablero, obtener las piezas ubicadas, etc.

En la clase *Pieza*, podemos crear una pieza de cierto tamaño, rotarla, etc.

Utilizamos top-down y clases porque nos pareció la mejor forma de organizar el programa para resolver el ejercicio. Haciendo más prolijo el código y *tirar* los problemas del tablero se encarga su clase y el de la pieza el de ella misma.

3.2.1. Correctitud

En cuanto la problema de la generación de todos los subconjuntos posibles de un conjunto, utilizamos un método que dada la representación binaria de un número los elementos presentes en un conjunto son aquellos donde la posición es igual al id del elemento y el bit está en 1, es decir, es un elemento presente. Como se recorre desde el valor 1 hasta 2^n (n es cantidad de elementos), se recorre todas las posibles combinaciones de elementos presentes y no presentes. Si bien esta forma sigue siendo un algoritmo exponencial, creemos que es más eficiente que haber hecho una función recursiva que genere todos estos conjuntos.

3.2.2. Análisis de complejidad

Las funciones que utlizamos del tablero son:

- **getColor:** dada una posición del tablero, devuelve el color, negro o blanco. Tiene un costo O(1) debido a que solo tiene que devolver un valor accediendo a una posición de una matriz.
- **posiblesPosiciones:** recorre todo el tablero, fijándose en cada posición, si es posible ubicar la pieza ahí. Para esto llama a la función encaja. Guarda en un

vector resultado las posiciones posibles de ubicar la pieza. La complejidad termina siendo la cantidad de posiciones del tablero (filas por columnas) iteraciones de la función encaja: O(filas * columnas * filasPieza * columnasPieza)

- **encaja:** dada una pieza y una posición, devuelve verdadero si es posible ubicar esa pieza en la posición pasada por parámetro. Para esto, recorre toda la superficie de la pieza, por lo que cuesta O(filasPieza * columnasPieza)
- **ocupado:** dada una posición, devuelve si ya está ocupada por una pieza o no. Se tiene que acceder a una matriz, por lo que cuesta O(1).
- **completo:** determina si el tablero está totalmente cubierto por piezas. Para ello, tiene que recorrer todas las posiciones y llamar a la función ocupado. Tiene coste O(filas * columnas)
- **ubicarFicha:** dada una pieza y una posición modifica la instancia de tablero, seteando las posiciones correspondientes para considerarlas ocupadas. Ademaá, guarda en un vector el id de la pieza, el grado de rotación de la misma, y la posición del extremo superior izquierdo. Tiene un costo O(filasPieza * columnasPieza)
- **obtenerPiezas:** la clase tablero guarda un vector con las piezas ubicadas en el mismo. Esta función devuelve una copia de ese vector. Tiene costo O(n) donde n es la cantidad de piezas.
- **cubreTodo:** es una función para podar casos no posibles de ser solución como se explicó anteriormente. Calcula la superficie del tablero. Recorre todas las piezas, restando a la superficie del tablero la superficie de la pieza. Para cubrirlo exactamente tiene que quedar en 0. Caso contrario, se poda. Tiene un costo de O(n) donde n es la cantidad de piezas.

Vistas estas consideraciones, analicemos la complejidad de *resolverJuego*. Lo haremos en función de la cantidad de piezas.

Primero hagamos una aclaración: si bien es cierto que muchas de la funciones que usa este métodos basan su complejidad en el tamaño de la pieza que está probando en ese momento, hemos decidido acotar la cantidad de filas de una pieza por la cantidad de filas del tablero y lo mismo para la cantidad de columnas. Esto es porque al momento de ejecutar *resolverJuego* es esperable que todas las piezas que se analizan tienen dimensiones menores o iguales a las del tablero. Por lo tanto, la cota de complejidad que mostraremos en esta sección es quiźa burda.

Para esta análisis definiremos k como las dimensiones del tablero. Es decir que k es la cantidad de filas del tablero por la cantidad de columnas del tablero.

Como se puede observar, *resolverJuego* es una función recursiva en la cantidad de piezas. En el caso en que la cantidad de piezas sea nula, lo único que se chequea es si el tablero está completo en tiempo O(k). Si, en cambio, quedan piezas por analizar, el algoritmo chequea si se puede agregar al tablero alguna de las rotaciones de la pieza y luego se llama recursivamente a la función pero sin esa pieza. En el peor

caso, lo que puede ocurrir es que se tenga que chequear las cuatro rotaciones de la pieza.

Además de llamarse recursivamente el algoritmo realiza algunas operaciones. Por cada rotaciones, se fija las posibles posiciones con un costo de $O(k * filasPieza * columnasPieza)$. Hemos dicho que vamos a acotar las filas y las columnas de la piezas por las del tablero. Luego, supongamos que buscar las posibles posiciones cuesta $O(k^2)$.

Copiar la pieza que vamos a analizar cuesta O(filasPiezas*columnasPieza) y lo mismo para calcular cada rotación. Con la cota que dijimos antes cada una de estas operaciones cuesta O(k). Notemos que en el peor caso para cada pieza habría que chequear las cuatro rotaciones.

En el peor caso, las posibles posiciones de la pieza son todas las del tablero, es decir k. Se va a iterar en la cantidad de posibles posiciones.El costo de lo que está adentro del ciclo es O(k+ T(n-1)), siendo T(n-1) el costo de la siguiente llamada recursiva a la función.

Es decir que el costo de una llamada a la función con cantidad de piezas no nulas es de $4 * [k + k^2 + k * (k + T(n-1))]$, o lo que es lo mismo:
$4 * [k + k^2 + k^2 + k * T(n-1)]$

Pasando en limpio de manera más formal:
$T(0) = k$
$T(n) = 4 * [k + k^2 + k^2 + k * T(n-1)]$

Veamos más detenidamente $T(n)$:

$$T(n) = T(n) = 4*[k+k^2+k^2+k*T(n-1)] = 4*k+4*k2+4*k^2+4*k*T(n-1) =$$
$$= 4*k+4*k^2+4*k^2+4*k*[4*k+4*k^2+4*k^2+4*k*T(n-2)] =$$
$$= 4*k+4*k^2+4*k^2+4^2*k^2+4^2*k^3+4^2*k^3+4^2*k^2*T(n-2) =$$
$$= 4*k+4*k^2+4*k^2+4^2*k^2+4^2*k^3+4^2*k^3+4^2*k^2*[4*k+4*k^2+4*k^2+4*k*T(n-3)] =$$
$$= 4*k+4*k^2+4*k^2+4^2*k^2+4^2*k^3+4^2*k^3+4^3*k^3+4^3*k^4+4^3*k^4+4^3*k^3*T(n-3) =$$
$$= .... =$$
$$= (4*k)^n * T(0) + \sum i = 1^n (4*k)^i + 2*(4^i * k^{i+1}) =$$
$$= (4*k)^n * k + \sum i = 1^n (4*k)^i + 2*(4^i * k^{i+1})$$

Es decir que :

$T(0) = k$
$T(n) = 4^n * k^{n+1} + \sum i = 1^n (4*k)^i + 2*(4^i * k^{i+1})$

Conjeturamos que esto cuesta $O(4^n * k^{n+1})$. Veámoslo por inducción:

Queremos ver que existe un d real positivo y n_0 natural positivo tales que para todo $n \geq n_0 valeque T(n) \leq d * 4^n * k^{n+1}$.

Caso base: n=1

begincenter $T(1) = 4*k^2+4*k+2*4*k^2 \leq 4*k^2+4*k^2+2*4*k^2 \leq 4*(4*k^2)$
Es decir que para $n = 1$ considerando un $d \geq 4$ alcanza.
endcenter

Paso inductivo: Supongo que $T(n-1) \leq d*4^{n-1}*k^n$. Quiero ver que esto implica que $T(n) \leq d * 4^n * k^{n+1}$

$$T(n) = 4*k+4*k2+4*k^2+4*k*T(n-1) \leq 4*k+4*k2+4*k^2+4*k*4^{n-1}*k^n \leq$$
$$\leq 4*k+4*k2+4*k^2+4*k*4^{n-1}*k^n \leq 4*k+4*k2+4*k^2+4^n*k^{n+1} \leq$$
$$\leq 4^n*k^{n+1}+4^n*kn+1+4^n*k^{n+1}+4^n*k^{n+1} \leq 4*4^n*k^{n+1}$$

indent que es lo que queriamos ver.

Luego, *resolverJuego* cuesta $O(4^n * k^{n+1})$.

3.3. Resultados

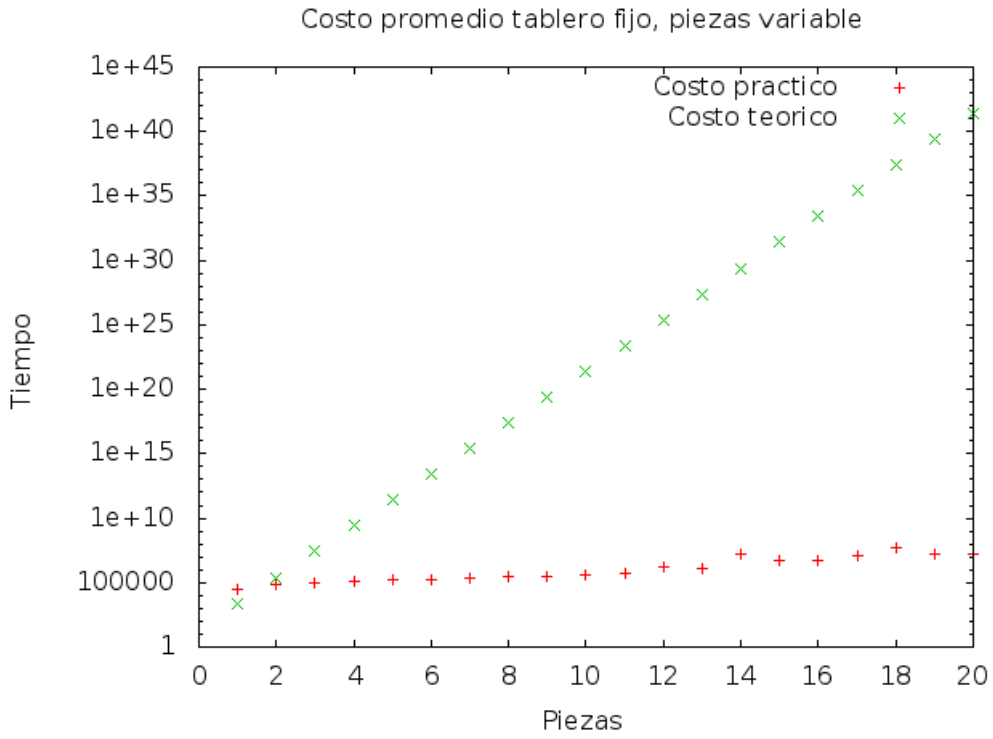

Figura 7: En escala logarítmica

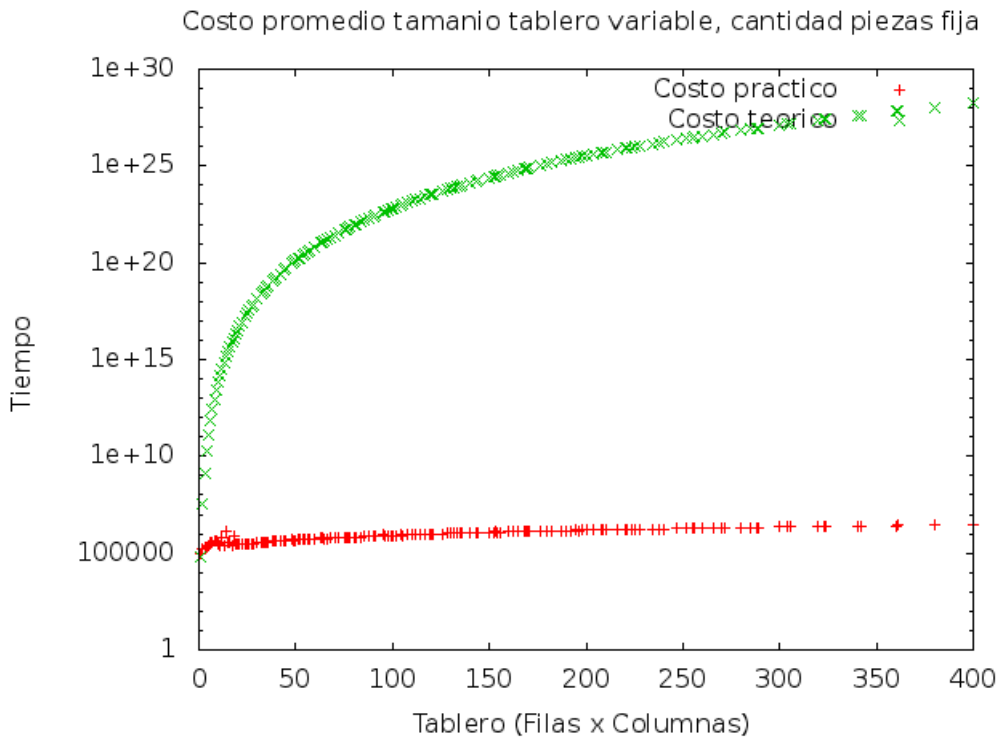

Figura 8: En escala logarítmica

3.4. Conclusiones

Considerando posibles mejoras sería que en vez de buscar *linealmente* cual es la menor cantidad de piezas necesarias, se podría utilizar una busqueda binaria. Con linealmente nos referimos en el sentido de que probamos con una pieza, luego dos, y así sucesivamente hasta probar con todas las piezas. En cambio, con búsqueda lineal, se prueba con la mitad de piezas, si es solución, entonces existe una solución menor o igual a esa. Por lo que se busca una nueva solución entre la 1era mitad en cantidad de piezas. Sino, entonces era necesario más piezas y se busca en la 2da mitad en cantidad de piezas.

Hay que tener en cuenta que al ser un problema que no se resuelve polinomialmente, intentar resolver con un valor grande en cantidad de piezas puede demorar demasiado en comparación si se hubiese hecho búsqueda lineal. Si se pudiera hacer un estudio sobre las piezas y el tablero y se conociera una estadística sobre la entrada del problema, se podría determinar casos donde conviene utlizar uno u otro.

Con este problema, pudimos notar la importancia de determinar buenas podas o puntos de corte para tratar de realizar un backtracking eficiente. De esta forma, se puede llegar a resolver problemas difíciles muchísimo más eficiente que si se hubiese realizado fuerza bruta.