Algoritmos y Estructuras de Datos III, TP2

Nicolás Chehebar, Matías Duran, Lucas Somacal

Índice

1. Problema 1

1.1. El Problema

1.1.1. Descripcion

Planteado de otra forma, el problema a resolver consiste en una situación en la que tenemos n trabajos $t_1, t_2, ..., t_n$ y dada cualquier división de los trabajos en dos secuencias $A = (t_{a_1}, t_{a_2}, ..., t_{a_{|A|}})$ y $B = (t_{b_1}, t_{b_2}, ..., t_{b_{|B|}})$ con $a_i < a_j \wedge b_i < b_j sii < j$ (cada secuencia representa los trabajos que realizo una máquina) tiene asociado un costo; donde este viene dado por la suma del costo de A y el de B. El costo de A es $\sum_{i=1}^{|A|} costo(t_{a_i}, t_{a_{i-1}})$ donde *costo* es una función que toma valores en $\mathbb{N}_0$ y $costo(t_i, t_j)$ esta definidio si $i > j$ con $i \in [1, 2, .., n] \wedge j \in [0, 1, .., n-1]$ y represnta el costo de poner el trabajo i sobre el j (el costo de poner sobre el trabajo t_0 es el de ponerlo sobre la máquina vacía y $a_0 = 0$). El costo de B se calcula análogamente.

El problema pide dados los trabajos y la funcion de costo, dar A o B que minimice el costo y decir cuanto es este costo (basta dar uno de los dos ya que el otro se deduce por ser el complemento -en el conjunto de trabajos-)

1.1.2. Ejemplos

- En el caso en que la entrada es
$$\begin{matrix} 4 & & & \\ 2 & & & \\ 300 & 3 & & \\ 300 & 3 & 3 & \\ 300 & 3 & 3 & 3 \end{matrix}$$
tenemos 4 trabajos que sacando el primero son excesivamente caros de poner por primera vez en una maquina, luego si ponemos todos en la misma, el costo sera $2+3+3+3 = 11$ y una máquina tendrá todos los trabajos (si todos no estan en la misma, en algun momento pagamos 300 y el costo ya seria mayor a 11).

- En el caso en que la entrada es
$$\begin{matrix} 4 & & & \\ 2 & & & \\ 4 & 1 & & \\ 300 & 3 & 300 & \\ 300 & 300 & 300 & 3 \end{matrix}$$
tenemos 4 trabajos, ponemos el primero en una maquina y nos da costo 2, si bien en el proximo paso lo mejor es poner el nuevo trabajo encima (si hicieramos un algoritmo goloso), en ese caso el siguiente trabajo costara 300 haciendo el total > 299, y si no hibieramos puesto el segundo encima, si bien costaba mas en ese paso, reducía el costo del próximo, dando un costo total de 12 estando los trabajos 1, 3 y 4 en una maquina.

1.2. El Algoritmo

1.2.1. La función de dinámica

Para resolver el problema, utilizaremos programación dinámica. La idea de esto se basa en que la solución de nuestro problema es calculable en base a la solucion de subproblemas (utilizamos optimalidad de subproblemas). Definimos así $f(q, h)$ como la función que asigna el mínimo costo posible para llegar al trabajo q-ésimo hecho (habiendo hecho de la 1 hasta la q inclusive) con el trabajo h como el último que se hizo en algunas de las máquinas. Tomamos como dominio de la f a los $q \in [1, 2, ..., t] \wedge h \in [0, 1, ..., q-1]$. donde $h = 0$ significa que hay una maquina vacía. Es clave notar que siempre que luego de que realizamos el trabajo q en una de las máquinas estará en el tope dicho trabajo, por ende basta definir que hay en la

otra. Definimos a continuación la función para los valores en el dominio ya mencionado:

$$f(q,h) = \begin{cases} costo(q,0) & si\ q = 1 \wedge h = 0 \\ f(q-1,h) + costo(q,q-1) & sih < q-1 \\ \min\limits_{0 \leq h \leq q-2} f(q-1,h) + precios[q][h] & \text{caso contrario (h=q-1)} \end{cases} \quad (1)$$

Esta función hace efectivamente lo que queremos:

- En el primer caso lo hace pues si $q = 1$ esto implica $h = 0$ por restricciones de dominio y es la minima cantidad dado que coloqué solo el primer trabajo, pues si o si el costo será el de colocar la primera sobre la maquina vacía, por ende será el mínimo.
- En el segundo caso también lo hace pues si esta un trabajo $h < q-1$ en una máquina es porque el ultimo trabajo colocado (el q) se colocó en la otra, por ende previo a finalizar el trabajo q, estaba en una máquina el $q-1$ y en otra el h. Más aún sabemos que el q lo colocamos sobre el $q-1$. Supongamos $f(q,h)$ el costo mínimo dado el trabajo q hecho y el trabajo h en alguna impresora (analogo para $f(q-1,h)$), si es $f(q,h) < f(q-1,h) + costo(q,q-1)$ luego es absurdo pues $f(q-1,h)$ no es el minimo, ya que hago la secuencia que da el mínimo en q trabajos hechos con h en una máquina sin el ultimo paso (resto su costo, o sea el de poner a q sobre $q-1$) y me queda que tengo una forma de tener $q-1$ trabajos hechos con h en una maquina con costo $f(q,h) - costo(q,q-1) < f(q-1,h)$ lo que es absurdo pues $f(q-1,h)$ era el minimo.
- En el tercer caso también sucede pues si está el trabajo $q-1$ en una máquina con la impresión q ya hecha, es porque la impresión q se colocó sobre alguna impresión h con $0 \leq h \leq q-2$. Dada dicho trabajo, de forma totalmente análoga al caso de arriba, debe ser el mínimo buscado con q trabajos hechos y el $q-1$ en una máquina $f(q,q-1) = f(q-1,h) + costo(q,h)$. Luego como no se cual trabajo de todos pudo haber sido, me quedo con el mínimo moviendo los h en el rango dado.

Así, definimos una función que resuelve el problema pedido si hayamos el $\min\limits_{0 \leq h \leq t-1} f(t,h)$ pues es el mínimo costo de realizar hasta el trabajo t (o sea todos) con el trabajo h en alguna máquina (me fijo todos los escenarios posibles como puede terminar la otra máquina, o sea todos los posibles h).

Así, podemos implementar la f dada, donde podemos ir recordando los valores que toma f y evitar calcularlos varias veces. Más aún podríamos mantener una lista (ordenada) de cuales son los elementos que hay en alguna máquina y cada vez que agregamos un trabajo, checkeamos si lo agregamos sobre el ultimo de la lista y en ese caso lo incluimos al final de esta (sino es porque fue a la otra máquina).

Lo que sucede es que tenemos varios subproblemas y en este caso siempre resolvemos todos, por lo que no parece tener una clara ventaja hacerlo top-down. Más aún, hacerlo bottom-up nos permitirá solo guardarnos los subproblemas relativos a tener hecho exactamente hasta el anterior trabajo (con $q-1$ y para todos los h, los menores no los utilizo en el calculo de $f(q,h)$), o sea nos reduce la complejidad espacial! Ya que incialmente debíamos guardar el valor de f para todo $q \in [1,2,...,t] \wedge h \in [0,1,...,q-1]$ lo que era $\mathcal{O}(n^2)$, y de esta forma solamente guardamos los valores para $q-1$ lo que es $\mathcal{O}(n)$.

Veamos todo esto en un pseudocódigo:

1.2.2. El Pseudocódigo

Cabe aclarar que en el pseudcódigo (como también en la implementación) numeramos los trabajos desde 0 a exceión de en *costos* (que es una matriz) en el segundo indice, los trabajos estan numerados

desde 1 (ya que el 0 se reserva para el costo de poner sobre la máquina vacía).

Algorithm 1: Devuelve el mínimo costo de entre todas las formas posibles de realizar todas las impresiones

```
1  Backtracking (confiables, actual, matriz, maximo);
   Input  : trabajos ∈ ℕ0; costo ∈ ℕ0^(trabajos×trabajos)
   Output: costo ∈ ℕ0,lista vector enteros
2  Inicializo en 0 actualcosto y anteriorcosto vectores de enteros (de tamaño trabajos);
3  Inicializo en vectores vacíos actuallista y anteriorlista vectores de vectores de enteros (de tamaño
     trabajos);
4  for q ∈ [0, 1, .., trabajos) do
5      for h ∈ [0, 1, .., q) do
6          actualcosto[h] = anteriorcosto[h] + costo[q][q];
7          actuallista[h] = anteriorlista[h];
8          if Estaba q − 1 en anteriorlista[h] then
9              Agrego q a anteriorlista[h];
10         end
11         actualcosto[q] = min(actualcosto[q], anteriorcosto[h] + costos[q][h]);
12         Recuerdo en elegido el h que consiguió el minimo;
13     end
14     actuallista[q] = anteriorlista[elegido];
15     if NO Estaba q − 1 en anteriorlista[elegido] then
16         Agrego q a anteriorlista[elegido];
17     end
18     anteriorcosto=actualcosto;
19     anteriorlista=actuallista;
20 end
21 costo = minimo de actualcosto (se alcanza en actualcosto[posicion]);
22 lista = actuallista[posicion];
23 return costo, lista;
```

En el pseudocódigo basicamente lo que hacemos es aplicar la f pero en orden, es importante esto ya que hay que tener cuidado en el orden en que resolvemos las dependencias (es porque estamos haciendo bottom-up). Es claro que cada fila, usa la fila anterior, o sea para calcular $f(q,h)$ para todo h uso todos los valores de $f(q-1,h)$ para todo h. Por esto es que ambos for se anidan de dicha manera. Al principio del for actualizamos el costo segun nos dice la f y la lista de los trabajos que hay en alguna máquina se modifica solo si era el de la máquina que tenía a $q-1$ ya que es a la que le agrego el trabajo q. Además, como voy a recorrer todos los h, voy actualizando el $actualcosto[q]$ si tengo un menor $anteriorcosto[h] + costos[q][h]$; una vez que iteré en todos los h calculé el minimo que es $f(q, q-1)$. Ahí salimos del primer for y como hacía con $actuallista[h]$ actualizo si corresponde la $actuallista[q]$. Finalmente, antes de pasar a la siguiente iteración pongo en $anteriorcosto$ el $actualcosto$ y en $anteriorlista$ la $actuallista$, ya que en la proxima iteración los actuales serán anteriores y sobre actual pisaré y guardaré nuevos resultados. Finalmente, se devuelve el mínimo buscado y su lista asociada (lo que nos pedían era $\min_{0 \le h \le t-1} f(t,h)$ que en nuestro caso es el minimo de $actualcosto$.

1.3. Complejidad

Cabe aclarar que para el análisis de complejidad tomaremos n como la cantidad de trabajos. Como pudimos ver en la explicación de la función de dinámica, tenemos n^2 subproblemas y cada uno se resuelve en $\mathcal{O}(1)$ salvo los subproblemas donde$h = q-1$ que toman $\mathcal{O}(n)$. Luego tengo $\mathcal{O}(n^2)$ subproblemas (son n^2 en total y le saco los que no se resuelven en $\mathcal{O}(1)$ que son n) resueltos en $\mathcal{O}(1)$ cada uno y $\mathcal{O}(n)$

subproblemas (hay uno por cada q) resueltos en $\mathcal{O}(n)$ cada uno. Luego se deduce que la complejidad sera $\mathcal{O}(n*n)+\mathcal{O}(n^2*1)=\mathcal{O}(n^2)$

Más aún esto se ratifica si miramos el pseudocódigo ya que realizamos todas operaciones que son $\mathcal{O}(n)$ u $\mathcal{O}(n)$ fuera del ciclo (inicialización o recorrido de vectores de tamaño a lo sumo n). Veamos que sucede dentro del ciclo: tenemos dos ciclos anidados que se ejecuta cada uno a lo sumo n veces, por ende todo se ejecuta a lo sumo n^2 veces y todo lo de adentro son operaciones $\mathcal{O}(1)$ (checkear si $q-1$ esta en $anteriorlista[h]$ es $\mathcal{O}(1)$ porque inserto siempre ordenado y si es un elemento, es el ultimo; lo mismo vale para checkear si $q-1$ esta en $anteriorlista[elegido]$).Luego salimos del segundo for (el anidado)y cabe aclarar que copiar el vector $actualcosto$ y $actuallista$ no es $\mathcal{O}(1)$ sino $\mathcal{O}(n)$, pero esta solo en uno de los ciclos, por lo que se repite n veces y aporta una complejidad de $n*\mathcal{O}(n)=\mathcal{O}(n^2)$. Por ende en el ciclo tenemos $n^2*\mathcal{O}(1)+\mathcal{O}(n^2)=\mathcal{O}(n^2)$ y sumado a lo que esta fuera del ciclo nos da $\mathcal{O}(n)+\mathcal{O}(n^2)=\mathcal{O}(n^2)$

1.4. Experimentación

1.4.1. Contexto

La experimentacion se realizó toda en la misma computadora, cuyo procesador era Intel(R) Atom(TM) CPU N2600 @ 1.60GHz, de 36 bits physical, 48 bits virtual, con una memoria RAM de 2048 MB. Para experimentar, se calculó el tiempo que tardaba el algoritmo sin considerar el tiempo de lectura y escritura ni el tiempo que llevaba armar la matriz (ya que se leía un dato, se escribía la matriz y luego se leia el siguiente). El tiempo se medía no como tiempo global sino como tiempo de proceso, calculando la cantidad de ticks del reloj (con el tipo clock_t de C++) y luego se dividìa el delta de ticks sobre CLOCKS_PER_SEC. En todos los experimentos el tiempo se mide en segundos.

1.4.2. Experimentos

Para empezar a experimentar, se corrió el programa con una serie de 2000 casos generados aleatoriamente donde aleatoriamente fue con una cantidad de trabajos aleatoria entre 1 y 10^3 con una distribución uniforme [1] en dicho intervalo. Para la matriz de costos, también se tomo para cada celda un costo aleatorio generado de la misma forma con un costo entre 1 y 10^6. A continuación graficamos estas instancias:

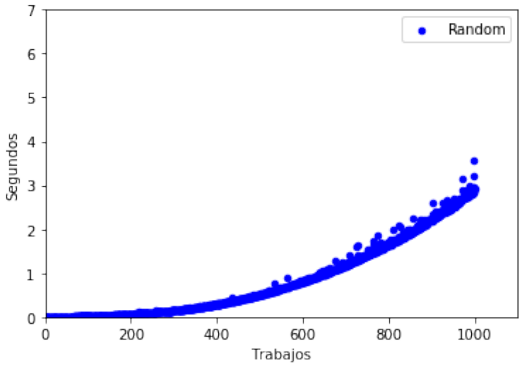

Figura 1: Gráfico de segundos de ejecución en función de cantidad de trabajos para instancias aleatorias.

Como podemos ver en la Figura 1, pareciera haber un gráfico semejante a una parábola lo que ratificaría la relación cuadrática que propusimos entre la cantidad de trabajos y la cantidad de operaciones realizadas. Sin embargo no nos da del todo información para aseverar eso este gráfico,ya que podría tratarse de alguna función con crecimiento similar. Es por esto que se realizó un gráfico de la relación entre tiempo de ejecución y $trabajos^2$ y si la relación es efectivamente cuadrática este gráfico debería ser constante. Como se puede ver en el gráfico de la figura 2 efectivamente se trata de una constante, lo que verifica nuestra hipótesis y complejidad teórica de la relación cuadrática de dependencia entre la cantidad de operaciones y de trabajos. Podemos observar que no pareciera haber practicamente dispersión, ni mejores ni peores casos, analizaremos esto luego.

[1] se utilizó la función rand() de librerías de C++ en el rango correspondiente, para mas detalle ver http://en.cppreference.com/w/cpp/numeric/random/rand

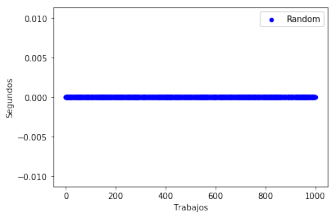

Figura 2: Gráfico de segundos de ejecución en función de cantidad de trabajos al cuadrado para instancias aleatorias.

Primero, debemos notar que segun el análisis realizado, en el tiempo de ejecución solo influye la cantidad de trabajos, más alla del costo que pueda tener cada trabajo ya que en lo unico que influye es en la suma (y como trabajamos con enteros acotados, no influye considerablemente en el tiempo de ejecución la suma). Por esto se ejecutaron las mismas 2000 instancias que se ejecutaron sumando una constante k que se movio entre 10^i con $i = 1, 2, 3, 4, 5, 6, 7, 8, 9$ a todo costo y no cambió considerablemente el tiempo de ejecución (notar que no cambia el resultado de cuales trabajos quedan en una máquina respecto de la ejecución respecto de no haber sumado la constante) en todos los i como vemos a continuación en los casos $i = 4$ e $i = 8$ en las figuras 3, 4 y 5

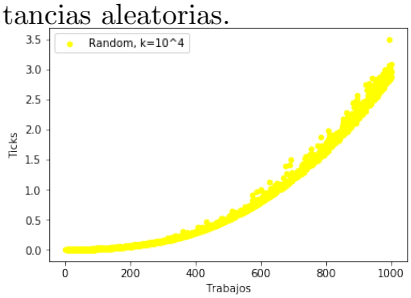

Figura 3: Gráfico de segundos de ejecución en función de cantidad de trabajos par instancias generadas aleatoriamente, sumando $k = 10^4$ en costos

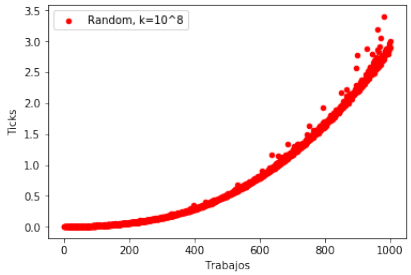

Figura 4: Figura 3: Gráfico de segundos de ejecución en función de cantidad de trabajos par instancias generadas aleatoriamente, sumando $k = 10^8$ en costos

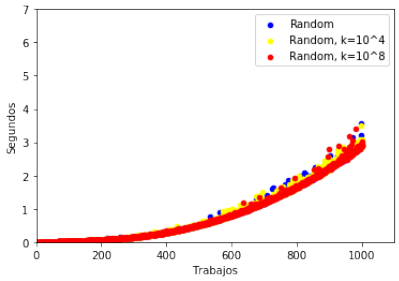

Figura 5: Figura 3: Gráfico de segundos de ejecución en función de cantidad de trabajos par instancias generadas aleatoriamente (en azul), sumando $k = 10^4$ (amarillo) y sumando $k = 10^8$ (rojo) en costos

Podemos ver que en todos los casos la dependencia sigue siendo, en rasgos generales la misma, cuadrática (se verifico haciendo el gráfico de $segundos/trabajos^2$ para cada i, los excluimos por una cuestión de espacio, pero todos resultaron constantes). Más aún al comparar instancias de diversos i podemos ver que tienen similar tiempo de ejecución lo que nos indica que (sumado a que tomamos costos aleatorios) no hay influencia de los costos en el tiempo de ejecución, lo que tiene sentido por lo que hace el algoritmo y la complejidad teorica calculada

Como decidimos implementar el algortimo de forma Bottom-Up siempre calculamos todos los subproblemas, esto es una ventaja en el sentido de que siempre todas las instancias de igual cantidad de trabajos tardan lo mismo, como se vio a lo largo de esta experimentacion, por lo que no hay mejores ni peores casos. Al ver la implementación y el pseudocódigo podemos ver que lo que realizamos depende exclusivamente de la cantidad de trabajos total (los costos solo cambian el resultado de cada cuenta, pero no la cantidad de operaciones ni su orden). También tomar esta decisión de implementar Bottom-Up nos permitió ahorrar en memoria ya que no requeríamos memorizar todos los subproblemas, sino que solo utilizabamos la información del subproblema anterior. La única desventaja es que a veces respecto de Top-Down, se calculan todos los subproblemas y no solo los necesarios. Pero si nos detenemos a ver la f que definimos al explicar el algoritmo (como ya también explicamos antes) siempre se van a calcular todos los subprobemas pues son todos necesarios, por ende esa tampoco es una ventaja del Top-Down en este caso. Todo esto se pudo ver experimentalmente ya que todas las instancias tuvieron un tiempo de ejecución muy similar y la dispersión fue practicamente nula.

1.5. Conclusiones

Concluimos entonces que la complejidad es de $\mathcal{O}(n^2)$ como se vio teóricamente y además se pudo verificar de forma experimental. Como se analizó al implementar bottom-up se resolvían todos los subproblemas siempre, por lo que (lo que también se vió experimentalmente) no había diferencia entre casos, no había

ni peores ni mejores casos, todas las instancias de igual cantidad de trabajos tomaban, practicamente, el mismo tiempo de ejecución. Más aún se vió también experimentalmente que (como se esperaba y se deducía del algoritmo) no había influencia alguna de los valores de los costos en el tiempo de ejecución.

2. Problema 2.1

3. Problema 2.2

4. Problema 3