

Algoritmos y Estructuras de Datos III, TP2

Nicolás Chehebar, Matías Duran, Lucas Somacal

Índice

1. Problema 1	2
1.1. El Problema	2
1.1.1. Descripción	2
1.1.2. Ejemplos	2
1.2. El Algoritmo	2
1.2.1. La función de dinámica	2
1.2.2. El Pseudocódigo	3
1.3. Complejidad	4
1.4. Experimentación	5
1.4.1. Contexto	5
1.4.2. Experimentos	5
1.5. Conclusiones	6
2. Problema 2	7
2.1. El Problema	7
2.1.1. Descripción	7
2.1.2. Ejemplos	7
2.2. Consultora 1	7
2.2.1. El algoritmo	7
2.2.2. Complejidad	8
2.3. Consultora 2	8
2.3.1. El algoritmo	8
2.3.2. Complejidad	9
2.4. Experimentación	9
2.4.1. Contexto	9
2.4.2. Experimentos	9
2.5. Conclusiones	9
3. Problema 3	9
3.1. El Problema	9
3.1.1. Descripción	9
3.1.2. Ejemplos	9
3.2. El Algoritmo	10
3.2.1. Resumen	10
3.2.2. El Pseudocódigo	11
3.3. Complejidad	11
3.4. Experimentación	12
3.4.1. Contexto	12
3.4.2. Experimentos	12
3.5. Conclusiones	14

1. Problema 1

1.1. El Problema

1.1.1. Descripción

Planteado de otra forma, el problema a resolver consiste en una situación en la que tenemos n trabajos t_1, t_2, \dots, t_n y dada cualquier división de los trabajos en dos secuencias $A = (t_{a_1}, t_{a_2}, \dots, t_{a_{|A|}})$ y $B = (t_{b_1}, t_{b_2}, \dots, t_{b_{|B|}})$ con $a_i < a_j \wedge b_i < b_j \implies i < j$ (cada secuencia representa los trabajos que realiza 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 $\sum_{i=1}^{|A|} \text{costo}(t_{a_i}, t_{a_{i-1}})$ donde costo es una función que toma valores en \mathbb{N}_0 y $\text{costo}(t_i, t_j)$ está definido si $i > j$ con $i \in [1, 2, \dots, n] \wedge j \in [0, 1, \dots, n-1]$ y representa 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 función 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

4
2
300 3
300 3 3
300 3 3 3

 tenemos 4 trabajos que sacando el primero son exce-

sivamente caros de poner por primera vez en una máquina, luego si ponemos todos en la misma, el costo será $2 + 3 + 3 + 3 = 11$ y una máquina tendrá todos los trabajos (si todos no están en la misma, en algún momento pagamos 300 y el costo ya sería mayor a 11).

- En el caso en que la entrada es

4
2
4 1
300 3 300
300 300 300 3

 tenemos 4 trabajos, ponemos el primero en una

máquina y nos da costo 2, si bien en el próximo paso lo mejor es poner el nuevo trabajo encima (si hicieramos un algoritmo goloso), en ese caso el siguiente trabajo costará 300 haciendo el total > 299 , y si no hubieramos puesto el segundo encima, si bien costaba más en ese paso, reducía el costo del próximo, dando un costo total de 12 estando los trabajos 1, 3 y 4 en una máquina.

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 solución 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 $q \in [1, 2, \dots, t] \wedge h \in [0, 1, \dots, q-1]$. donde $h = 0$ significa que hay una máquina 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:

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

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 mínima 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 (análogo para $f(q-1, h)$), si es $f(q, h) < f(q-1, h) + \text{costo}(q, q-1)$ luego es absurdo pues $f(q-1, h)$ no es el mínimo, 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 $f(q, h) - \text{costo}(q, q-1) < f(q-1, h)$ lo que es absurdo pues $f(q-1, h)$ era el mínimo.
- 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 $f(q, q-1) = f(q-1, h) + \text{costo}(q, h)$. 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 $\min_{0 \leq h \leq t-1} f(t, h)$ 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, chequeamos 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 inicialmente debíamos guardar el valor de f para todo $q \in [1, 2, \dots, t] \wedge h \in [0, 1, \dots, q-1]$ 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 pseudocódigo (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 Dinámica (trabajos, costo);
   Input : trabajos  $\in \mathbb{N}_0$ ; costo  $\in \mathbb{N}_0^{\text{trabajos} \times \text{trabajos}}$ 
   Output: costo  $\in \mathbb{N}_0$ , 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  $q \in [0, 1, \dots, \text{trabajos})$  do
5   for  $h \in [0, 1, \dots, q)$  do
6     actualcosto[ $h$ ] = anteriorcosto[ $h$ ] + costo[ $q$ ][ $h$ ];
7     actuallista[ $h$ ] = anteriorlista[ $h$ ];
8     if Estaba  $q - 1$  en anteriorlista[ $h$ ] then
9       | Agrego  $q$  a anteriorlista[ $h$ ];
10    end
11    actualcosto[ $q$ ] =  $\min(\text{actualcosto}[q], \text{anteriorcosto}[h] + \text{costo}[q][h])$ ;
12    Recuerdo en elegido el  $h$  que consiguió el mínimo;
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 = mínimo de actualcosto (se alcanza en actualcosto[posicion]);
22 lista = actuallista[posicion];
23 return costo, lista;

```

En el pseudocódigo básicamente 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 según 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] + *costo*[q][h]; una vez que iteré en todos los h calculé el mínimo 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 próxima 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 $\min_{0 \leq h \leq t-1} f(t, h)$ que en nuestro caso es el mínimo 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 leía 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 dividía 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 uniforme¹ 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:

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.

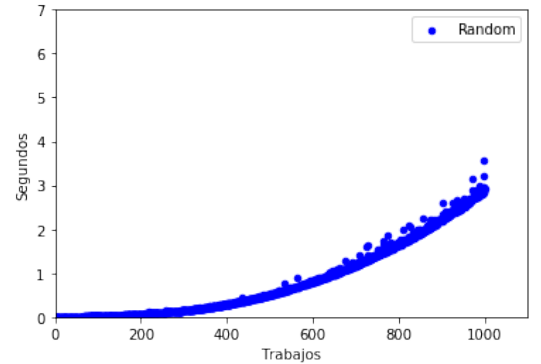


Figura 1: Gráfico de segundos de ejecución en función de cantidad de trabajos para instancias aleatorias.

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>

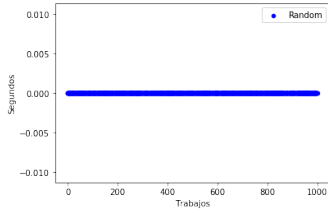


Figura 2: Gráfico de segundos de ejecución en función de cantidad de trabajos al cuadrado para instancias aleatorias.

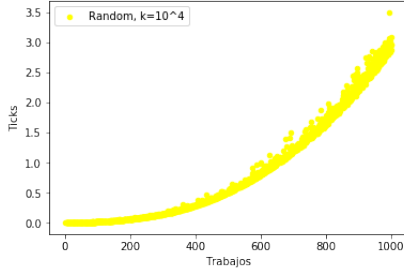


Figura 3: Gráfico de segundos de ejecución en función de cantidad de trabajos par instancias generadas aleatoriamente, sumando $k = 10^4$ en costos

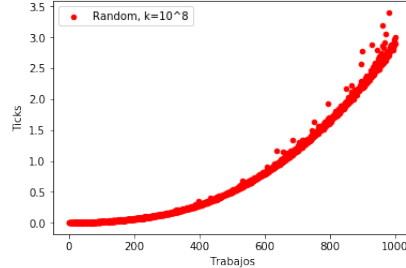


Figura 4: Gráfico de segundos de ejecución en función de cantidad de trabajos par instancias generadas aleatoriamente, sumando $k = 10^8$ en costos

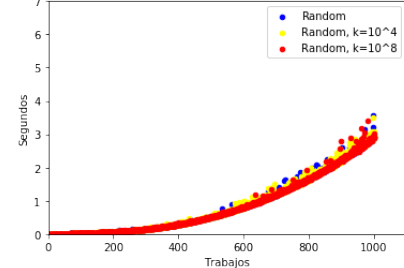


Figura 5: Gráfico de segundos de ejecución en función de cantidad de trabajos par instancias generadas aleatoriamente (en azul), sumando $k = 10^4$ (amarillo) y sumando $k = 10^8$ (rojo) en costos

Podemos ver que en todos los casos la dependencia sigue siendo, en rasgos generales la misma, cuadrática (se verificó haciendo el gráfico de $\text{segundos}/\text{trabajos}^2$ 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 teórica calculada.

Como decidimos implementar el algoritmo 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 experimentación, 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 utilizábamos 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 subproblemas 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 prácticamente 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 vio 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

2.1. El Problema

2.1.1. Descripción

Si nos abstraemos de los detalles del problema, este nos describe una situación en la cual tenemos un grafo G (no orientado) conexo con pesos no negativos. Lo que nos piden en la parte 1 del problema es encontrar un conjunto de aristas $E' \subseteq X(G)$ del grafo que cumpla que la suma de sus costos sea la mínima posible (minimizar $\sum_{e \in E'} \text{peso}(e)$ y que el subgrafo H con nodos $V(G)$ y aristas E' sea conexo. En la parte dos del problema nos piden, dado un E' que cumple lo antes descripto, elegir un nodo $v \in V(G)$ tal que si consideramos el subgrafo H sin pesos, minimice la máxima distancia de v a otro nodo (minimice $\max_{w \in V(H)} \text{distancia}(v, w)$).

2.1.2. Ejemplos

- Si consideramos K_n (el grafo completo de n vertices) con pesos constantes, todos 1 por ejemplo, la solución será cualquier conjunto de aristas que conecte todos los vertices (son al menos $n - 1$ aristas) y con exactamente $n - 1$ aristas se alcanza el minimo (pues cada arista es de peso positivo, si no tuviese la minima cantidad de aristas saco una y disminute el peso). Así, la solución tendra peso $(n - 1) * 1 = n - 1$ y podemos elegir tales aristas que cumplan que el subgrafo sea conexo (tomo la arista $(i, i + 1)$ con $i = 1, 2, \dots, n - 1$ donde los nodos estan numerados $1, \dots, n$). Esta sería entonces una solución posible
- Si consideramos C_n (el ciclo simple de n vertices) con pesos todos distintos positivos, la solución debe tener la minima cantidad de aristas posibles (pues cada arista es de peso positivo, si no tuviese la minima cantidad de aristas saco una y disminute el peso) y para que sea conexo estas son $n - 1$. Luego basta excluir solo una arista y como quiero minimizar el peso y saque cual saque queda conexo, saco la de mayor peso y ya (es unica la solución en este caso, pues son todos distintos y la arista de peso maximo es única). Las aristas buscadas serán todas menos la excluida y el peso, la suma de sus pesos.

2.2. Consultora 1

2.2.1. El algoritmo

Si nos detenemos a evaluar lo que pide la primer parte del problema, notamos que la solución debe permitir que sea conexo el grafo (debe tener $n - 1$ aristas al menos) y debe minimizar la cantidad de aristas (pues cada arista es de peso positivo, si no tuviese la minima cantidad de aristas saco una y disminute el peso), luego debe tener exactamente $n - 1$ aristas. Y además debe ser conexo!, luego se trata de un arbol, y como debe tener como nodos a $V(G)$ es un arbol generador. Pero buscamos la solución de peso minimo (o una de ellas), por ende la solución es un AGM (arbol generador minimo).

Para esto utilizamos el algoritmo de Prim (no pondremos su pseudocódigo por ser un algoritmo ya visto en clase y muy conocido, igual se puede ver el pseudocódigo en el problema 3, es cuestión de cambiarle las unicas dos modificaciones que estan claramente marcadas y comentadas). Tomamos la opcion de Prim en la que se utiliza un vector para implementar la cola de prioridad que tiene las distancias al AGM de los nodos no incluidos. **CHECKEA ESTO, NO SE BIEN QUE USAS....** Un breve resumen y descripcion de lo que hace es que itera n veces y a cada paso agrega un nodo al AGM que tenemos hasta ahora, utilizando la

arista de menor peso. Utilizamos como representación del grafo **ACA PONER BIEN LO QUE USES MATO, ACLARA BIEN!!!!**

2.2.2. Complejidad

Como bien sabemos, la complejidad del algoritmo de Prim puede ser o bien $\mathcal{O}(n^2)$ si se utiliza un vector para implementar la cola de prioridad que tiene las distancias al AGM de los nodos no incluidos (tomar el mínimo es $\mathcal{O}(n)$, pero actualizar una distancia es $\mathcal{O}(1)$) o bien $\mathcal{O}((m+n)\log(n))$ si se utiliza un heap (tomar el mínimo y actualizar son ambos $\mathcal{O}(\log(n))$). Ambas cumplen la complejidad pedida, pero en nuestro caso lo implementamos de la primera forma, por lo que la complejidad es $\mathcal{O}(n^2)$ que cumple lo pedido.

2.3. Consultora 2

2.3.1. El algoritmo

El algoritmo en si es muy simple, la idea es encontrar el camino máximo del árbol que nos devuelve el algoritmo de la consultora 1 y tomar el nodo que está en la mitad del camino (o alguno de los dos si tiene una cantidad par de nodos el camino). Y para tomar el camino mas largo, lanzamos BFS desde un nodo cualquiera v para medir los caminos minimos (notar que los caminos son unicos, pues es un árbol) a todos los demas nodos (BFS es aplicable pues todas las aristas tienen el mismo peso en este caso) y sea w el que esta mas lejos. Luego lanzamos BFS desde w y sea z el que este a mayor distancia de w . Luego el unico camino entre z y w (unico pues es un árbol) es el camino de máxima longitud que buscamos. **MATO USASTE BFS NO??? CAMBIAR SINO DONDE CORRESPONDA SI USASTE DFS**

Lo que es quizás mas complejo es entender por qué efectivamente esto funciona. Lo que nos piden es dado el árbol que devuelve la consultora 1, encontrar un nodo tal que si lo elegimos como raíz, la altura del árbol sea minima (i.e, minimizar la máxima de las distancias). Veamos primero que esta distancia tiene que ser $\geq x/2$ donde $x = longituddelcaminosimplemaximo$. Supongamos que no, luego es $< x/2$ y por ende el camino entre dos nodos siempre sera $< x$ ya que un camino posible entre dos nodos ayb (no necesariamente simple, por ende de mayor longitud que el simple) es ir desde a hasta el nodo que elegimos como raíz y luego ir desde la raíz al b , como ambos caminos son de longitud $< x/2$ (es ir desde un nodo a la raíz y el árbol tiene altura $< x/2$), se deduce que el camino de unir ambos tiene longitud $< x$. Luego, finalmente todo camino entre un par de nodos tiene longitud $< x$, luego x no era camino simple de longitud maxima (notar que el camino entre dos nodos es unico), pues todo camino simple tiene longitud menor. Hemos visto que la distancia debe ser $\geq x/2$, por ende demostramos que encontrar el camino maximo y tomar como raíz un nodo de la mitad del camino, minimiza la altura.

Falta ver entonces que usar dos veces BFS como dijimos nos da efectivamente los dos nodos que dan el camino maximo. Sean a, b los dos nodos que son extremos del camino máximo. Y sea v el nodo desde el que inicialmente lanzamos BFS y w sobre el segundo que lanzamos BFS (el mas lejano de v). Si quitamos v del árbol, este se nos divide en c componentes conexas. Si ayb pertenecen a distintas componentes conexas, luego el camino (es un árbol, luego es unico) que los une pasa por v . Supongamos sin perdida de generalidad que w no esta en la misma componente conexa que a (si no lo tomamos respecto a b , siempre hay uno con el que no esta en la misma componente conexa, pues no puede estar en dos componentes conexas a la vez). Luego, si consideramos el camino desde w a v es de longitud mayor (o igual quizas) que el camino de a a v (pues w es el mas lejano de v). Luego el camino de w a v unido con el de v a b tiene longitud mayor (o igual), lo que nos dice que necesariamente uno de esos nodos debe ser w . Luego tomamos el más lejano a w lanzando nuevamente BFS y obtenemos el camino maximo. Ya lo probamos si a y b estan en diferentes componentes conexas, veamos que sucede si a y b quedan en la misma componente conexa, pero en ese caso, repetimos el mismo argumento en la componente conexa desde el único elemento que estaba conectado con v como la raíz, luego w sigue siendo el más lejano a este (si alguno w' fuese el nuevo mas lejano, estaría mas lejos que w de v pues solo sumo uno mas en ambas distancias para llegar desde la

nueva raíz a v y debo pasar si o si por ella pues es lo que une a la componente conexa con v , absurdo). Iteramos así y a cada paso reducimos en uno la altura del árbol que nos va quedando, si en algún momento a y b quedan en componentes conexas distintas, ya esta por lo que porbamos antes, si no repetimos el argumento, hasta que en un momento (cuando la altura del árbol sea 2) al sacar un nodo nos quedan componentes conexas triviales y forzadamente a y b deben estar en componentes conexas distintas y vale lo que dijimos.

Luego hemos probado que hacer BFS dos veces de esta forma nos da el camino simple máximo, en realidad nos da sus extremos, pero como el camino es unico, se deduce, usando **DECI QUE USAS ACA PARA AGARRAR EL CAMINO EN SI SABIENDO YA LOS VERTICES**. Hemos probado ademas que un nodo de la mitad del camino simple máximo realiza el mínimo buscado (probamos que ninguna otra distancia menor funciona, por ende este es el mínimo). Luego, demostramos que nuestro algoritmo es correcto y hace lo que efectivamente queremos

2.3.2. Complejidad

Ejecutamos dos veces BFS, que como bien sabemos es $\mathcal{O}(n + m)$ (ejecutarlo dos veces lo sigue siendo), pero como estamos en un árbol, $m = n - 1$ luego $\mathcal{O}(n + m) = \mathcal{O}(n + n - 1) = \mathcal{O}(n)$. Luego, una vez que tenemos los extremos del camino máximo, recorreremos el grafo buscando las aristas que nos llevan entre ambos extremos, que lo hacemos en $\mathcal{O}(n)$ nuevamente **DECI QUE USAS ACA PARA AGARRAR EL CAMINO EN SI SABIENDO YA LOS VERTICES, ACLARA BIEN QUE HACES Y PORQUE ES $\mathcal{O}(n)$** . Finalmente tomamos el nodo de la mitad de la lista de nodos que nos dio este recorrido. Como solo hicimos tres cosas que son $\mathcal{O}(n)$, la complejidad total es esa y cumple lo pedido.

2.4. Experimentación

2.4.1. Contexto

2.4.2. Experimentos

2.5. Conclusiones

3. Problema 3

3.1. El Problema

3.1.1. Descripcion

Planteado de otra forma, la situación que tenemos es un grafo (no orientado) con pesos positivos en las aristas G en el cual tenemos una particion de $V(G) = F, C$ dada (con $|C| \geq |F|$) y queremos hallar un subconjunto de aristas $E' \subseteq X(G)$ que tenga peso minimo (o sea minimizar $\sum_{e \in E'} peso(e)$) y que cumpla que para todo nodo en C existe un camino a algun nodo en F (o sea, para toda componente conexa W del grafo $H = (V(G), E')$, $\exists v \in F$). Nos piden hallar ese costo minimo (i.e $\sum_{e \in E'} peso(e)$) cuantas y qué aristas lo logran.

3.1.2. Ejemplos

- Consideremos C_n (supongamos n par) con pesos asociados todos iguales y bipartito con F y C los elementos de la partición (o sea, en el ciclo, si lo recorremos en algun sentido, hay un nodo de F , luego uno de C , luego uno de F y así sucesivamente). Es claro que debemos minimizar la cantidad de aristas (todas pesan lo mismo y tienen costo positivo) y como tenemos $|C| = n/2$ necesitaremos al menos $n/2$ aristas y consideramos alguna de las que la unen con alguno de sus vecinos para cada elemento de C . Así tenemos nuestro conjunto de aristas que minimizan la suma (será $k * (n/2)$ si k

es el peso de cada arista), de hecho es claro en este ejemplo que el conjunto de aristas que minimiza la suma no es único (de hecho hay $2^{n/2}$ -pues para cada elemento de C elijo una de las dos aristas que inciden en el)

- Consideremos ahora un grafo compuesto de q componentes conexas donde cada una es un C_n como mostramos en el ejemplo anterior (con n par para toda componente conexa y con un elemento de C y uno de F alternadamente y pesos iguales k). Para minimizar la cantidad de aristas, debemos resolver el problema en cada una de las componentes conexas, pues la única forma de llegar a un elemento de una componente conexa es desde alguna fábrica que este en esa componente. Luego, como vimos cada componente se resuelve con $k * (n/2)$ de peso total, por ende la solución total tendrá $q * k * (n/2)$

3.2. El Algoritmo

3.2.1. Resumen

Lo que tenemos que hacer es algo bastante parecido a encontrar un AGM, pero como hemos visto incluso en los ejemplos, la solución no tiene por qué ser un árbol. Más aún ni siquiera tiene que generar (puede que haya un nodo que no sea alcanzable, en ese caso sería uno de F -por ejemplo uno que tiene un costo altísimo cada arista que lo une con cualquier otro y siempre es más barato llegar a sus vecinos desde otro elemento de F -). Pero si nos detenemos a pensar, no tiene sentido que haya un ciclo, ya que sacamos una arista y (como todas tienen peso positivo) disminuye el peso. Luego nuestro grafo solución no es un AGM, pero si es un bosque que tenga a todos los elementos de C y sea de peso mínimo. Y un bosque es un conjunto de árboles, queremos hacer prácticamente lo mismo que en un AGM, pero sin mantener necesariamente la conexión en el grafo que vamos generando (al que le agregamos un nodo y una arista en cada iteración). De hecho, sabemos que este grafo tendrá exactamente $|C|$ aristas, una por cada iteración ya que en cada iteración agregaremos al cliente "más cercano". Así surge la idea de lo que realizamos: hacer Prim pero comenzando en vez de con un nodo, con todos los nodos de F .

3.2.2. El Pseudocódigo

Como se trata efectivamente de una variación del algoritmo de Prim, incluiremos un pseudocódigo

Algorithm 2: Devuelve un conjunto de aristas que conectan a todo elemento de C con alguno de F con menor costo y su costo

```

1 PrimModificado ( $G$ );
   Input :  $GrafoG$ ,  $F \subseteq V(G)$ 
   Output:  $costo \in \mathbb{N}_0$ ,  $lista$  vector de aristas
2 for  $v$  en  $V(G)$  do
3   distancia[u] =  $\infty$ ;
4   padre[u] = NULL;
5   Añadir a la cola ( $u$ , distancia[u]);
6 end
7 for  $f$  en  $F$  do
8   distancia[f]=0;                                ▷ Cambio respecto de Prim
9 end
10 while NO esta vacia la cola do
11   for  $v$  adyacente a u do
12      $u = \text{extraer el de menor distancia de la cola que } \notin F$ ;          ▷ Cambio respecto de Prim
13     if  $v \in \text{cola} \wedge \text{distancia}[v] > \text{peso}(u, v)$  then
14       padre[v] =  $u$ ;
15       distancia[v] = peso( $u, v$ );
16       Actualizar la cola ( $v$ , distancia[v]);
17     end
18   end
19 end

```

Como se ve en el pseudocódigo, para resolver el problema usamos el mismo algoritmo que usa Prim, solo que en vez de empezar con algun nodo (cualquiera) marcado como el primer elemento del AGM, empezaremos con todos los elementos de F marcados. Además, cuando tomemos el nodo de menor distancia, tomamos el de menor distancia y que $\in C$. Así en cada iteración incluimos al nodo de C que esta a menor distancia del grafo hasta entonces generado, cada vez aumentamos el grafo inicial en un nodo hasta que no queden mas elementos en C (a diferencia de prim en que este grafo siempre era un arbol), y en cada paso siempre incluimos al que suma menor costo (y le actualizamos la distancia a todos sus vecinos). Así, cuando no queden más elementos de C por incluir, tendremos el conjunto de aristas que buscamos y serán las de peso mínimo (el argumento es exactamente el mismo que el de correctitud de Prim, siempre a cada paso agregamos el más cercano - si hubiera una arista e que convenía ser incluida en vez de otra f porque disminuiría el peso, no habría elegido a f en ningun momento pues siempre habría algun elemento a incluir con un costo menor al de f -).

3.3. Complejidad

Como ya analizamos en el problema 2, la complejidad de prim, por como lo implementamos, es $\mathcal{O}(n^2)$ y por ende ahora será $\mathcal{O}((|F| + |C|)^2)$ pero si recordamos, el enunciado nos asegura que $|C| \geq |F|$, luego $(|F| + |C|) \leq 2 * |C| = \mathcal{O}(|C|)$ y por lo tanto, $\mathcal{O}((|F| + |C|)^2) \leq \mathcal{O}(|C|^2)$. Mostramos así que el algoritmo cumple la complejidad propuesta. Es claro que nuestra variación no afecta en absoluto la complejidad de prim, pues como implementamos la cola como un arreglo y buscar el minimo nos tomaba $\mathcal{O}(n)$, encontrar el minimo dentro de los que pertenecen a C sigue siendo $\mathcal{O}(n)$ (donde n es el tamaño de la cola). Agregamos si un ciclo que inicializa todas las distancias de los elementos que estan en F , eso es $|F|$ operaciones $\mathcal{O}(1)$, lo que es $\mathcal{O}(|F|) \leq \mathcal{O}(|C|)$ y por ende no suma complejidad. Así, hemos probado que la modificación del algoritmo de Prim, mantiene la misma complejidad teórica, por ende hemos probado que nuestro algoritmo

es complejidad $\mathcal{O}(|C|^2)$.

3.4. Experimentación

3.4.1. Contexto

La experimentación 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 leía 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++). En todos los experimentos se medía en Ticks el tiempo de ejecución.

3.4.2. Experimentos

Primero, se generó una serie de casos aleatorios, generados de la misma forma que en el Problema 2.

La única diferencia radicó en que en vez de a partir del segundo nodo conectarlo con alguno de los anteriores para asegurar conexidad, esto se hizo a partir del nodo $F + 1$. Se corrieron casos con $C + F$ entre 1 y 60 y para cada uno de ellos, se movió el F entre 1 y $(C + F - 1)/2$ (para respetar que siempre $C > F$) y para cada uno de esos valores de $C + F$ y F se ejecutaron 100 casos aleatorios (en donde todo se realizó como se describió en el problema 2).

Como vemos en los gráficos, parece haber un crecimiento del tiempo de ejecución cuando crece la cantidad de clientes. Para verificar que este crecimiento hace que efectivamente lo estemos resolviendo en $\mathcal{O}(|C|^2)$, hicimos un gráfico de Ticks en función de clientes al cuadrado, donde se puede ver (salvo para pocos clientes) que el gráfico es acotado por una constante (ver gráfico ampliado). En los primeros casos de clientes no sucede ya que al ser pequeña la cantidad de clientes, toma mucha más importancia el término constante. Más aún, la notación \mathcal{O} solo nos habla de la complejidad asintótica, por lo que esto tiene sentido.

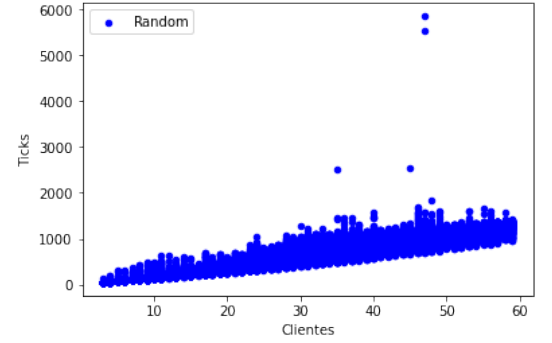


Figura 3: Gráfico de segundos de ejecución en función de cantidad de clientes para instancias aleatorias.

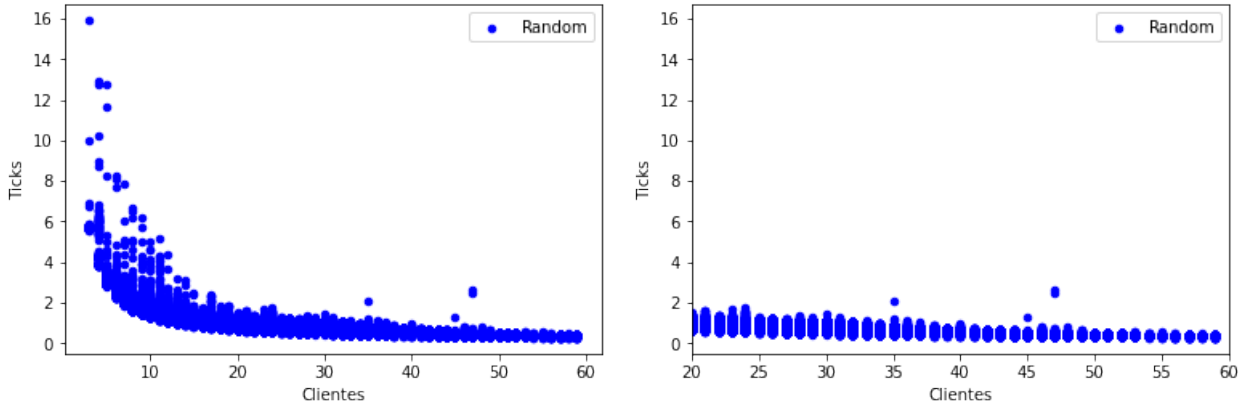


Figura ??: Gráfico de segundos de ejecución en función de cantidad de clientes al cuadrado para instancias aleatorias.

Figura ??: Gráfico de segundos de ejecución en función de cantidad de clientes al cuadrado para instancias aleatorias, con el eje x acotado entre 20 y 60.

Como ya hemos analizado en el problema 2 y se ve en el pseudocódigo, no hay influencia alguna del peso de las aristas, o sea el costo de reparar cada ruta. Por ende esta variable siempre se toma aleatoria, teniendo la certeza de que no influiría en las otras. Más aún, no debería haber dependencia con la cantidad de fábricas por como hemos visto, porque simplemente "pintamos" todas al principio de Prim y luego realizamos normalmente el algoritmo de Prim (salvo elegir el mínimo entre los clientes e incluir a cada paso el cliente mas cercano). Pero si recordamos el enunciado y la condición de que $|F| < |C|$, notamos que una vez elegida la cantidad de fábricas, sabemos que los clientes son más seguros y eso sí afectaría el tiempo de ejecución directamente. Por esto realizamos un gráfico para comprobarlo:

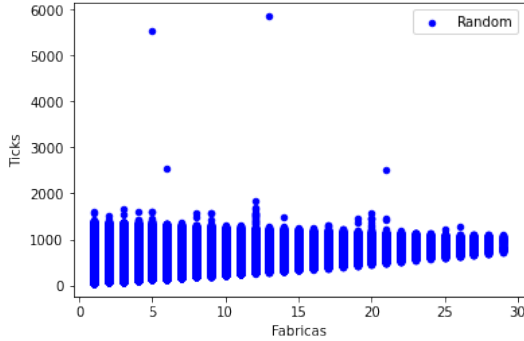


Figura 4: Gráfico de segundos de ejecución en función de cantidad de fábricas para instancias aleatorias.

Si vemos el gráfico, efectivamente hay correlación entre la cantidad de fábricas y el tiempo de ejecución. Vemos que cada vez la dispersión es menor cuando aumenta la cantidad de fábricas, lo que tiene sentido ya que como experimentamos con grafos de tamaño hasta 60 ($|F| + |C| = 60$) cuando fijamos $|F|$ en un número grande, reducimos considerablemente el rango de valores que puede tomar $|C|$ y por ende el rango de valores que toma el tiempo de ejecución (por ejemplo, cuando $|F|$ es 30, obligatoriamente $|C|$ debe serlo). Por esto, cuando $|F|$ crece, el tiempo de ejecución del caso más rápido crece ya que se da con el menor $|C|$ posible y al crecer $|F|$ lo hace $|C|$ pues $|F| < |C|$. Pero como vemos, el tiempo de ejecución de los casos mas lentos, se mantiene siempre igual, ya que se da con el caso de máximo $|C|$ posible y para cualquier $|F|$ dicho caso es posible. Basicamente pudimos notar que la cantidad de fábricas si influye en la cota inferior del tiempo de ejecución,

pero no en la superior ya que esta estará dada por la cantidad de clientes.

Queda analizar la dependencia del tiempo de ejecución en función de la variable R .

Si bien sabemos que la complejidad del algoritmo es $\mathcal{O}(|C|^2)$, pero si recordamos la complejidad teórica cuando calculamos la del algoritmo de Prim, en un momento recorremos todos los vecinos de un nodo, y como repetimos esto para todos los nodos que son clientes, en total lo hacemos $X(G)$ veces que en este caso es R . Luego, como $|V(G)|^2 \geq |X(G)|$ para todo grafo G , acotamos el R por $|C|^2$, pero influye en el tiempo de ejecución este R , lo que nos lleva a pensar que dentro de instancias de igual tamaño, las que tengan una mayor cantidad de aristas tendrán un mayor tiempo de ejecución y las que tengan menos aristas un menor tiempo de ejecución.

SI ESTO NO SUCEDIO ASI EFECTIVAMENTE DE + ARISTAS + TIEMPO Y - ARISTAS - TIEMPO CAMBIARLO Para ver esto, realizamos un gráfico de tiempo de ejecución en función de R . Efectivamente en el gráfico parecía verse una correlación, lo que verificamos utilizando el índice de pearson, como se puede ver en la siguiente figura:

Efectivamente, hay correlación (ya que el p-value es 0) y como el índice de pearson es positivo, este nos indica que al crecer la cantidad de rutas, crece el tiempo de ejecución. Pero también podemos ver que pareciera suceder lo mismo que con las fábricas ya que cuando crece la cantidad de rutas, el rango de valores que toma el tiempo de ejecución decrece.

Trataremos de entender si esto es porque al crecer R crece $|F| + |C|$, lo que fuerza a que crezca $|C|$ (puesto que $R \leq (|C| + |F|)(|C| + |F| - 1)/2$ ya que la máxima cantidad de aristas -rutas- se da en el

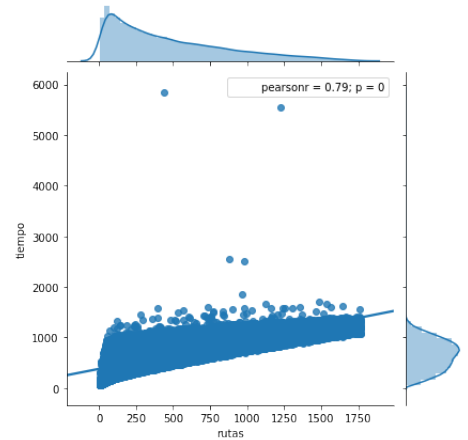


Figura 5: Gráfico de segundos de ejecución en función de cantidad de rutas para instancias aleatorias con el índice de pearson.

completo $K_{|C|+|F|}$, que es el que cumple la igualdad). Trataremos de buscar como es esta relación entre R y el tiempo de ejecución, como también mejores y peores casos:

SOMA ACA EMPEZAS VOS, TODO TUYO, MEJORES Y PEORES CASOS

Salvo una pequeña modificación en el código, sabemos que es un problema muy parecido al 2 (más aún porque ambos se implementaron con Prim y con un arreglo como estructura para la cola de prioridad).

3.5. Conclusiones

ACORDARSE DE AL FINAL DEL INFORME, CHECKEAR TODAS LAS NUMERACIONES DE LAS FIGURAS Y CADA VEZ QUE MENCIONEMOS UNA, LE PONGAMOS EL NUMERO, ASI SE ENTIENDE DE QUE HABLAMOS. NO BORRAR ESTO HASTA EL FINAL DEL INFORME