DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP2 de AED3

4 de mayo de 2024 — Algoritmos y Estructuras de datos III

Estudiante	LU	Correo electrónico
Lucas Mas Roca	122/20	lmasroca@gmail.com
Luciana Skakovsky	131/21	lucianaskako@gmail.com
Mateo Cantagallo	143/21	mateocantagallo@gmail.com
Alan Yacar	174/21	alanroyyacar@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introduccion

En este informe se encontrarán las explicaciones y demostraciones de correctitud y complejidad de los algoritmos propuestos para la resolución de los problemas planteados. Se describirá primero una breve explicación del problema (en muchos casos de forma más abstracta) y luego se desarrollara la solución pensada.

2. Problema 1: DFS

2.1. Problema

Este primer problema tenemos un laberinto que tiene habitaciones y pasillos entre ellas. Estos pasillos se pueden recorrer en ambas direcciones y se quiere poder determinar dadas 2 habitaciones si existe un único camino simple entre ellas.

2.2. Solución

Podemos ver fácilmente que este problema se puede analizar como un grafo no dirigido siendo los nodos las habitaciones y los pasillos las aristas entre ellas. Para empezar a resolver este problema tenemos que considerar primero que nada la complejidad pedida, ya que esta requiere que las consultas de cada par de habitaciones se respondan en tiempo O(1). Para poder conseguir esto se debe realizar un precomputo de la solución.

Para que un camino simple P entre 2 nodos sea único hace falta que todas la aristas del camino sean puentes. De lo contrario, es fácil ver que el camino no es único ya que, se podrían sacar del grafo las aristas de P que no son puentes y como no cambia la cantidad de componentes conexas seguiría existiendo un camino entre las 2 habitaciones distinto de P. Debido a esto podemos reducir nuestro problema a determinar si existe este camino P.

Gracias a las clases prácticas de la materia contamos con un algoritmo con complejidad lineal para buscar puentes. Una vez que tenemos el conjunto de aristas puente necesitamos poder determinar para que pares de habitaciones hay un camino que pase solo por esas aristas. Como no podemos ir probando par a par (por la complejidad pedida), corremos un DFS en el grafo que contiene solo las aristas puente y vemos para cada habitación a que árbol del bosque pertenecen, si 2 habitaciones pertenecen al mismo árbol entonces existe P.

2.3. Complejidad

Con respecto a la complejidad, creamos primero el grafo del laberinto representado como una lista de adyacencias esto toma tiempo $O(R + C)$, luego corremos el algoritmo visto en clase para buscar puentes ($O(R + C)$) [1]. Una vez que tenemos los puentes, rearmar el grafo con solo estas aristas y correr DFS desde este nuevo grafo es también $O(R+C)$. Para identificar al bosque resultante usamos un vector de tamaño R donde guardamos para cada habitación a que árbol pertenecen ($O(R)$).
Finalmente, con todo este precomputo al momento de consultar un par de habitaciones simplemente indexamos en nuestro vector en las posiciones correspondientes y las comparamos ($O(1)$).

[1] DFS y detección de puentes de la clase práctica de este cuatrimestre (2C2022)

2.4. Demostración

Lema: Dado un grafo G conexo (si no es conexo vale para cada componente conexa) y un camino simple P entre 2 vértices s, t.
P es único $\iff$ todas las aristas $e \in P$ son puentes.
$\implies$ (Contrarreciproco) Supongo que existe una arista $e \in P$ que no es puente. Tomo el grafo $G_2 = G - e$. Por definición de puente se que este nuevo grafo G_2 sigue siendo conexo por lo tanto existe un camino P_2 en G_2 que no pasa por e (porque la saque). Entonces P y P_2 pertenecen a mi grafo original por lo tanto P no es único.
$\impliedby$ (Absurdo) Supongo que existen 2 caminos distintos P y P_2 que unen s y t. Llamo e a alguna arista $e \in P$ y $e \notin P_2$ que se que existe porque $P \neq P_2$. Tomo el grafo $G_2 = G - e$. Como sigo teniendo el otro camino veo que no aumenta la cantidad de componentes conexas en G_2, porque puedo reemplazar todo lo que se conectaba con e con $P + P_2 - e$ por lo tanto e no era una arista puente.
Una vez que demostramos este lema es fácil ver que el algoritmo propuesto resuelve el problema, ya que por lo visto en clase luego de correr DFS en el grafo nuevo, 2 nodos pertenecen a un mismo árbol si y solo si existe un camino entre ellos. Por como definimos este nuevo grafo, todas sus aristas son puentes del original, entonces si ambos pertenecen a un mismo árbol de DFS, existe un camino entre ellos compuesto solo de puentes, lo cual significa que el camino es único por el lema demostrado arriba.

3. Problema 2: AGM

3.1. Problema

Este segundo problema se reduce a encontrar componentes candidatas en un grafo pesado. Una componente candidata es un conjunto de vértices que se conectan entre si y que cumplen que todas las aristas internas de la componente tienen peso mayor a las que salen de ella. Se pide devolver la sumatoria de los tamaños de las componentes candidatas.

3.2. Solución

Nuestro algoritmo se basa en una modificación del algoritmo de Kruskal visto en clase usado para conseguir AGMs (ya sea mínimo o máximo). Gracias a la ayuda presentada, solamente tenemos que checkear si una componente es candidata cuando se forma como resultado de uno de los pasos de Kruskal ordenando las aristas de forma descendiente. La forma que tenemos de checkear esto es la siguiente: cada vez que Kruskal agrega una arista que une dos componentes conexas, vemos si esa nueva componente conexa es candidata. Como lo que queremos ver es que todas las aristas internas sean mayores a todas las del borde en la nueva componente solo hace falta comprobar que la mínima interna sea mayor a la máxima del borde. Para poder hacer esto hace falta mantener para cada componente la mínima arista interna y la máxima externa, pero también conviene tener el valor de la arista de mínimo y máximo peso entre cada par de componentes, si no hay una arista que las conecta de forma directa los valores son infinito y -1 respectivamente. Esta información conviene tenerla, ya que facilita el cálculo de máximos y mínimos al conectar una arista.

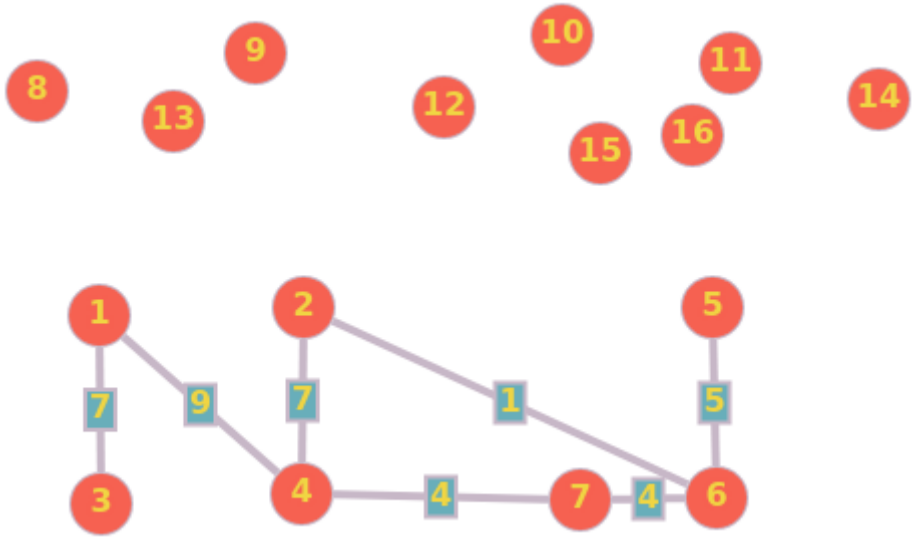

Figura 1: Grafo ejemplo (los vértices sueltos son para mostrar que los vértices 1 a 7 son solo una parte del grafo, podría haber conexiones entre los demás vértices pero no son relevantes para el ejemplo)

Por ejemplo si tenemos las componentes 1,2,3,4 y por otro lado la 5,6,7 el mínimo interno de la primera componente es 7 y de la segunda es 4, pero al unirlas el mínimo interno es 1. Por lo tanto a la hora de elegir el mínimo de una nueva componente conexa se tienen que tener en cuenta estos 3 valores (el mínimo de la primera componente, el de la segunda y el mínimo que conecta a ambas componentes). Una vez que tenemos el nuevo mínimo calculado de esta forma se tiene que actualizar la información de los mínimos con las demás componentes (para poder utilizarlo en el próximo paso). La arista mínima que conecta a una nueva componente (que resulta de unir A con B) con cualquier otra componente C es el mínimo de la mínima arista entre A y C y la mínima arista entre B y C. La actualización del máximo es más directa ya que esta solo depende de las aristas que salen, entonces simplemente se hace el mismo proceso que para actualizar los mínimos con todas las demás componentes conexas de manera análoga pero con el máximo y se calcula el máximo de todas (calculo el nuevo máximo entre $A \cup B$ y C, y me quedo con la mayor de todos esos máximos). Una vez que tenemos estos valores calculados podemos hacer la comparación correspondiente y determinar si esta nueva componente es candidata o no y continuar con el siguiente paso del algoritmo de Kruskal con la información ya actualizada.

3.2.1. Complejidad

Si miramos el cuerpo del loop del algoritmo de Kruskal [2], podemos ver que el if solo puede ser verdadero n veces, ya que en el peor caso nos queda como AGM un solo árbol en vez de un bosque si el grafo original es conexo, con lo cual solo podemos encontrarnos n veces con alguna arista que une dos árboles de la ejecución de Kruskal. Luego, el cuerpo del if solo podrá ser ejecutado n veces, como todo el código agregado a Kruskal en este caso fue agregado adentro de ese if, la complejidad del cuerpo del if es $O(n\alpha^{-1}(n))$ (hacemos n veces find). El cuerpo del for será $O(n^2\alpha^{-1}(n) + m\alpha^{-1}(n))$ este último término se debe a que evaluar la condición del if es $O(\alpha^{-1}(n))$ y eso se hace m veces, el primer

[2]Kruskal (y Union-Find) de la práctica de este cuatrimestre (2C2022)

término es simplemente repetir n veces el cuerpo del if. Notemos que sabemos que $m \leq n^2$, con lo cual la complejidad final quedaría $O(n^2\alpha^{-1}(n) + mlog(n))$ (este último término es la complejidad del sort).

3.3. Demostración

Para demostrar la correctitud, separamos la demostración en dos partes, primero demostraremos el lema que el enunciado daba como ayuda y luego demostraremos usando ese lema que nuestro algoritmo identifica componentes candidatas:

Sean V, E y $SE : E \mapsto \mathbb{Z}$ el conjunto de vértices (islas), aristas y función de peso (efecto de sinergia) respectivamente

Lema: Toda componente candidata va a ser, en alguna iteración, una de las componentes conexas del bosque que mantiene el algoritmo Kruskal para buscar un árbol generador máximo.

Supongamos que el lema no se cumple, es decir, que existe una candidata que nunca va a ser una de las componentes conexas del bosque de Kruskal maximal. Llamamos a esta candidata $C \subset V$, $E_i = \{(v \to w) \in E | v, w \in C\}$ y $E_e = \{(v \to w) \in E | v \in C \wedge w \notin C \vee v \notin C \wedge w \in C\}$.

Para que C aparezca como componente de Kruskal el algoritmo debe elegir suficientes aristas de E_i para que C sea conexo antes de considerar las de E_e. Asumiendo que esto no sucede, el algoritmo debe considerar una de las aristas de E_e antes que el algoritmo tome suficientes aristas de E_i para que C sea conexo. Pero por definición de candidata, todas las aristas de E_i tienen peso mayor a las de E_e, con lo cual no puede suceder que se considere una arista de E_e antes de una de E_i por qué Kruskal toma aristas en orden de peso (en este caso maximal). Por lo tanto, no puede existir una candidata que no aparezca en alguna iteración como una de las componentes conexas del bosque de Kruskal.

Ahora queremos probar que nuestro algoritmo encuentra todas estas candidatas en las iteraciones de Kruskal.
Lema: Sean C_1 y C_2 dos componentes conexas, E_1 y E_2 sus conjuntos de aristas respectivamente en la i-ésima iteración de Kruskal
La componente $C = C_1 \cup C_2$ es candidata $\iff$ El mínimo entre las aristas que conectan a C_1 y C_2, las internas de C_1 y las internas de C_2 es mayor a la arista de mayor costo entre las que conectan a C_1 y C_2 con otras componentes pero no entre ellas.

$\implies$) Llamemos e_1 a la arista de menor costo entre las componentes y las aristas que las conectan y e_2 a la arista de mayor costo que conectan a las componentes con otras distintas a ellas. La arista e_1 conecta dos nodos en $C_1 \cup C_2$, es decir C, por lo que es (por definición) una arista interna de C. La arista e_2 conecta a algún nodo de $C_1 \cup C_2$ con un nodo que no existe en $C_1 \cup C_2$ (por definición), es una arista externa de C. Como C es candidata toda interna es mayor a toda externa, por lo tanto vale $SE(e_1) > SE(e_2)$.

$\impliedby$)
Toda arista interna de C va a unir o dos nodos de C_1 y C_2 o un nodo de cada uno, de estos tres tipos de aristas sabemos que e_1 será la de menor peso, es decir, e_1 es la artista interna de C con menor peso. Toda arista externa de C conectara a un nodo de C_1 o C_2 con alguno que no esté en ninguno

de esos conjuntos, de estas aristas sabemos que e_2 será la de mayor peso, es decir, e_2 es la arista externa de mayor peso. Por lo tanto vale:

$$\textit{toda arista externa de } C \leq e_2 < e_1 \leq \textit{toda arista interna de } C$$

C es candidata.

El primer lema nos garantiza que toda candidata posible aparecerá en una iteración de nuestro algoritmo para ser evaluada. El segundo lema nos garantiza que cualquier componente que aparezca en alguna iteración será candidata si y solo si cumple con nuestras guardas.

4. Problema 3: Camino Mínimo

4.1. Problema

En este problema hay una iglesia en la cual tenemos una bolsa con donaciones y un portero que quiere robar la mayor cantidad de monedas de la bolsa. En esta iglesia hay también feligreses que se van pasando la bolsa con ciertas reglas y agregando monedas hasta llegar a una capacidad c (una regla consiste en agregar una cantidad de monedas y pasar la bolsa a otro feligrés o al portero). Cuando se llega a esa capacidad se le devuelve la bolsa al sacerdote. El portero solo puede robar monedas cuando le pasan la bolsa y de una a la vez. El problema consiste en determinar un algoritmo para maximizar la cantidad de monedas que se puede robar el portero antes de que le llegue la bolsa al sacerdote, ya que en ese momento no va a poder robar más.

4.2. Solución

Si pensamos el problema como un digrafo, donde cada vértice representa un feligrés y cada arista una regla, podemos ver que este se puede reducir a encontrar el ciclo de costo mínimo que pasa por el portero. Esto se debe a que el peso de ese ciclo va a representar cuantas monedas se pusieron en la caja si los feligreses siguen esas reglas. Como c es un valor fijo y el portero siempre saca solamente una moneda, este ciclo mínimo va a permitir que se realicen la mayor cantidad de vueltas por él y así maximizar la cantidad de monedas que toma el portero, si se tomara un ciclo con costo más grande estaríamos acercándonos a c más rápidamente que si hubiéramos tomado el otro ciclo.

Para empezar, notamos que por como está definido el problema y la entrada si convertimos nuestras reglas en aristas directamente, terminaremos con un multigrafo y no un digrafo porque puede haber varias reglas entre dos feligreses. Debido a la naturaleza de nuestro problema, dadas más de una regla entre dos feligreses, solo nos interesa la de menor costo porque esas son las únicas aristas que aparecerán en los ciclos de costo mínimo. Por lo tanto, primero ordenamos las aristas por su origen, luego por su destino y por último su costo. Después esta lista ordenada la recorremos linealmente y vamos construyendo el digrafo representado como lista de adyacencias, agregando solamente la regla de menor costo entre las que comparten origen y destino.

Ahora teniendo el digrafo, debemos conseguir el ciclo de menor costo que contenga al portero. Esto sería equivalente a dado el conjunto de todos los nodos que tienen una arista que apunta al portero, hallar la distancia del camino mínimo del portero a esos nodos y sumarle el costo de la arista que cierra el ciclo y al final quedarnos con el de menor costo. Para ello usamos Dijkstra desde el portero en el digrafo para conseguir las distancias desde el portero hacia todos los nodos. Cuando

creamos el grafo aparte nos guardamos las aristas que le llegan al portero, a las distancias encontradas les sumamos sus respectivas aristas para completar el ciclo y nos quedamos con el costo más bajo.

Finalmente, sabiendo cuál es el costo del ciclo de costo mínimo y la capacidad c, calculamos la cantidad de vueltas que puede dar el ciclo antes de llegar a c, que equivale a la cantidad de monedas que puede robar.

4.2.1. Complejidad

En cuanto a la complejidad, creamos primero un multigrafo representado como un conjunto de aristas, luego ordenamos ascendentemente ese conjunto primero por origen, luego por destino y finalmente por costo ($O(rlog(r))$), notemos que si hay más de una arista que conecta el mismo par de nodos en el mismo sentido estas quedaran juntas, nos interesa quedarnos con la primera que aparezca para cada par origen-destino. Si bien esta complejidad podría ser mayor a $O(rlog(p))$ no lo es porque r es a lo sumo $1000*p$ (cada feligrés tiene a lo sumo 1000 reglas) por lo tanto, se cumple la complejidad pedida. Aun así, si r no estuviera acotada, se podría realizar este ordenamiento utilizando radix sort y ordenar por origen-destino y después recorrer linealmente para sacar la menor de cada par. Luego, creamos un digrafo nuevo representado como lista de adyacencias ($O(p)$) y recorremos en orden el conjunto de aristas del multigrafo, agregando al digrafo las aristas de menor costo que conectan cada par de nodos ($O(r+p)$). Por último, corremos Dijkstra [3] (versión min-heap) desde el portero en el digrafo para conseguir las distancias ($O(rlog(p))$) y luego recorremos el vector de los feligreses que devuelven la caja al portero buscando el que genera el ciclo de menor costo ($O(p)$). Finalmente, la complejidad quedaría $O(rlog(p)+r+p) = O(rlog(p)+p) = O(rlog(r))$.

4.3. Demostración

Comenzamos partiendo desde el digrafo, queremos demostrar que el resultado de nuestro algoritmo no existe una forma de que la caja pase más veces por el portero. Podemos separar esto en dos partes, primero que nuestro algoritmo encuentra el ciclo mínimo, segundo que un ciclo de costo menor permite robar una cantidad de monedas mayor o igual a un ciclo de costo mayor:

Dado V el conjunto de nodos (feligreses + portero), E el conjunto aristas (reglas) y W la función de peso $E \mapsto \mathbb{Z}$ (cantidad de monedas que suma cada regla). El digrafo $G(V, E, W)$ representa la iglesia.

Sea $v_0 \in V$ el portero y $A \subset V$ tq $a \in A \iff (a \to v_0) \in E$ el conjunto de todos los nodos (feligreses) que tienen una arista que apunta a v_0 entonces:
$\forall a \in A \mid \exists\ p_a$ un camino de v_0 a $a \implies p_a + (a \to v_0)$ forma un ciclo que contiene a al portero.
Dado un ciclo $J = v_0, v_1, ..., v_i, v_0$ definimos $W(J) = \sum_{e \in J} W(e)$ esta sumatoria se puede expresar como: $W(v_0 \rightsquigarrow v_i) + W(v_i \to v_0)$

Nuestro algoritmo elige el ciclo que minimiza esta suma, ya que como caminos tomamos los mínimos (osea la distancia de v_0 a a) y luego buscamos el nodo $a_r \in A$ que minimiza $d(v_0, a_r) + w(a_r \to v_0)$. Al usar Dijkstra en v_0 obtenemos $d(v_0, v_i)$ y particularmente $d(v_0, a_r)$. Supongamos que existe un ciclo de menor costo $\exists d'(v_0, a_r)$ t.q. $d'(v_0, a_r) + w((a_r \to v_0)) < d(v_0, a_r) + w((a_r \to v_0))$ entonces:

$$d'(v_0, a_r) + (a_r \to v_0) < d(v_0, a_r) + (a_r \to v_0) \iff$$

[3] Usamos la implementación de dijkstra de la práctica primer cuatrimestre 2022

$$d'(v_0, a_r) < d(v_0, a_r)$$

pero esto es absurdo, pues Dijkstra nos da el costo del camino mínimo de v_0 a cualquier nodo, así que $\nexists\, d'(v_0, a_r)$, por lo tanto no existe ciclo más corto. Si $A = \emptyset$ decimos que el ciclo de menor costo tiene costo infinito, con lo cual el portero no podrá robar ninguna moneda, algo similar pasa para cada nodo $a \in A$ t.q. $\nexists v_0 \rightsquigarrow a$ en ese caso definimos $W(v_0 \rightsquigarrow a) = \infty$.

Para demostrar lo segundo tomamos dos ciclos S, J donde $W(S) \leq W(J)$. Nuestra respuesta es la cantidad de veces que pasamos por el inicio del ciclo (v_0) antes que la sumatoria de las aristas recorridas sume c (restándole uno porque no se roba en el primer ciclo). Si queremos probar que S permite encontrar una mejor respuesta que J debe cumplirse lo siguiente: (Considerando que $c > 1$ ya que en caso contrario, la respuesta es 0 y esto nos permite dividir por $c - 1$)

$$\left\lceil \frac{c-1}{W(J)-1} \right\rceil - 1 \leq \left\lceil \frac{c-1}{W(S)-1} \right\rceil - 1 \iff$$

$$\left\lceil \frac{c-1}{W(J)-1} \right\rceil \leq \left\lceil \frac{c-1}{W(S)-1} \right\rceil \iff (\textit{ceiling es creciente})$$

$$W(J) - 1 \geq W(S) - 1 \iff$$

$$W(J) \geq W(S)$$

Esta es la manera en la que definimos a S y J por lo que siempre se cumple. Así demostramos que nuestro algoritmo encuentra el ciclo de costo mínimo y también que un ciclo con menor peso permite robar más que cualquiera de mayor peso, por lo tanto se resuelve el problema.

5. Problema 4: SRD

5.1. Problema

En este problema tenemos un sistema de ecuaciones de la forma $x_i - x_j \leq c$ y una lista ordenada de valores posibles para cada variable, buscamos decidir si es posible asignar un número de la lista a cada una de las variables y cumplir con todas las ecuaciones del sistema, en caso de que sea posible, dar alguna asignación posible válida.

5.2. Solución

Para resolver este problema implementamos el algoritmo de fishburn. La idea del algoritmo es empezar considerando la asignación posible en la cual todas las variables serán iguales al número más grande de nuestra lista ordenada, luego ira modificando las asignaciones para las variables que no

cumplen alguna ecuación. Para hacer esto, considera la ecuación $x_i - x_j \leq c$ de la forma $x_i \leq x_j + c$, si esa ecuación no se cumple, buscamos achicar x_i asignándole un número menor en la lista al que tenía antes, de forma tal que se cumpla la ecuación que antes no se cumplía, pero buscamos hacerlo con el valor más grande posible de la lista. Notemos que siempre repetimos este proceso para cada ecuación, con lo cual los valores de las variables solo pueden disminuir, entonces si una ecuación no se cumple para el menor valor de nuestra lista, no se puede satisfacer al sistema de ecuaciones. Repetimos este proceso para cada ecuación hasta que en alguna iteración pasemos por todas las ecuaciones sin cambiar el valor de ninguna variable, esto puede suceder por dos motivos: todas las ecuaciones se cumplen con esos valores o alguna no se cumple con ninguno de los valores posibles de la lista. Por último, debemos revisar en cuál de estos casos estamos (revisando si cumplimos todas las ecuaciones usando las asignaciones actuales de las variables) si se cumplían todas las ecuaciones sabemos que la asignación que tenemos es válida y la devolvemos, caso contrario sabemos que el sistema es insatisfacible con lo cual respondemos eso.

5.3. Complejidad

En cuanto a la complejidad, D es la lista ordenada de candidatos (tamaño m), X el conjunto de variables (tamaño n) y E el conjunto de ecuaciones (tamaño k). Como se explica en fishburn, como peor caso dentro del do while loop se reduce una variable solo una vez hasta llegar al inicio de las opciones de D. Esto nos indica que en el peor caso la cantidad de iteraciones del do while será $O(mn)$. Dentro de cada iteración se recorren todas las ecuaciones en E y dentro de cada iteración de las ecuaciones se recorre D para buscar el mayor candidato que cumpla la ecuación. Visto así nuestra cota terminaría siendo $O(m^2nk)$. Pero tenemos que considerar que estamos analizando el peor caso. En este caso dijimos que cada variable se reduce de a un valor de D, como D está ordenado y en nuestra búsqueda siempre empezamos desde el último valor que elegimos, si es que reduce uno cada iteración siempre se elegirá el siguiente al anterior y esa última búsqueda es $O(1)$. Por lo tanto, nuestra cota de peor caso termina siendo $O(mnk)$