González Alvarado Raúl
313245312

Tarea 4

Análisis de Algoritmos
30 de octubre de 2019

1. (El problema del escape) Una malla de $n \times n$ es una gráfica compuesta por n filas y n columnas de vértices. Denotamos el vértice en la i-ésima fila y j-ésima columna como (i, j). Todos los vértices en una malla tienen exactamente cuatro vecinos, excepto aquellos en la frontera, que son los vértices (i, j), con $i = 1$, $i = n$, $j = 1$ o $j = n$.

   Dados $m \leq n^2$ vértices de arranque (x_1, y_1), (x_2, y_2),..., (x_n, y_n) en la malla, el problema del escape es decidir si existen m trayectorias ajenas por vértices que conecten a cada vértice de arranque con algún vértice den la frontera (distintos).

   a) Considere una red de flujos cuyos vértices, así como las aristar, tengan capacidades. Esto es, el flujo positivo que entra a cualquier vértice está sujeto a una restricción de capacidad. Muestre que determinar el flujo máximo con capacidades tanto en los vértices como aristas puede ser reducido a un problema de flujo máximo ordinario en una red de flujo con tamaño similar.

   Respuesta: Para mostrar que puede ser reducido a el problema de flujo máximo vamos a explicar cómo se va a modelar este problema para que pueda ser representado por una gráfica. Empezamos notando que ya tenemos una malla **M** de tamaño $n \times n$ la cuál ya puede ser vista como una gráfica, pero además tenemos dos tipos de vértices destacados: los de la frontera (que son las *salidas* o *destinos*) y los nodos de *arranque*. A los nodos de la frontera los llamaremos **F** y a los de arranque **A**.

   Construiremos una gráfica **G** la cuál va a tener a todos los nodos y aríastas de **M** y otros dos nodos destacados que serán uno de inicio (**s**) y uno de fin (**e**). El nodo **s** va a ser adyacente a todos los nodos en **A** y el nodo **e** va a ser adyacente a todos los nodos en **F**. Para terminar con el modelado del problema necesitamos que todos los nodos y aristas tengan un peso que será de 1.

   La modelamos de esta forma para que podamos usar el nodo **s** como un punto de partida y **e** como punto final, así estamos obligados a que todos las trayectorias tengan que pasar primero por un nodo en **A** y antes de llegar a **e** tengan que pasar por un nodo en **F**, de este modo representamos que se tomó escapó desde un nodo en **A** hasta uno de **F**. El peso que ponemos en las aristas y nodos es para limitar que las personas pasen una sola vez por la arista o nodo, y necesitamos de este peso para asegurarnos que las trayectorias que estamos buscando sean ajenas por vértices y también por aristas. Cabe destacar el peso en los nodos, ya que si no tuvieran entonces tendríamos una gráfica de flujos como las que se han estado ocupando en clase, pero necesitamos de estos pesos por lo explicado anteriormente.

   Podemos decir que nuestra gráfica **G** es una gráfica de flujos *tradicional* por que (aún que en lo único que cambia es en que los vértices tienen un peso) el peso de los vértices solo se utilizará de igual forma que la capacidad de las aristas y del mismo modo se disminuirá en uno cuando estemos utilizando dicha arista. Así que se vuelve un problema de flujo máximo por que debemos buscar el flujo máximo de nuestro vértice **s** a **e**.

   b) Describe un algoritmo eficiente que de solución al problema del escape y analice su tiempo de ejecución.

   Respuesta: Como vimo en el inciso anterior, tenemos ante nosotros un problema de flujo máximo con la variación de que tenemos un peso en los vértices. Entonces para resolver nuestro problema vamos a describir y modificar un poco el algoritmo de *Ford-Fulkerson* que tome en cuenta dicha variación.

2. Sea $\mathbf{G} = (V, E)$ una gráfica conexa no dirigida. Dar un algoritmo $O(V + E)$ para obtener un camino en **G** que atraviese cada arista en E exactamente una vez en cada dirección. Describe como puedes

encontrar un camino fuera de un laverinto si tienes una cantidad muy grande de monedas.

Respuesta: Primero vamos a describir el algoritmo para atravezar dos veces cada arista.

Supondremos que cada nodo u tiene una lista *vecinos* con una referencia a los vecinos de u. También tendremos una lista vecinos-explorados que contendrá los vecinos que ya no necesitan ser explorados desde u. Por último utilizaremos una pila *padre*, que contendrá en el tope al último nodo por el que se llegó a u (empiezan con la pila vacía).

a) Inicialmente vamos a tomar un nodo cualquiera, digamos r y empieza siendo el nodo actual.

b) Para el nodo actual u vamos a verificar si su lista *vecinos* es igual a su lista vecinos-explorados, en caso de que sí:

   a) Si *padre* ya está vacía, entonces termina el algoritmo.

   b) Si no es vacía, entonces nuestro nodo actual va a ser el resultado de hacer pop en *padre* (sacamos el nodo de la pila) de u.

   En caso de que aún *vecinos* sea distinto a vecinos-explorados vamos a seleccionar un nodo de *vecinos* que no esté en vecinos-explorados, digamos s, metemos a s en vecinos-explorados, hacemos a s el nodo actual y por último añadimos a u en la lista vecinos-explorados de s. Repetimos *b)*.

Vamos a notar lo siguiente: las aristas solo se recorren cuando nos movemos a un nodo que no está en nuestra lista vecinos-explorados, y cuando volvemos hacia nuestro padre (por que ya no tenemos vecinos no explorados). También podemos ver que si se llegó por un nodo u a un nodo s entonces ya no nos movemos a u (desde s) a menos que hayamos explorado todos los vecinos de s, por lo tanto nos movimos por la arista (u, s) una vez para llegar de u a s y otra vez para volver de s a u. Y no hay otra ocasión en la que nos movamos por las arista. Además recorremos siempre todas las aristas ya que hacemos una búsqueda por profundidad, por lo tanto siempre recorremos todas y además pasamos por ellas dos veces.

El tiempo que tarda en ejecutarse este algoritmo es $O(E)$ ya que recorre dos veces todas las aristas.

Ahora para el problema del laverinto basta con aplicar el algoritmo descrito y viendo cada intersección del laverinto como un nodo y cada pasillo entre intersecciones como una arista. También necesitamos una forma de guardar información (la pila *padre* y vecinos-explorados) en las intersecciones del laverinto, lo cuál vamos a hacer con ayuda de las monedas de la siguiente forma:

a) Para los vecinos-explorados: Cuando añadimos un nodo a la lista vecinos-explorados de una intersección, vamos a representarlo con una moneda puesta en la entrada del pasillo que lleva al nodo que se quiere guardar.

b) Para la pila *padre*: Cuando queramos guardar un elemento en la pila primero vamos a ver si hay dos o más monedas en alguno de los pasillos de la intersección por la que llegamos, si no hay en ninguno más de dos, ponemos una moneda extra para marcar que es un padre. si ya hay una o más monedas en un pasillo, entonces nos fijaremos en el pasillo, digamos c, con más monedas y colocaremos en el pasillo que es el nuevo padre una moneda más de las que había en c.

c) Para cuando queramos hacer pop de la pila *padre* nos fijaremos en el pasillo c con más monedas, quitamos las monedas dejando solo una y nos movemos a la intersección que ese pasillo lleva.

El algoritmo se aplica hasta que se llegue a una intersección que tenga una salida.

3. Supongamos que una gráfica **G** tiene un árbol generador de peso mínimo $mathbfT$ que ya fué calculado. Qué tan rápido se puede actualizar el árbol de peso mínimo si un nuevo vértice con aristas en agregado a **G**?.

   Respuesta: Nos va a tomar $O(n \log n)$ siendo n el número de vértices originales de **G**.

   Para mostrarlo primero vamos a describir el algoritmo de **Prim**, que calcula el árbol generador de peso mínimo de una gráfica:

   a) Primero toma un vértice cualquiera y lo añade al árbol.
   b) De los nodos adyacentes al árbol les asigna una arista mínima (es la arista de peso mínimo que conecta al vértice con el árbol). Selecciona el vértice con la menor arista y lo añade al árbol.
   c) Repite el paso anterior hasta que ya no quedan vértices por añadir al árbol.

   Para la selección del nuevo vértice con la arísta mínima que se va a añadir al árbol, se hace uso de un *min-heap* con lo cuál obtenemos una complejidad de $O(|E| \log n)$ para este algorimo. Con esto dicho podemos seguir.

   Vamos a unir el nuevo vértice solamente con el árbol generador **T**. No usamos alsa aristas de **G** ya que solo maximizarían o mantendrían igual el peso de **T**. Lo unimos a **T** y no a **G** para así mantener la conexidad de la gráfica y al mismo tiempo mantenerla con la menor cantidad de aristas posibles sin perder el peso mínimo de la gráfica. El árbol generador **T** tiene actualmente $n-1$ aristas, así que al añadir el nuevo vértice con sus aristas (puede a lo más tener n aristas) nos quedaría una gráfica $\mathbf{G_1} = (V_1, E_1)$ tal que $|V_1| = n+1$ y $n \leq |E_1| \leq 2n+1$. Al aplicar el algoritmo de **Prim** tenemos entonces que su complejidad de tiempo es $O((2n+1)\log(n+1)) = O(n \log n)$.

4. Da un algoritmo eficiente para encontrar la longitud del ciclo mínimo de peso negativo en una gráfica.

   Respuesta: Vamos a utilizar una variación del algoritmo de **Floyd-Warshall**.

   Primero vamos a explicar como funciona dicho algoritmo:

   Suponiendo que tenemos la matríz de adyacencia M de nuestra gráfica, donde guardaremos en cada casilla el peso de la arista que las une. También utilizaremos otra matríz O del mismo tamaño que M.

   a) Tomamos k = 1
   b) Para toda i y j hacemos: si $M_{i,k} + M_{k,j} < M_{i,j}$ entonces $O_{i,j} = O_{k,j}$ y $M_{i,j} = M_{i,k} + M_{k,j}$.
   c) Si $k \leq n$ (con n el número de vértices de **G**) entonces detenemos el algoritmo, en otro caso hacemos $k = k+1$ y repetimos el paso anterior.

   Ahora a este algoritmo le haremos la modificación siguiente: vamos a detenernos cuando algúna casilla de la diagonal de M sea menor a 0, es decir que $M_{i,i} < 0$ en alguna iteración k.

   De esta forma obtenemos un camino de el vértice i a sí mismo, es decir un ciclo. Y además lo detenemos en el momento que encontremos uno de peso negativo. Como el algoritmo empieza encontrando caminos que pasan a lo más por un vértice, luego por dos, ya así sucesivamente, va aumentando en cada iteración en uno, entonces así nos aseguramos de que en cuanto encontremos una casilla de la diagonal negativa la k va a ser la mínima. Así que obtendríamos el tamaño mínimo de el ciclo de peso negativo de longitud mínima.

   La complejidad de este algoritmo se mantine $O(n^3)$ pero podría detenerse antes por la condición de la casilla diagonal negativa, así si la k es más chica podríamos decir que es $O(k * n^2)$.

5. Diseña un algoritmo de tiempo $O(V + E)$ que determine si una gráfica dirigida $G = (V, E)$ contine o no un ciclo.

   Respuesta: Vamos a utilizar un algoritmo que recorra nuestra gráfica de forma similar a el del ejercicio 2. En lo que va a cambiar es que se va a utilizar una variable extra que será un çolor", con el cuál vamos a marcar el nodo cuando ya hayamos pasado por el. Se van a utilizar dos colores $color_A$ y $color_B$. También vamos a suponer que la lista *vecinos* de cada nodo u contendrá los nodos v tal que existe la arista dirigida que va de u a v i.e. $(u, v) \in E$. Además la pila *padre* que utilizamos antes ahora va a ser una sola variable.

   I) Inicialmente vamos a tomar un nodo cualquiera, digamos r y empieza siendo el nodo actual.

   II) Para el nodo actual u vamos a verificar si su lista *vecinos* es igual a su lista vecinos-explorados, en caso de que sí:

   Primero le asignaremos a u el $color_B$ y luego revisaremos los dos casos siguientes:

   a) Si *padre* no está definido (es *null*), entonces termina el algoritmo y reportamos que no hubo ciclo.
   b) Si no es vacía, entonces nuestro nodo actual va a ser el *padre* de u.

   En caso de que aún *vecinos* sea distinto a vecinos-explorados vamos a seleccionar un nodo de *vecinos* que no esté en vecinos-explorados, digamos s, metemos a s en vecinos-explorados y comprobamos los siguientes casos:

   a) Si s tiene el color $color_A$, entonces terminamos el algoritmo y reportamos un ciclo.
   b) Si s tiene el color $color_B$, entonces nodo actual sigue siendo u y repetimos el paso II).
   c) Si s no tiene ningún color, entonces hacemos a s el nodo actual, a la variable *padre* de s le asignamos u, le asignamos a s el $color_A$ y repetimos el paso II).

   Una suposición adicional es que si se termina el algoritmo (entra en la condición *b)* del inciso *II)*) entonces nos situaremos en un nodo que aún no tenga color asignado y repetiremos el algoritmo hasta que no queden nodos sin colorear Una vez que se hayan coloreado todos los nodos y no se haya reportado un ciclo entonces decimos que no hay ciclos.

   Para mostrar que es correcto veremos las siguientes afirmaciones sobre un nodo:

   Afirmación 1. No tiene color si no lo hemos explorado. Esto pasa por que los nodos empiezan con su variable color en null, y no se les asigna un color hasta que se llega a ellos. Concluimos que entonces no han sido descubiertos.

   Afirmación 2. Tiene el $color_B$ si ya fueron explorados todos sus vecinos y tienen el $color_B$. Esto pasa por el inciso *II)* al momento de revisar la lista de vecinos y explorados. Además no se le cambia el color a el $color_B$ en ninguna otra ocasión.

   Afirmación 3. Tiene el $color_A$ cuando aún tiene vecinos que están siendo explorados o si tiene vecinos aún sin explorar. Esto pasa por la afirmación 1 y 2, no se le asigna el $color_A$ hasta que es explorado por primera vez, y no se le asigna el $color_B$ hasta que todos sus vecinos fueron explorados. Además no se le asigna el $color_A$ en ninguna otra ocasión.

   Así que, por la condición: "Si s tiene el color $color_A$, entonces terminamos el algoritmo y reportamos un ciclo", sabemos que el *padre* de s aún tiene el $color_A$ (por que s aún está siendo explorado). Y entonces si s ya tenía el $color_A$ sabemos que llegamos desde un nodo que tiene vecinos que están siendo explorados hasta él mismo (en este caso s), lo cuál forma un ciclo y lo podemos reportar.

La complejidad de este algoritmo es la misma que el del ejercicio 2, por que no altera el comportamiento solo aumenta en una variable la memoria utilizada. Aún que cabe destacar que puede terminar antes. Pero sigue siendo $O(E)$.

6. Supongamos que tenemos un conjunto de n ciudades $c_1, c_2, ..., c_n$, y una tabla $D[1, ..., n][1, ..., n]$ tal que $D[i, j]$ es la longitud de una carretera que une a la ciudadad c_i con la ciudad c_j (este valor puede ser ∞ si no hay carreterra entre c_i y c_j). Encuentre un algoritmo eficiente que encuentre la ruta más corta entre las ciudades c_1 y c_n tal que dicha ruta no pasa por más de k ciudades distintas (a c_1 y c_n).

   Respuesta: Nuevamente vamos a utilizar **Floyd-Warshall**, pero esta vez deteniendo nuestro algoritmo en la k-ésima iteración (esto para obtener los caminos de a lo más longitud k).

   Sabemos que **Floyd-Warshall** encuentra (en cada iteración k) el camino más corto entre dos nodos que utiliza a lo más k nodos intermedios, con lo cuál ya tendríamos el tamaño de la ruta entre nuestras dos ciudades c_1 y c_n. Y la complejidad de este algoritmo sería nuevamente $O(k * n^2)$ con $n = |V|$ por que se detine en la k-ésima iteración.

7. Sea G una gráfica cuyas aristas tienen asignados pesos positivos. Sea T un árbol generador de peso mínimo de G. Pruebe que si existen aristas $e \in T$ y $e' \notin T$ tales que $(T - e) \cup e'$ forman un árbol de mayor pero o igual que T, pero menor o igual a cualquier otro árbol generador de G, i.e. un segundo árbol generador de peso mínimo.

   Respuesta: Para este problema vamos a dar un algoritmo que encuentre dichas aristas (e y e'). Llamaremos T' al árbol que estamos buscando (i.e. el segundo árbol generador de peso mínimo).

   Antes vamos a considerar los siguientes casos para las aristas e y e':

   a) Si $(T - e) \cup e'$ forma un ciclo (o nos deja la gráfica inconexa) entonces esa e' no es la que estamos buscando.
   b) Si $(T - e) \cup e'$ no forma un ciclo, analicemos los siguientes dos casos:
      1) $peso(e') < peso(e)$: Esto es un caso imposible, por que si pasa significa que tenemos un árbol que tiene un peso menor al de T, por lo tanto T n sería el de peso mínimo.
      2) $peso(e') \geq peso(e)$: En este caso tenemos que $(T - e) \cup e'$ sí es un árbol generador posible y que su peso es mayor o igual al peso de T.

   Entonces, por el último caso, para encontrar a T' tenemos que encontrar a e y e' que acerquen lo más posible el peso de T' al de T. Esto se logra haciendo que la diferencia entre e y e' sea la menor posible.

   Para lograr eso vamos a utilizar el siguiente algoritmo:

   Vamos a utilizar una variable $diferencia$ (inicializada en ∞) en la que iremos guardando la menor diferencia entre e y e', una variable $arista'$ en la que iremos guardando la arista e' y otra $arista$ donde guardaremos e.

   Iteramos por cada arista $e \in T$ y por cada una también iteramos por cada arista $e' \notin T$ y hacemos lo siguiente:

   a) Si $peso(e') - peso(e) < diferencia$ entonces buscamos si $(T - e) \cup e'$ forma un ciclo.
      1) En caso de que lo forme entonces no hacemos nada y continuamos la iteración.
      2) En caso contrario asignamos a $diferencia = peso(e') - peso(e)$ y a $arista = e$ y en $arista' = e'$.
   b) Si $peso(e') - peso(e) \geq diferencia$ no hacemos nada y continuamos la iteración.

Una vez terminadas las iteraciones tenemos en $arista'$ la e' y en $arista$ a e.

Cabe destacar que en el caso particular en que G sea un árbol, su árbol generador de peso mínimo sería él mismo, de este modo no podemos encontrar ninguna arista $e' \notin T$, por lo tanto no podemos probar la existencia de e' y por lo tanto es un caso en el que no se cumple.

Para justificar las aristas e y e' que nos da nuestro algoritmos son tales que $(T - e) \cup e' = T'$ es el segundo árbol generador de peso mínimo vamos a primero ver que $peso(T') \geq peso(T)$.

Como ya mencionamos en el analisis de los casos, no es posible que $peso(e') < peso(e)$ así que siempre escojemos una e' tal que $peso(e') \geq peso(e)$. Ahora para ver que $peso(T') > peso(T)$:

$$peso(T') = peso(T) - peso(e) + peso(e')$$

Y como $peso(e') < peso(e)$ entonces $peso(e) - peso(e') \geq 0$. Tenemos que:

$$peso(T') \geq peso(T)$$

Ahora para mostrar que $peso(T')$ es menos o igual a cualquier otro árbol generador (exceptuando T), basta con ver el inciso *a)* de nuestro algoritmo, que ya nos da el mínimo e' tal que $peso(e) - peso(e')$ es el mínimo, además de mencionar que se prueban todas las combinaciones de aristas. Por lo tanto minimizamos la diferencia de nuestras aristas e y e' y tampoco formamos ciclos (que es lo que queríamos en nuestro análisis).