Estructuras de Datos y Algoritmos – iic2133
Control 3

17 de abril, 2019

Nombre: _____

1) Una tienda quiere premiar a sus k clientes más rentables. Para ello cuenta con la lista de las n compras de los últimos años, en que cada compra es una tupla de la forma $(ID\ Cliente, Monto)$. La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras.

Explica cómo usar *tablas de hash* y *heaps* para resolver este problema en tiempo esperado —o promedio— $O(n + n \log k)$.

Solución: El problema se separa en dos partes. Se asignará puntaje por separado para cada una.

- **Calcular la rentabilidad de cada cliente (2pts)**

Queremos obtener el set de tuplas $(ID\ Cliente, Rentabilidad)$, donde la rentabilidad para el $ID\ Cliente = i$ es la suma de todos los montos de las tuplas de la forma $(i, Monto)$.

Para esto creamos una tabla de hash T donde se almacenan tuplas $(ID\ Cliente, Monto)$. Cada vez que se inserte un $ID\ Cliente$:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las n tuplas hemos efectivamente encontrado la rentabilidad para cada cliente. **[1pt]**

La inserción en esta tabla tiene tiempo esperado $O(1)$ como se ha visto en clases. Como se realizan n inserciones, esta parte tiene tiempo esperado $O(n)$. **[1pt]**

- **Buscar los *k* clientes más rentables (4pts)**

Posible solución:

Sea m el total de clientes distintos. Creamos un *min-heap* de tamaño k que contendrá los k clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k* más rentables** encontrados hasta el momento. **[1pt]**

Iteramos sobre las tuplas en T, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre T. Para cada elemento que

veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *sift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior. **[2pts]** (También es posible extraer la cabeza del heap e insertar el nuevo elemento normalmente)

Cada inserción en el heap toma $O(log\ k)$. En el peor caso insertamos los m elementos en el heap, por lo que esta parte es $O(m\ log\ k)$. Pero como en el peor caso $m = n$, esta parte es $O(n\ log\ k)$. **[1pt]**

Para cada sección, no se asignará puntaje si no explican correctamente el método propuesto o este no resuelve correctamente el problema. Para la segunda sección, se asignará como máximo 1pt si no alcanzan la complejidad solicitada.

2) La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores en existencia en todos los multiversos, y necesita ayuda para hacer más eficientes la atención de los pedidos, ya que recibe millones por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando una pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar *tablas de hash* para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

Solución: Necesitamos resolver las siguientes operaciones:

- Registrar pedido (*pizza*, *nombre*). Para una misma *pizza* los nombres deben guardarse por orden de llegada.
- Buscar el siguiente *nombre* para una *pizza* dada.
- Eliminar el pedido del sistema.

Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo *key*. En este caso *key* corresponde al tipo de pizza y un *value* corresponde al nombre de la persona que lo pidió. **[1.5pts]**

Los *values* de un mismo *key* se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea $O(1)$ y se atienda en orden de llegada. **[1pt].** (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son $O(log\ n)$, por lo que sólo obtienen **[0.5pts]**)

La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura. **[1pt]**

Pero como el dominio de las *key* es infinito, debemos ir despejando las celdas de la tabla cuando una *key* se queda sin *values*, ya que no tenemos memoria infinita. **[0.5pts]** Para esto es necesario usar **encadenamiento**, ya que permite eliminar *key*s de la tabla sin perjudicar el rendimiento de esta. **[2pts]**

Estructuras de Datos y Algoritmos - IIC2133
Control 4
6 de mayo, 2019

1) Un *puente* en un grafo no direccional es una arista (*u, v*) tal que al sacarla del grafo hace que el grafo quede desconectado -o, más precisamente, aumenta el número de componentes conectadas del grafo; en otras palabras, la única forma para ir de *u* a *v* en el grafo es a través de la arista (*u, v*). **[4 pts.]** Explica cómo usar el algoritmo DFS para encontrar eficientemente los puentes de un grafo no direccional; y **[2 pts.]** justifica qué tan eficientemente. Recuerda que DFS asigna tiempos de descubrimiento (y finalización) a cada vértice que visita.

R: Para hallar los puentes en el grafo lo que se hará, básicamente, es buscar aquellas aristas que no pertenezcan a algún ciclo dentro de este. Es posible encontrar dichas aristas utilizando el algoritmo DFS. Teniendo nuestro bosque, generado por el algoritmo, supongamos que tenemos una arista (*u, v*) y que en el bosque no es posible llegar desde un descendiente de *v* hasta un ancestro de *u*, entonces diremos que dicha arista no pertenece a ningún ciclo y en consecuencia, es un puente. Para detectar de manera eficiente si una arista es un puente, lo que se hará es guardar, en cada nodo, el menor de los tiempos de descubrimiento de cualquier nodo alcanzable (no necesariamente en un paso) desde donde me encuentre, llamémoslo *v.low*. Si la arista es la (*u, v*), esto es:

$$u.low = min\{u.d, v.low\}$$

Ahora, al momento de retornar el método *dfsVisit*, lo que se hará es comparar dicho valor con el tiempo de descubrimiento del nodo en el que estoy parado. Digamos estoy parado en el nodo *u* y visité *v*, si se tiene que $u.d < v.low$ entonces no se puede alcanzar ningún ancestro de *u* desde algún descendiente de *v*, en consecuencia dicha arista no pertenece a ningún ciclo y es un puente.

Esta forma de detectar puentes en el grafo posee la misma complejidad que el algoritmo DFS, O(|*V*| + |*E*|). Esto ya que lo único que se está haciendo es actualizar un valor y compararlo, lo que no suma mayor complejidad (se hace en tiempo constante).

- **[4 pts]** Se explica un algoritmo o bien las modificaciones que se le deben realizar a DFS para lograr el objetivo de manera clara y concisa. Solución debe ser eficiente. Si no se cumple con los puntos anteriores no hay puntaje.
- **[2 pts]** Solo si el algoritmo es correcto (efectivamente encuentra los puentes) y se justifica el por qué de la eficiencia mostrada.

2) Sea *G*(*V, E*) un grafo no direccional y $C \subseteq V$ un subconjunto de sus vértices. Se dice que *C* es un *k-clique* si |*C*| = *k* y todos vértices de *C* están conectados con todos los otros vértices de *C*.

Dado un grafo cualquiera *G*(*V, E*) y un número *k*, queremos determinar si existe un *k-clique* dentro de *G*. Esto se puede resolver usando *backtracking*.

a) [2pts.] Describe la modelación requerida para aplicar *backtracking* a este problema: explica cuál es el conjunto de **variables**, cuáles son sus **dominios**, y describe en palabras cuáles son las **restricciones** sobre los valores que pueden tomar dichas variables.

R: Cualquier descripción que describa correctamente la modelación y explique de manera clara las variables, los dominios de las variables y restricciones sobre los valores que pueden tomar dichas variables, de tal forma que se pueda resolver con backtracking, tendrá el puntaje correspondiente. Recordar que al ser no direccional, si (v, v') ϵ E $\rightarrow$ (v', v) ϵ E. A continuación, se plantean dos posibles formas de solucionar el problema:

Propuesta 1:

Las **variables** son nodos pertenecientes al k-clique. El **dominio** son los posibles vértices que puede ir asociado al nodo. Las **restricciones** son que cada vértice que se escoge del dominio tiene que estar conectado a todos los otros nodos que se han escogido anteriormente.

Ejemplo en pseudocódigo (para guía del alumno; no era necesario implementar):

```
X = nodos 1, …, k
D = V
R = {(v, v') ϵ E ∀ v' ϵ C}

is_solvable(X, D, C):
    si X = {} return True
    x ← alguna variable de X
    para v ϵ D:
        si v no cumple R, continuar
        si is_solvable(X - {x}, D - {v}, C ∪ {x}):
            retornar True
        C = C - {x}
    retornar False
```

Luego, podemos llamar a is_solvable({1, …, k}, D, {}).

Propuesta 2:

Las **variables** son los vértices de G. El **dominio** es binario: 1 si el vértice está presente en el k-clique y 0 si no. Las **restricciones** son que el vértice asignado como presente en el k-clique (con valor 1 en la variable) tiene que estar conectados a todos los otros vértices que han sido asignados como presente en el k-clique, y el número de vértices asignados como presentes en el grafo debe ser igual a k.

Ejemplo en pseudocódigo (para guía del alumno; no era necesario implementar):

```
X = V
D = {0, 1}
R = {(v, v') ϵ E ∀ v' ϵ C}

is_solvable(X, D, C, k):
    si k = 0 return True
    si X = {} return False
    x ← alguna variable de X
    para v ϵ D:
        si v = 1:
```

```
                si v no cumple R, continuar
                si is_solvable(X - {x}, D, C ∪ {x} , k  - 1):
                        retornar True
                C = C - {x}
        si v = 0:
                si is_solvable(X - {x}, D, C, k):
                        retornar True

    retornar False
```

Luego, podemos llamar a is_solvable({V, D, {}, k).

- **[1 pt]** Explica el conjunto de variables y su dominio.
- **[1 pt]** Explica las restricciones sobre los valores que pueden tomar dichas variables.

b) [2 pts.] Propón una **poda** y explica la modelación a nivel de código requerida para implementarla eficientemente.

R: Si el alumno menciona una poda acorde a su modelación que sea efectiva, y explica cómo implementarla, tendrá el puntaje correspondiente. Algunas podas posibles:

- Si el vértice que estamos revisando tiene un número de aristas tiene menos de k - 1 aristas que se conectan con él, es imposible que éste pertenezca al k-clique. Para implementarla, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta acceder a esta variable y si es menor a k - 1, se detiene la ejecución.
- (Propuesta 2) Si quedan menos variables a asignar, que el valor de k, podemos terminar esa ejecución ya que es imposible agregar suficientes elementos a C para que pertenezcan al k-clique. Para esto, podemos llevar un contador de cuántas variables nos quedan por asignar, y cuántos elementos llevamos en nuestro k-clique. Si la cantidad de variables que nos queda por asignar es menor a (k - (N° elementos asignados que llevamos en k-clique)), detenemos la ejecución.

- **[1 pt] Propone** una poda que efectivamente sea útil para el problema a resolver
- **[1 pt] Explica** cómo implementarla eficientemente

c) [2 pts.] Propón una **heurística para el orden de las variables** y explica la modelación a nivel de código requerida para implementarla eficientemente.

R: Si el alumno menciona una heurística acorde a su modelación que sea efectiva, y explica cómo implementarla, tendrá el puntaje correspondiente. Algunas heurísticas posibles:

- (Propuesta 2) Podemos ordenar los vértices de mayor a menor en función de la cantidad de aristas que poseen. De esta manera, es más probable que éstos sean parte de algún k-clique. Para esto, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta con ordenar este arreglo de vértices en función de la cantidad de aristas hacia otros nodos, mediante algún algoritmo eficiente de los que se han visto en clases (como MergeSort, QuickSort, etc.)

- (Propuesta 1) Se puede ordenar el dominio en función de la cantidad de aristas que tienen las variables. De esta manera, es más probable que al asignar a la variable su valor, éste pertenezca al k-clique. El procedimiento es similar a la heurística anterior: podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta con ordenar este arreglo de vértices en función de la cantidad de aristas hacia otros nodos, mediante algún algoritmo eficiente de los que se han visto en clases (como MergeSort, QuickSort, etc.), y utilizar esto como el dominio de la función de backtracking.

- **[1 pt] Propone** una heurística que efectivamente sea útil para el problema a resolver
- **[1 pt] Explica** cómo implementarla eficientemente

Estructuras de Datos y Algoritmos - IIC2133
Control 5

1) Se tiene un *stream* s = d1 d2 d3 … de largo indefinido donde cada dato d = (u,v) representa una arista no direccional que se agrega a un grafo G inicialmente vacío. La idea es que G permanezca acíclico, por lo que si la arista a agregar forma un ciclo se detiene la lectura del stream y se retorna G. **Explica en detalle** cómo llevar a cabo este proceso, determinando de manera eficiente en cada paso si la arista a agregar forma un ciclo en G. **Cuidado**: no conoces todos los vértices por adelantado.

Propuesta de solución:

Mencionar que se puede resolver mediante una implementación de un algoritmo similar a Kruskal, mediante conjuntos disjuntos. En este caso, no es necesario ordenar las aristas (dado que llegan a través del stream).

Para cada elemento del stream d = (u,v), se revisa si ya existen. Debo revisar en una tabla de hash/arreglo dinámico de manera que la complejidad de determinar si los vértices fueron o no explorados anteriormente sea O(1) **[1 punto]**.

Si alguno no existe, se realiza makeset(u) y/o makeset(v) y se indica que estos son su propios representantes. Se agregan a la tabla de hash/arreglo dinámico **[1 punto]**.

Se hace find(u) y find(v), identificando a sus representantes **[1 punto]**.

Si tienen el mismo representante, significa que pertenecen al mismo conjunto y no los uno
Justificar su correctitud. Si dos vértices tienen el mismo representantes, implica necesariamente que pertenecen al mismo subconjunto (fueron unidos en algún momento por una arista del stream). De esta manera, el algoritmo debe finalizar su ejecución sin incorporar la arista que genera el ciclo. En caso de que no tengan el mismo representante, implica que no pertenecen al mismo subconjunto del grafo G, lo que implica que unirlos no generaría un ciclo **[2 puntos]**.

Si tienen diferente representante, los uno con Union(u,v) **[1 punto]**.

2)

a) El primer for toma un tiempo O(V). La línea 6 consiste en inicializar el min heap binario con todos los vértices, esto tiene complejidad de O(V) igualmente. **[0.5 puntos]**

En el loop while se recorren todos los vértices y en cada uno de ellos hay que extraer el menor, complejidad O(1), y reordenar el heap, complejidad O(logV), es decir, tiene complejidad O(VlogV). **[1.5 puntos]**

Para el caso del for dentro del while se recorrerán en total todas las aristas, y se puede tener que ordenar el heap para cada caso, por lo que tiene complejidad O(ElogV). **[1.5 puntos]**

La complejidad total sería O((E + V)logV). **[0.5 puntos]**

b) Cualquier ejemplo que demuestre lo dicho en el enunciado y que sea correcto **[2 puntos]**