Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Computación
IIC2133 – Estructuras de datos y algoritmos
Primer Semestre 2020

Pauta Interrogación 3

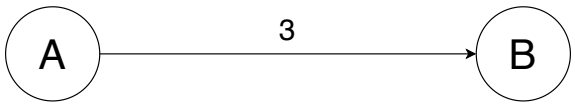
Problema 1

Parte A

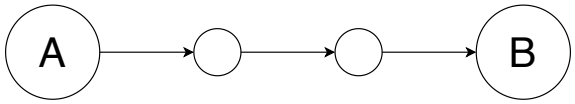
Este algoritmo usa BFS para buscar la ruta más barata de s a f. El problema es que BFS no ve los costos de las aristas, sino que busca la ruta más barata en términos de **cantidad de aristas**. Esto significa que para una arista cualquiera con costo w, BFS la ve como una arista de costo 1.

[1 pts] Por mencionar esta propiedad de BFS.

Lo que hace el for al principio del algoritmo es eliminar las aristas con costo $w > 1$ y reconectar los nodos mediante w aristas de manera que BFS vea el costo de ir entre ese par de nodos como w (aristas) y no 1.



Distancia según BFS: 1



Distancia según BFS: 3

[1 pts] Por mencionar como el algoritmo transforma el grafo para que BFS pueda funcionar.

La pregunta pedía justificar. Si demuestran también se considera correcto.

Parte B

En primer lugar, la transformación recorre todas las aristas, por lo que tenemos E pasos.

Para cada arista de costo $w > 1$, se crean $w - 1$ vértices y w aristas, lo que son $2w - 1$ pasos.

Definimos W como la suma de los costos de todas las aristas del grafo.
La transformación va a incurrir en $\mathcal{O}(W)$ pasos para crear todos los nodos y aristas auxiliares.
[1pts] por calcular correctamente la complejidad de la transformación o primera mitad.

(Es igualmente válido reemplazar W por $|E| * W$ si W se define como el promedio de los costos.)
(Es posible que algunos consideren que esto toma $\mathcal{O}(1)$. Después W aparece en la complejidad de BFS así que

no afecta resultado final, pero primera mitad no obtiene puntaje.)

Luego de la transformación, el grafo va a tener $V' = V + W - E = \text{V} + \text{O(W)}$ vértices, y $E' = W$ aristas.

Por lo tanto, BFS se va a demorar $\mathcal{O}(V' + E') = \mathcal{O}(V + W + W) = \mathcal{O}(V + W)$

Por lo que el algoritmo en total va a demorar:

$$E + \mathcal{O}(W) + \mathcal{O}(V + W + W)$$

$$\mathcal{O}(V + W + W)$$

[No es necesario pero correcto mencionar que $E \in \mathcal{O}(W)$, por lo que la complejidad queda $\mathcal{O}(V + W)$]

[1pts] por reemplazar correctamente V' y E' y expresar la complejidad final.

Parte C

Si queremos obtener el costo de la ruta más corta entre s y f, es llegar y ejecutar Dijkstra sin modificar el grafo.
[1 pts] Por explicar como usar Dijkstra para este problema.

La complejidad de Dijkstra (usando un min-heap) es

$$\mathcal{O}((V + E) \cdot log(V))$$

Convendrá usar Dijkstra cuando tome menos tiempo que SPW, es decir:

$$(V + E) \cdot log(V) < V + E + W$$

Despejando W, tenemos que nos va a convenir usar Dijkstra cuando

$$W > (V + E) \cdot log(V) - V - E$$

O si simplificaron la E,

$$W > (V + E) \cdot log(V) - V$$

Esto se puede interpretar como que conviene usar Dijkstra cuando el costo de usar la Open es menor que el costo de transformar el grafo y realizar el BFS en SPW. Esto se traduce en cuando los números de vértices son bajos y, sobre todo, cuando los costos totales o promedio son altos.
[1pts] Por llegar a condición a partir de comparación planteada y explicar cuándo conviene Dijkstra

Problema 2

Este problema se parece mucho a los que estudiamos en clases cuando vimos orden topológico. Si queremos aplicar algoritmos de grafos para este problema, debemos primero convertirlo en un grafo.

Definimos el grafo $G(V, E)$ de la malla de la siguiente forma:

- ◇ Cada vértice $v \in V$ corresponde a un curso distinto.
- ◇ Si el curso a tiene como requisito el curso b, entonces existe una arista $(a, b) \in E$

[1 pts] Por definir correctamente el grafo.

Esto se parece mucho a los ejemplos de orden topológico vistos en clases. El problema es que el algoritmo de orden topológico **no tiene noción de tareas que pueden realizarse simultáneamente**, por lo que no podemos usarlo directamente.

[1 pts] Por mencionar este problema con orden topológico

Lo único que sabemos es que los nodos a los que no les llega ninguna arista se pueden tomar en el primer semestre, pero a partir de ahí no es trivial como procedemos.

Opción 1: Algoritmo Cuadrático $\mathcal{O}(V^2)$ [1 pts]

Una vez que tenemos los nodos del primer semestre, podemos eliminarlos a ellos y a las aristas que salen de ellos, y los nodos a los que no les lleguen aristas serán los nodos del 2do semestre. Repitiendo esta estrategia para cada semestre:

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    S ← lista de listas, vacía
    while V ≠ ∅:
        x ← lista vacía
        for v ∈ V:
            if v.incoming = 0:
                Insertar v al final de x
                Eliminar v de V
        for x ∈ X:
            foreach arista (x, v) que sale de x:
                v.incoming ← v.incoming − 1
        Agregar x al final de S
    return S
```

Opción 2: Algoritmo *Linearítmico* $\mathcal{O}(V \cdot log(V))$ [2.5 pts]

La misma estrategia anterior, pero usando un heap para que sea más eficiente

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    Convertir V en un min − heap, ordenando por v.incoming
    S ← lista de listas, vacía
    while V ≠ ∅:
        x ← lista vacía
        while V ≠ ∅ ∧ V.root.incoming = 0:
            Extraer v de V
            Insertar v al final de x
        for x ∈ X:
            foreach arista (x, v) que sale de x:
                v.incoming ← v.incoming − 1
        Agregar x al final de S
    return S
```

Opción 3: Algoritmo Lineal $\mathcal{O}(V + E)$ [4 pts]

Recorrer el grafo con DFS/BFS a partir de los cursos del primer semestre, entrando a un curso (vértice) sólo una vez que ya se entró a **todos** sus requisitos (ancestros). El semestre de un curso será 1 más que el semestre de su requisito con el mayor semestre.

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
        v.semestre ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    Open ← lista vacía
    for v ∈ V:
        if v.incoming = 0:
            Agregar v a Open
            v.semestre ← 1
    while Open ≠ ∅:
        Extraer un vértice x cualquiera de Open
        foreach arista (x, v) que sale de x:
            v.incoming ← v.incoming − 1
            if v.incoming = 0:
                Insertar v en Open
            if x.semestre + 1 > v.semestre:
                v.semestre ← x.semestre + 1
```

Al terminar el algoritmo, cada vértice tiene asociado su semestre. Podemos recorrer los vértices una última vez para formar la lista con los elementos de cada semestre.

Problema 3

Podemos separar las aristas E' en dos grupos:

1. Las aristas que están **dentro** de una CFC, E'_o
2. Las aristas que están **fuera** de todas las CFC, E'_x

En el primer caso, tenemos que por cada CFC G con $n > 1$ nodos, E'_o tiene n aristas, ya que es necesario preservar **un** solo ciclo que pase por todos los nodos de la CFC para que siga siendo una CFC en G'. Es decir, si definimos

$$C = \{c \mid c \text{ es una CFC de } G\}$$

Entonces

$$|E'_o| = \sum_{c \in C,\ |c| > 1} |c|$$

[2 pts] Por identificar cuantas aristas aportan las CFC; si no menciona el caso borde de las CFCs de un nodo, 1pt. (1).

En el segundo caso, tenemos que las aristas que no están dentro de ninguna CFC son las aristas que van de una CFC a otra. En G' no pueden haber múltiples aristas que conecten un mismo par de CFCs, ya que en el grafo de componentes se reducen a 1, por lo que sobrarían aristas.

Es decir, si $H(C, F)$ es el grafo de componentes de G,

$$|E'_x| = |F|$$

[2 pts] por identificar cuantas aristas quedan fuera de las CFC; si asume grafo conexo, 1pt. (2).

Teniendo esto, podemos calcular $|E'|$ sin necesidad de computar un G'. Pero sí va a ser necesario calcular el grafo de componentes. El algoritmo es como sigue:

Primero, para identificar las CFCs, usamos el algoritmo de Kosaraju como se vió en clases, pero le hacemos una pequeña modificación a la función que asigna representantes para calcular el tamaño de la CFC:

```
assign(u, rep):
    total ← 0
    if u.rep = ∅:
        u.rep ← rep
        foreach v in α[u]:
            total ← total + assign(v, rep)
    return total
```

Teniendo eso podemos calcular $|E'_o|$ sumando la cantidad de nodos de cada CFC con más de un nodo, mientras que para calcular $|E'_x|$ simplemente computamos F:

```
|E'|(G(V, E)):
    |E'_o| ← 0
    Crear lista L vacía
    Ejecutar dfs(G) con tiempos
    Insertar vértices en L en orden decreciente de tiempos f
    for each u in L:
        n ← assign(u, u)
        if n > 1:
            |E'_o| ← |E'_o| + n
    F ← ∅
    for each (u, v) ∈ E:
        if u.rep ≠ v.rep:
            if (u.rep, v.rep) ∉ F:
                Agregar (u.rep, v.rep) a F
    return |E'_o| + |F|
```

Y una vez obtenido $|E'|$, podemos fácilmente retornar $\Delta E = |E| - |E'|$

Este algoritmo es $\mathcal{O}(V + E)$, ya que Kosaraju es $\mathcal{O}(V + E)$ y computar F es $\mathcal{O}(E)$

[4 pts] Por algoritmo correcto (Correctitud)

[2 pts] por que sea eficiente, siempre y cuando sea correcto (Eficiencia)

El puntaje de la pregunta será `max((1)+(2),(Correctitud)) + (Eficiencia)`.