UNIVERSIDAD DE CONCEPCION
FACULTAD DE CIENCIAS
FISICAS Y MATEMATICAS
DEPARTAMENTO DE INGENIERIA MATEMATICA

OPTIMIZACIÓN III (525551)

Tarea 1

(Fecha de entrega: 27 de abril de 2020 antes de las 12:00 HRs.)

1. Determine si las siguientes proposiciones son verdaderas o falsas. Justifique su respuesta.

   a) $2^{2^{n+1}} \notin \Theta(2^{2^n})$.

   b) $n^{\log n} = O(n!)$.

   c) $2n^2 + O(1) = \Omega(n\log(n))$.

   d) $f(n) = O(g(n)) \Longrightarrow 2^{f(n)} = O(2^{g(n)})$.

   Soln:

   a) (Verdadero) Como

   $$\lim_n \frac{2^{2^{n+1}}}{2^{2^n}} = \lim_n \frac{2^{2^n\cdot 2}}{2^{2^n}} = \lim_n \frac{(2^{2^n})^2}{2^{2^n}} = \lim_n 2^{2^n} = \infty,$$

   entonces, por resultado visto en clase, $2^{2^{n+1}} \notin O(2^{2^n})$, y así $2^{2^{n+1}} \notin \Theta(2^{2^n})$.

   b) (Verdadero) Notar que $\lim_n \frac{\log(n)}{\sqrt{n}} = 0$. Luego,

   $$\forall c > 0, \exists n_0 \in \mathbb{N}, \log(n) \leq c\sqrt{n}.$$

   Así, para $c = 1$ existe $n_0 (= 16) \in \mathbb{N}$ tal que $\log(n) \leq \sqrt{n}$. De aquí, $\forall n \geq n_0$:

   $$n^{\log n} = (2^{\log(n)})^{\log(n)} \leq (2^{\sqrt{n}})^{\sqrt{n}} = 2^n.$$

   Por otro lado, por resultado visto en clase, se tiene que: $\forall n \geq 2,\ 2^n \leq 2n!$. Por lo tanto, $\forall n \geq \text{máx}\{n_0, 2\} (= n_0)$ se tiene que:

   $$n^{\log n} \leq 2^n \leq 2n!.$$

   En conclusión, $n^{\log n} = O(n!)$.

   c) (Verdadero) Sea $f : \mathbb{N} \to \mathbb{R}^+$ una función tal que $f(n) = O(1)$, i.e. $\exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0 :\ 0 < f(n) \leq c$. Por otro lado, como $\forall n \in \mathbb{N},\ n \geq \log(n)$, entonces $\forall n \in \mathbb{N}, n^2 \geq n\log(n)$. Así,

   $$\forall n \in \mathbb{N},\ 2n^2 + f(n) \geq 2n^2 \geq 2(n\log(n)) \Longrightarrow 2n^2 + f(n) = \Omega(n\log(n)).$$

   Como el resultado anterior es válido para toda $f(n) = O(1)$, entonces $2n^2 + O(1) = \Omega(n\log(n))$.

d) (Falso) Sea $f(n) = 2n$ y $g(n) = n$. Como $\forall n \in \mathbb{N},\ f(n) = 2n \leq 2n = 2g(n)$, entonces $f(n) = O(g(n))$. Sin embargo,

$$\lim_n \frac{2^{f(n)}}{2^{g(n)}} = \lim_n \frac{2^{2n}}{2^n} = \lim_n 2^n = \infty \Longrightarrow 2^{f(n)} \neq O(2^{g(n)}).$$

(Recordar que por resultado visto en clase si $\lim_n \frac{\alpha(n)}{\beta(n)} = \infty$, entonces $\alpha(n) \neq O(\beta(n)) \ \wedge \ \alpha(n) = \Omega(\beta(n)))$.

2. Sea $G = (V, E)$ un grafo no dirigido. Recordar que G se dice bipartito si existe una partición $\{V^1, V^2\}$ de V tal que $E \subseteq \{\{u, v\} : \ u \in V^1 \ \wedge \ v \in V^2\}$.

   a) Explique por qué un algoritmo que examine todas las posibles particiones de V y chequee que las aristas tienen sus extremos en conjuntos distintos de la partición no puede ser polinomial.

   b) Use el algoritmo BFS para construir un algoritmo polinomial que resuelva el siguiente problema:

   BIPARTITE GRAPH: Dado $G = (V, E)$ un grafo no dirigido ¿Es G un grafo bipartito?

   c) Use el algoritmo en *b*) para construir un algoritmo polinomial que resuelva el siguiente problema en grafos dirigidos:

   ODD CYCLE: Dado $G = (V, A)$ un grafo dirigido ¿Existe un ciclo de largo impar en G?

Para los algoritmos construídos justifique su buen funcionamiento y tiempo de ejecucin.
Soln:

a) **(6 ptos.)** Notar que el nmero de particiones no triviales (con repetición) del conjunto $\{1, \ldots, n\}$ en dos subconjuntos no vacíos está dado por:

$$\sum_{k=1}^{n-1} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} - 2 = 2^n - 2.$$

De aquí, el número de particiones no triviales y sin repetición a evaluar es $2^{n-1} - 1$. Como para todo $k \in \mathbb{N}$:

$$\lim_{n \to \infty} \frac{2^{n-1} - 1}{n^k} = \infty,$$

entonces, por resultado visto en clase, $2^{n-1} - 1 \notin O(n^k)$ y por lo tanto cualquier algoritmo que examine todas las particiones (sin repetición) no sería polinomial.

b) **(8 ptos.)** El siguiente algoritmo construye para cada componente conexa de G_i de un grafo no dirigido G una partición de los vértices en dos conjuntos: uno con distancias impar a un vértice dado s y el otro su complemento. Posteriormente chequea si cada conjunto de la partición es un conjunto independiente de vértices (i.e. no hay arista entre ellos). Si esto es así para cada compoenten conexa, entonces el grafo será bipartito y no lo será en caso contrario.

Algorithm BipartiteGraph(G)

Require: $G = (V, E)$ un grafo

```
1:  G_1, ..., G_K ← Connected(G)
2:  i ← 1
3:  while i ≤ k do
4:     Seleccionar s ∈ V(G_i)
5:     (d, π) ← BFS(G_i, s)
6:     for all u, v ∈ V(G_i), u ≠ v do
7:        if d[u] = d[v] mód 2 ∧ {u, v} ∈ E then
8:           return n
9:        end if
10:    end for
11:    i ← i + 1
12: end while
13: return s
```

Donde Connected es cualquier algoritmo que recibe un grafo G no (dirigido) y determina en tiempo $O(|V|^3)$ (como el algoritmo Sconnected) sus componentes conexas. Recordar que un grafo es bipartito si y sólo si cada una de sus componentes es bipartita. Además, un grafo conexo es bipartito si y sólo si no tiene ciclos de largo impar.

El algoritmo anterior en la línea 7 determina si existe una arista conectando dos vértices distintos u, v a distancia par (o impar) de un vértice s. Si no existe tal arista en ningún caso entonces quiere decir que los conjuntos $V^1 = \{u \in: \ d[u] \text{ es par}\}$ y $V^2 = \{u \in: \ d[u] \text{ es impar}\}$ son independientes en cada G_i y por ende cada componente G_i es bipartita, y así G es bipartito. Por el contrario, si $\exists i \in \{1, \ldots, k\},\ u, v \in V(G_i),\ u \neq v$ tal que $d[u] = d[v] \mod 2$ y $\{u, v\} \in E$ entonces existe en G_i caminos p_{su} y $p_{s,v} (= p_{v,s})$ de largo ambos pares o impares. Así el circuito $C : p_{su}, v, p_{v,s}$ es de largo impar y por resultado explicado en clase este contiene entonces un ciclo de largo impar, lo que implica que G_i no es bipartito y por lo tanto G tampoco. Por lo tanto, el algoritmo BipartiteGraph entrega solución al problema planteado.

El tiempo de ejecución del algoritmo está dado por el tiempo del algoritmo Connected, igual a $O(|V^3)$ más el tiempo del ciclo While. Es fácil ver que el número de iteraciones en el peor caso del ciclo While es $O(|V|)$ y que en cada iteración se ejecuta BFS que es $O(|E(G_i) + |V(G_i)|)$ que es a su vez $O(|V(G)|^2)$. Además, el número de operaciones elementales del ciclo for es $O(|V(G_i)|^2)$. Así el número total de operaciones elementales del ciclo While en el peor caso es $O(|V|^3)$. Por lo tanto, el algoritmo es polinomial en el tamaño de su entrada.

c) **(6 ptos.)** Por resultado explicado en clase, se tiene que si $G = (V, A)$ es un digrafo fuertemente conexo, entonces G tiene un ciclo de largo impar si y sólo si su grafo fundamental $G^F = (V, E)$ tiene un ciclo de largo impar, donde

$$\forall u, v \in V,\ u \neq v,\ \{u, v\} \in E \iff (u, v) \in A \ \vee (v, u) \in A.$$

Por lo tanto, se propone el siguiente algoritmo para resolver el problema ODD CYCLE.

Algorithm OddCycle(G)

Require: $G = (V, E)$ un grafo dirigido

```
G_1, ..., G_K ← SConnected(G)
i ← 1
while i ≤ k do
    G_i ← G_i^F
    if BipartiteGraph(G_i^F)==n then
        return s
    end if
    i ← i + 1
end while
return n
```

Donde SConnected es cualquier algoritmo que recibe un grafo G dirigido y determina en tiempo $O(|V|^3)$, como el algoritmo visto en clases, sus componentes fuertemente conexas.

De esta forma, si G tiene un ciclo dirigido de largo impar, entonces este ciclo estará en una componente fuertemente conexa de G. Asi, por resultado explicado previamente, para encontrar dicho ciclo es necesario y suficiente chequear si en cada componente G_i de G su grafo fundamental asociado G_i^F tiene un ciclo (no dirigido), lo que es equivalente a que G_i^F no sea bipartito.

Por último, un razonamiento similar al hecho en la parte b) permite concluir que el tiempo de ejecución de OddCycle es $O(|V|^4)$ y por lo tanto es un algoritmo polinomial.

3. Sea $G = (V, A)$ un digrafo. Se dice que G tiene un **ordenamiento topológico** de sus vértices si ellos pueden ser ordenados en una secuencia $v_1, v_2, \ldots, v_n$, donde $V = \{v_1, \ldots, v_n\}$ y tal que si $(v_i, v_j) \in A$, entonces $i < j$.

   a) Pruebe que G tiene un ordenamiento topológico de sus vértices si y sólo si G es acíclico. (Ind: Si G es acíclico, entonces $\exists j \in V,\ d^-(j) = 0$).

   b) Construya un algoritmo polinomial que reciba como entrada un digrafo G y retorne un orden topológico de sus vértices cuando exista y retorne *no existe* en caso contrario.

   c) Programe en Matlab o Python el algoritmo definido en b) que recibe como entrada la matriz de adyacencia de un digrafo $G = (V, A)$ cualquiera con $V = \{1, \ldots, n\}$.

Soln:

a) **(2 ptos.)** ($\Longrightarrow$) Supongamos que $v_1, v_2, \ldots, v_n$ es un ordenamiento topológico de los vértices de G. Sea $C : v_{i_1}, \ldots, v_{i_k}, v_{i_1}$ un ciclo de G. Luego, como $\forall j = 1, \ldots, k, (v_{i_j}, v_{i_{j+1}}) \in A$, con $v_{i_{k+1}} = v_{i_1}$, entonces se tiene que:

$$i_1 < i_2 < \cdots < i_k < i_1,$$

lo cual es una contradicción.

($\Longleftarrow$) Para probar esta implicancia usaremos la indicación dada, que es un resultado bien conocido de digrafos:

Proposición: *Sea $G = (V, A)$ un digrafo. Si $\forall v \in V,\ d^-(v) \geq 1$ ó $\forall v \in V,\ d^+(v) \geq 1$, entonces G tiene un ciclo.*

Luego, si G es acíclico, entonces $\exists v_1 \in V$ tal que $d^-_G(v_1) = 0$. Definamos el digrafo $G_2 := G - v_1$. Como, G_2 es también acíclico (pues es subgrafo de G) entonces $\exists v_2 \in V - \{v_1\}$ tal que $d^-_{G_2}(v_2) = 0$. De esta forma, podemos construir de manera recursiva una secuencia de los vértices de $V : v_1, v_2, \ldots 1, v_n$ tal que $\forall j = 2, \ldots, n,\ d^-_{G_j}(v_j) = 0$ con $G_j := G_{j-1} - v_{j-1} = G - \{v_1, \ldots, v_{j-1}\}$ donde $G_1 := G$. Mostremos que $v_1, v_2, \ldots, v_n$ es un orden topológico de G. En efecto si $(v_i, v_j) \in A$, entonces $v_i \notin V(G_j)$, pues como $v_j \in V(G_j)$, entonces en caso contrario $d^-_{G_j}(v_j) \geq 1$, lo cual es una contradicción con $d^-_{G_j}(v_j) = 0$. Por lo tanto, como $v_i \notin V(G_j)$, entonces $i < j$.

b) **(8 ptos.)** El siguiente algoritmo ToplogicalOrder está basado en el resultado anterior. Se usa el algoritmo Acyclic visto en prática que determina si un digrafo es acíclico o no. El tiempo de ejecución está dado por el algoritmo Acyclic que ejecuta $O(|V|^3)$ operaciones elementales y por el ciclo while que ejecuta $O(|V|)$ iteraciones en el peor caso. Determinar si $d^-_G(v) = 0$ y calcular $G - v$ pueden ser hechos en $O(|V|)$ operaciones elementales en el peor caso. El resto de las operaciones son $O(1)$. Luego, el tiempo de ejecución de TopologicalOrder es $O(|V|^3) + O(|V|) \cdot O(|V|)$ lo que es $O(|V|^3)$, ie el algoritmo es polinomial.

Algorithm TopologicalOrder(G)

Require: $G = (V, A)$ un digrafo con $|V| = n$
1: $T \leftarrow Null$, $i \leftarrow 1$
2: **while** $i \leq n$ **do**
3: **if** $\nexists v \in V,\ d^-_G(v) = 0$ **then**
4: **return** No existe
5: **end if**
6: Seleccionar $v \in V$ tal que $d^-_G(v) = 0$.
7: $T[i] \leftarrow v$
8: $G \leftarrow G - \{v\}$
9: $i \leftarrow i + 1$
10: **end while**
11: **return** T