IIC2133 – Estructuras de Datos y Algoritmos

Interrogación 2

Hora inicio: 14:00 del 16 de noviembre del 2020

Hora máxima de entrega: 23:59 del 16 de noviembre del 2020

0. Responde esta pregunta en papel y lápiz, incluyendo tu firma al final. Nos reservamos el derecho a no corregir tu prueba según tu respuesta a este ítem. Puedes adjuntar esta pregunta a cualquiera de las preguntas que subas a SIDING.

    a. ¿Cuál es tu nombre completo?
    b. ¿Te comprometes a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta?

Responde sólo 3 de las 4 preguntas a continuación. Si respondes las 4, se escogerá arbitrariamente cuales 3 corregir, y la otra se considerará como no entregada.

Si una pregunta pide que hagas algo en una complejidad específica, debes justificar por qué tu respuesta tiene esa complejidad. Si no, tendrás como máximo 1/3 del puntaje correspondiente.

Deberás citar tus fuentes. Si detectamos plagio tendrás un 1 en la pregunta correspondiente.

1. Un grafo no dirigido $\boldsymbol{G(V, E)}$ se dice $\boldsymbol{k}$-coloreable ($\boldsymbol{k} \in \mathbb{N}$), si existe una manera de asignar a cada vértice $\boldsymbol{v} \in \boldsymbol{V}$ un color $\boldsymbol{c} \in \{\boldsymbol{1}, \boldsymbol{2}, \cdots, \boldsymbol{k}\}$, tal que para todo $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$ se cumple que $\boldsymbol{u.color} \neq \boldsymbol{v.color}$
   Considera el siguiente algoritmo que determina si un grafo es $\boldsymbol{k}$-coloreable, considerando que inicialmente $\boldsymbol{v.color} = \boldsymbol{0}$ para todo vértice $\boldsymbol{v} \in \boldsymbol{V}$, y que $\boldsymbol{\alpha}$ son las listas de adyacencia del grafo.

```
is_k_coloreable(V, α, k):
    if V = ∅:
        return true
    Sea v un vértice cualquiera de V
    for c ← 1..k:
        valid ← true
        for each u ∈ α[v]:
            if u.color = c:
                valid ← false
                break
        if valid:
            v.color ← c
            if is_k_coloreable(V − {v}, α , k):
                return true
            v.color ← 0
    return false
```

   Llamamos ***coloración parcial*** al subconjunto $\boldsymbol{U} \subseteq \boldsymbol{V}$ de todos los vértices que tienen asignado un color. Luego de cada asignación, el algoritmo ha generado una nueva ***coloración parcial***.

   Justifica por qué el algoritmo nunca va a generar dos veces una misma ***coloración parcial*** con los mismos colores.

2. Kojima-san es un desarrollador que quiere crear su propio videojuego. Sin embargo, para realizar esta tarea primero necesita instalar un *engine* y todas sus librerías. Cada librería puede depender a su vez de otras librerías, a las que llamaremos dependencias. Una librería **no puede** ser instalada a menos que **todas** sus dependencias ya se encuentren instaladas. Kojima-san podría intentar instalarlas manualmente, sin embargo, él sabe que le tomaría una eternidad. Examinando los requisitos de instalación se percata que existe un total de $\boldsymbol{L}$ librerías, donde cada librería tiene a lo más $\boldsymbol{D}$ dependencias. Considera que el *engine* en si es una librería con sus propias dependencias.

   Definimos un grafo $\boldsymbol{G(V, E)}$ donde cada vértice $\boldsymbol{v} \in \boldsymbol{V}$ corresponde a una librería, si $\boldsymbol{u}$ depende de $\boldsymbol{v}$, entonces hay una arista $(\boldsymbol{v}, \boldsymbol{u}) \in \boldsymbol{E}$.

   a) Dado $\boldsymbol{G(V, E)}$, describe un algoritmo $\boldsymbol{\mathcal{O}(L + LD)}$ que entregue un orden de instalación de todas las librerías.
   b) Kojima-san se percata de que ya tiene algunas de las librerías instaladas en su computador, sólo quedando $\boldsymbol{R}$ por instalar. Dado $\boldsymbol{G(V, E)}$, describe un algoritmo $\boldsymbol{\mathcal{O}(R + RD)}$ que entregue un orden de instalación de las librerías que faltan. Asume que detectar si una librería ya está instalada es $\boldsymbol{\mathcal{O}(1)}$.

3. Dado un grafo $\boldsymbol{G(V, E)}$, dirigido y costos $\boldsymbol{w(u, v)} \in \mathbb{R}$, el algoritmo de ***Dijkstra*** busca las rutas más cortas desde un vértice $\boldsymbol{s} \in \boldsymbol{V}$ a cada otro vértice del grafo. El algoritmo opera bajo dos supuestos:

   a) El grafo no contiene aristas de costo negativo

   Para un grafo $\boldsymbol{G(V, E)}$ con aristas de costo negativo, podemos modificarlo para dejar todas las aristas con costo no negativo. Para esto tomamos el costo de la arista de menor costo en $\boldsymbol{G(V, E)}$, y se lo restamos a los costos de todas las demás aristas. Así, esta arista queda con costo $\boldsymbol{0}$ y las demás con costo positivo.

   Demuestra mediante un ejemplo que, si hacemos este cambio, las rutas encontradas por ***Dijkstra*** no necesariamente son rutas más cortas en el grafo original.

   b) El costo de una ruta está definido como la <u>suma</u> de los costos de cada arista en la ruta

   ¿Qué pasa si definimos el costo de la ruta entre dos nodos como la **multiplicación** de los costos de cada arista en la ruta?

   Podemos modificar cómo el algoritmo de ***Dijkstra*** calcula la distancia $\boldsymbol{d}$ de un vértice:

   Con suma, se define como $\boldsymbol{d(v) = d(u) + w(u, v)}$, con $\boldsymbol{d(s) = 0}$
   Con multiplicación, se define como $\boldsymbol{d(v) = d(u) \cdot w(u, v)}$, con $\boldsymbol{d(s) = 1}$

   Demuestra mediante un ejemplo que, bajo esta definición de $\boldsymbol{d}$, las rutas encontradas por ***Dijkstra*** no necesariamente son las más cortas.

4. Considera la siguiente implementación de una tabla de hash con encadenamiento: la tabla es un arreglo $\boldsymbol{T}$ en que cada casillero $\boldsymbol{T}[\boldsymbol{i}]$ tiene tres campos: dos punteros, y un espacio para guardar lo que queremos guardar en la tabla. En todo momento, todos los casilleros vacíos de la tabla se mantienen en una única lista $\boldsymbol{L}$ doblemente ligada de casilleros disponibles, usando los campos punteros de los casilleros para indicar el anterior y el siguiente.

   Para insertar un elemento con clave $\boldsymbol{k}$ en la tabla, hacemos lo siguiente: miramos el casillero $\boldsymbol{T}[\boldsymbol{h}(\boldsymbol{k})]$; si el casillero está disponible, entonces lo extraemos en $\boldsymbol{\mathcal{O}(1)}$ de la lista $\boldsymbol{L}$ y lo usamos para guardar el elemento; en caso contrario, sacamos el primer casillero disponible de la lista $\boldsymbol{L}$, guardamos allí el elemento, y colocamos este casillero a la cola de una lista simplemente ligada que parte en $\boldsymbol{T}[\boldsymbol{h}(\boldsymbol{k})]$.

   Compara este esquema con el esquema de encadenamiento visto en clases. En particular, compara la complejidad esperada de las operaciones de inserción, búsqueda y eliminación.