Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

Interrogación 2

Pregunta 1

Un grafo no dirigido $G(V, E)$ se dice **k**-coloreable ($\mathbf{k} \in \mathbb{N}$), si existe una manera de asignar a cada vértice $v \in V$, un color $c \in \{1, ..., k\}$ tal que, para todo $(u, v) \in E$ se cumple que $u.color \neq v.color$.

Considera el siguiente algoritmo que determina si un grafo es **k**-coloreable, considerando que inicialmente $v.color = 0$ para todo vértice $v \in V$, y que α 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 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.

Solución Pregunta 1)

Opción 1: Árbol de asignaciones

El algoritmo propuesto es backtracking, ya que busca todas las asignaciones válidas posibles de cada vértice, para asignarles un color de forma recursiva.

La estructura del árbol de asignaciones que genera este algoritmo a partir de los vértices es la siguiente (se consideró una versión simplificada donde se muestran todas las opciones posibles siendo que el algoritmo recorta las que incumplen la condición de **k**-coloreable, por ejemplo, root $\rightarrow 1 \rightarrow 1$ será podado por el algoritmo por tener el mismo color en vértices vecinos)

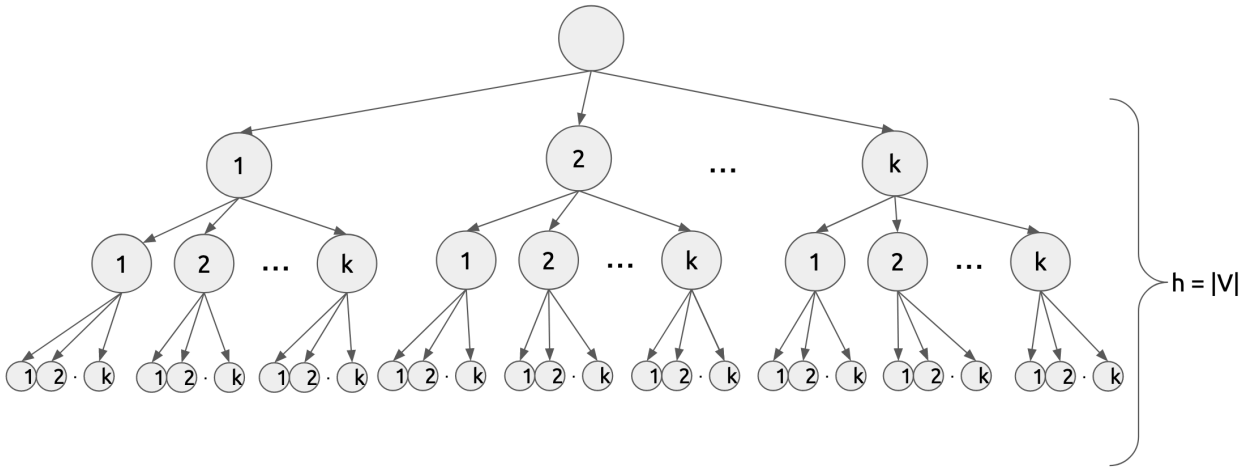

Figura 1: Árbol de asignaciones

Las coloraciones parciales en cada iteración del algoritmo representan un camino por el árbol de asignaciones. Por lo tanto, para que hayan 2 asignaciones parciales iguales en cualquier iteración del algoritmo, esto significaría que existen dos caminos iguales en el árbol de asignaciones, lo que por construcción es imposible. Además, backtracking recorrerá cada camino una sola vez. Debido a estas dos condiciones, cada coloración parcial en cada iteración del algoritmo será diferente.

[**1 pt**] Por mencionar que es un algoritmo de backtracking.

[**2 pts**] Por explicar la estructura del árbol de asignaciones.

[**3 pts**] Por justificar que rutas distintas en el árbol son coloraciones parciales distintas.

Opción 2: Fundamentación por código o casos

Importante: Esta opción no está completa debido a que no se usaron las propiedades de backtracking esperadas o la generalización de que se cumplía para todos los casos no era correctamente justificada, por lo que el puntaje máximo a obtener en este caso es 4 puntos.

El algoritmo is_k_coloreable es un algoritmo de backtracking. Para que dos coloraciones sean iguales, necesariamente su conjunto de vértices debe ser igual. Esto es equivalente a mencionar que dos asignaciones sucesivas son siempre distintas. Para demostrar que dos asignaciones no sucesivas son siempre distintas, se argumenta a través de la lógica o funcionamiento del algoritmo. Por ejemplo, a través del `for` sobre los colores.

[1 pt] Por mencionar que es un algoritmo de backtracking.

[1 pt] Por justificar conjuntos de vértices distintos son coloraciones parciales distintas.

[2/4 pt] Por mostrar que dos asignaciones no sucesivas son coloraciones parciales distintas argumentando a través de la lógica o funcionamiento del algoritmo. Ojo: El puntaje es 2/4 por la incompletitud de esta respuesta mencionada anteriormente.

Pregunta 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 $\boldsymbol{(v, u)} \in \boldsymbol{E}$

(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 $\boldsymbol{\mathcal{O}(1)}$

Solución Pregunta 2a)

Es posible notar que se está solicitando la ordenación topológica del grafo formado por las librerías y sus dependencias. De esta forma, se debe realizar el algoritmo `topsort` visto en clases para conseguir el orden de instalación. Cabe notar que si se añaden los nodos según tiempos descendientes es necesario invertir la lista resultante.

La complejidad de `topsort` es de $\boldsymbol{\mathcal{O}(L + LD)}$ debido a que se poseen $\boldsymbol{L}$ nodos y cada nodo posee a lo más $\boldsymbol{D}$ aristas. Finalmente, invertir la lista resultante de $\boldsymbol{L}$ elementos posee una complejidad de $\boldsymbol{\mathcal{O}(L)}$. De esta forma, la complejidad del algoritmo es de $\boldsymbol{\mathcal{O}(L + LD)}$.

[1,5 pt] Por describir el algoritmo correctamente. Pueden aplicar orden topológico directamente.

[0,5 pt] Por justificar la correctitud del algoritmo.

[1 pt] Por justificar la complejidad, indicando que es $\boldsymbol{\mathcal{O}(L + LD)}$.

En caso de plantear un algoritmo no topológico se debe justificar su correctitud y complejidad de manera explícita.

Si el algoritmo propuesto no es $\boldsymbol{\mathcal{O}(L + LD)}$ entonces el puntaje máximo a obtener es 1pt.

Solución Pregunta 2b)

Cabe notar que no es suficiente usar el algoritmo de `topsort` visto en clases: si pintáramos todos los nodos entonces añadiríamos un $\boldsymbol{O(L)}$ a la complejidad, y el algoritmo sería incorrecto.

En cambio, debemos transponer el grafo. Una vez transpuesto, recorremos el grafo desde el nodo inicial (***engine***) para ir verificando las dependencias instaladas. Notemos que transponer el grafo debe ser a lo

más $\mathcal{O}(R + RD)$, por lo que no cualquier algoritmo nos servirá. En particular, asumiremos que tenemos la matriz de adyacencia del grafo, tal que transponerlo sea constante.

Durante el recorrido, cada vez que encontremos una dependencia instalada se deben eliminar o pintar todas las aristas del grafo que lleven al nodo correspondiente y posteriormente eliminar dicho nodo. También, es posible pintar el nodo, tal que lo saltemos cuando lo veamos nuevamente.

Finalmente, se debe realizar `topsort` con el grafo resultante. En el caso de aplicar el algoritmo visto en clases, es necesario invertir la lista resultante para obtener el orden adecuado.

Respecto a la complejidad, la transposición del grafo representado mediante una matriz de adyacencia es $\mathcal{O}(1)$ como ya mencionamos anteriormente. Respecto al recorrido, este tiene una complejidad asociada de $\mathcal{O}(RD)$ debido a que dependencia tiene a lo más D aristas y se visitan a lo más R nodos. Por otra parte, realizar `topsort` en el grafo resultante es $\mathcal{O}(R + RD)$ e invertir la lista retornada es $\mathcal{O}(R)$, obteniendo así una complejidad final de $\mathcal{O}(R + RD)$.

[**1,5 pt**] Por describir el algoritmo correctamente.

[**0,5 pt**] Por justificar la correctitud del algoritmo.

[**1 pt**] Por justificar la complejidad, indicando que es $\mathcal{O}(R + RD)$.

En caso de plantear un algoritmo no topológico se debe justificar su correctitud y complejidad de manera explícita.

Pregunta 3

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

a. **El grafo no contiene aristas de costo negativo**

   Para un grafo $G(V, E)$, 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 $G(V, E)$, se lo restamos a los costos de todas las aristas. Así, esta arista queda con costo $\mathbf{0}$ 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 suma 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 d de un vértice:

   Con suma, se define como $d(v) = d(u) + w(u, v)$, con $d(s) = 0$

   Con multiplicación, se define como $d(v) = d(u) \cdot w(u, v)$, con $d(s) = 1$

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

Solución Pregunta 3a)

Se debe mostrar un Grafo el cual contenga al menos una de sus aristas con costo negativo.

Por ejemplo para buscar el camino más corto entre A y C.

Tenemos el siguiente Grafo:

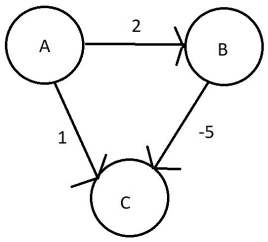

Figura 2: Ejemplo de Grafo con arista negativa

Se debe aplicar la modificación señalada en la prueba, restando el costo de la arista más negativa a **todas** las aristas.

Continuando con el ejemplo, tenemos el siguiente Grafo:

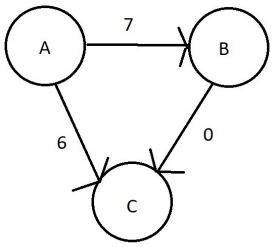

Figura 3: Ejemplo de Grafo aplicando modificación propuesta

Aplicando Dijkstra sobre ambos grafos tenemos que:

1) Sobre el primer Grafo la ruta óptima es **A-B-C**, pero aplicando Dijkstra, este encuentra como óptima la ruta **A-C** ya que *pinta* de negro **C** antes de revisar el nodo **B**.

2) Sobre el segundo Grafo la ruta óptima aplicando Dijkstra es **A-C**. En este caso es la ruta óptima ya que no tenemos costos negativos.

Se demuestra entonces con este ejemplo que haciendo la modificación, la ruta óptima del segundo Grafo no es necesariamente la ruta óptima en el primer Grafo.

Distribución de puntaje:

[**1 pt**] Mostrar ejemplo con arista negativa y realizar la modificación señalada en enunciado.

[**1.5 pt**] Aplicar Dijkstra correctamente sobre grafo con costos positivos.

- [**1 pt**] Si no hay explicación

[**0.5 pt**] Concluir que la modificación señalada en enunciado no implica que necesariamente la ruta óptima del segundo Grafo es ruta óptima en el primer Grafo.

Solución Pregunta 3b)

Se debe mostrar un Grafo adecuado sobre el cual haga sentido aplicar la modificación al algoritmo de Dijkstra planteado en el enunciado.

Por ejemplo para buscar el camino más corto entre A y C.

Tenemos el siguiente Grafo:

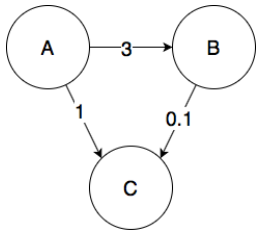

Figura 4: Ejemplo de Grafo con costo total la multiplicación de los costos de cada ruta

Podemos ver que la ruta encontrada con Dijkstra es **A-C** con un costo de 1 dado por la multiplicación entre los costos de las arista **A-C** y el costo inicial de $\mathbf{A = 1}$, es decir, $\mathbf{1 \cdot 1}$. Dijkstra encuentra esta ruta porque *pinta* de negro el nodo C antes de revisar el nodo B, ya que $\boldsymbol{d(c) < d(B)}$.

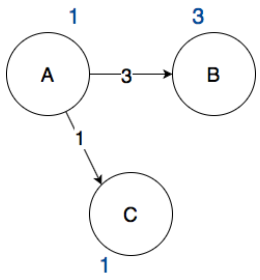

Figura 5: Ruta más corta con dijkstra

Sin embargo, la ruta óptima incluye la arista **B-C** ya que su costo de **0,1** disminuye el costo de la ruta. Por lo que el costo total mínimo para ir de A a C es la ruta **A - B - C** con costo $\mathbf{1 \cdot 3 \cdot 0{,}1 = 0{,}3}$

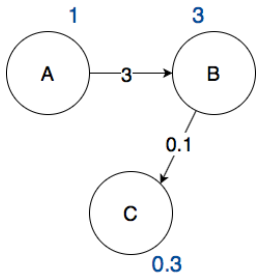

Figura 6: Ruta óptima

Se demuestra entonces con este ejemplo que haciendo la modificación del costo de la ruta, la ruta encontrada con el algoritmo Dijkstra no necesariamente es la ruta óptima en el Grafo.

Distribución de puntaje:

[**1 pt**] Plantear ejemplo donde Dijkstra modificado no encuentra la ruta óptima

[**1.5 pt**] Aplicar correctamente Dijkstra modificado al ejemplo planteado

- [**1 pt**] Si no hay explicación

[**0.5 pt**] Concluir que la ruta encontrada con Dijkstra modificado no es necesariamente la ruta óptima

Pregunta 4

Considera la siguiente implementación de una tabla de hash con encadenamiento: la tabla es un arreglo $\boldsymbol{T}$ en que cada casillero $\boldsymbol{T[i]}$ 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 $\boldsymbol{T[h(k)]}$; si el casillero está disponible, entonces lo extraemos en $\boldsymbol{O(1)}$ 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 $\boldsymbol{T[h(k)]}$. 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.

Solución Pregunta 4)

Por cada una de las operaciones (inserción, búsqueda y eliminación):

- 1 punto por el cálculo correcto de la complejidad (dependiendo de si es caso promedio o peor caso según mencionado por estudiante): O(n) en el peor caso y O(1) en el caso promedio. 0.5 puntos si no se justifica o se justifica parcialmente.

  Elementos que se pueden usar para el cálculo del caso promedio:

  - Al asumir que h(k) es uniforme, se puede hacer un argumento a que en el caso promedio las listas ligadas con elementos de cada uno de los nodos tendrán un número acotado de elementos, con esto se puede argumentar que es O(1).
  - Alternativamente se puede hacer el símil con el esquema encadenamiento visto en clases, y argumentar que la diferencia de operaciones en cada uno de los pasos tiene la misma complejidad asintótica. Con esto, se puede acudir a los resultados de complejidad vistos en clase.

  En el peor caso, también se puede acudir al símil con el esquema de encadenamiento visto en clases, o también explicar cuál es el peor caso, donde todos los elementos resultan tener el mismo h(k), por lo que para insertar un elemento sería O(n), para acceder o eliminar a un elemento el peor caso es cuando el elemento está al final de la lista ligada, en el mismo caso anterior, por lo que también sería O(n) porque habría que recorrer esa lista ligada en su completitud.

- 1 punto por la comparación con encadenamiento. Independiente del caso, el estudiante debe llegar a que la complejidad es igual a la del esquema de encadenamiento visto en clases. En la comparación, el estudiante debe aludir al símil en las operaciones realizadas por cada uno de los esquemas, llegando a la conclusión que tienen la misma complejidad.

Puntaje total de la pregunta: 6 puntos.