Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2133— Estructuras de Datos y Algoritmos 2020-2

FECHA: 16 de Noviembre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Interrogación 1

Pregunta 1

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

Sea una U_1 una asignación parcial, **producida en algún momento de la ejecución del algoritmo**, y sea U_2 otra coloración parcial **producida en un momento distinto**. Por contradicción, supongamos que $U_1 = U_2$, es decir, U_1 y U_2 poseen los mismos nodos, y estos tienen los mismos colores. Notar que dado que estos conjuntos se generan en cada asignación, no pueden ser vacíos, y tienen al menos un elemento.

Sea v_0 el elemento inicial, es decir, el elemento (nodo) desde el cual se comenzó la ejecución del algoritmo. Tenemos que v_0 de U_1 debe ser igual a v_0 de U_2, debido a que ambas ejecuciones partieron de una raíz en común. Además, por el supuesto inicial, se tiene que los colores son iguales.

Notar que según la línea "Sea v un vértice cualquiera de V", los nodos son accedidos de manera arbitraria, por lo que se pueden dar dos casos:

- Los nodos $U_1 - \{v_0\}$ y $U_2 - \{v_0\}$ fueron accedidos en el mismo orden
- Los nodos $U_1 - \{v_0\}$ y $U_2 - \{v_0\}$ fueron accedidos en un orden distinto

En el primer caso, se evidencia la contradicción trivialmente, debido a que tanto U_1 como U_2 son iguales, pero **fueron producidos en el mismo momento de la ejecución del algoritmo**, lo que contradice el supuesto inicial.

Para el segundo caso, **sin pérdida de generalidad**, supongamos que en la ejecución del algoritmo primero se llegó a U_1, y en una iteración posterior se llegó a U_2. Dado que U_1 y U_2 tienen los mismos nodos, esto significa que en algún momento posterior a encontrar U_1, se encontró un "error", es decir, un nodo no podía tomar ningún color válido, y se realizó "backtracking", volviendo a un punto anterior a U_1. Sin embargo, esto implica que se debió haber realizado un cambio de color en algún nodo contenido en U_1 (por lo menos uno). Así, en cualquier iteración posterior a este cambio, se generará una coloración parcial distinta a U_1, por lo tanto, no se podrá encontrar un U_2 posterior a U_1 tal que $U_1 = U_2$, lo cual contradice el supuesto inicial.

De esta manera, se puede comprobar que no es posible que $U_1 = U_2$ para momentos distintos, y que la única posibilidad es que estos hayan surgido en el mismo momento de la ejecución, o que no existan.

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo 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"
– Matías Duhalde 12/11/19

FECHA: 16 de Noviembre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Pregunta 2

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

a) Dado $G(V, E)$ describe un algoritmo $O(L+LD)$ que entregue un orden de instalación de todas las librerías

Para generar este algoritmo, se puede ocupar un concepto similar al ordenamiento topológico de grafos (`topSort`), que en esencia utiliza **DFS**.

Como se aclaró en las *issues*, podemos asumir que no existe ningún tipo de dependencia circular entre las librerías.

Para implementar el algoritmo, se usan los algoritmos `dfs` y `dfsVisit` vistos en clases.

```
function dfs(startNode, V, alpha, array, i)
    let tiempo = 1
    for each (u in V) do
        u.color = "blanco"
    end for each
    tiempo = dfsVisit(alpha, startNode, tiempo, array, i)
end function

function dfsVisit(alpha, u, tiempo, array, i):
    u.color = "gris"
    u.start = tiempo
    tiempo += 1
    for each (v in alpha[u]) do
        if (v.color == "blanco") then
            tiempo = dfsVisit(v, tiempo)
        end if
    end for each
    u.color = "negro"
    u.end = tiempo
    array[i] = u // Insertar nodo en la array
    i -= 1
    tiempo += 1
    return tiempo
end function

function ordenLibrerias(startLib, V, alpha):
    let arrayOrdenada = |V|[ ] // array vacía de largo |V| (cantidad de libs)
    let i = |V| - 1
    dfs(startLib, V, alpha, arrayOrdenada, i)
    return arrayOrdenada
end function

```

```
// Llamada inicial
// startLib es la librería de la cual se quieren instalar sus dependencias,
// en este caso, correspondería a engine (está dentro de L)
// alpha es la estructura de datos que contiene los vecinos de cada nodo (aristas)
let orden = ordenLibrerias(startLib, L, alpha)
```

Complejidad:

Este algoritmo corresponde básicamente a DFS, con ciertas modificaciones. Como se discutió en clases, la complejidad en el peor caso de DFS corresponde a $O(|V| + |E|)$.

A este algoritmo se le agrego el tracking de los tiempos, y agregar elementos a una array (ver líneas 20 y 21). Todas estas operaciones extra, tienen complejidad constante $O(1)$, por lo que no modifican a $O(|V|+|E|)$. Además, notar que en lugar de iterar sobre todos los nodos en `dfs`, solamente se parte desde `startNode` (que sería engine en nuestro caso), debido a que tenemos certeza que desde este se pueden alcanzar todos los nodos de V (engine vendría siendo como la "base" del grafo). Esto no modificaría la complejidad, debido a que de igual manera se visitarían todos los nodos de V.

Ahora, V son los nodos del grafo, los cuales corresponden a las librerías L. E son las aristas del grafo, los cuales corresponden a las dependencias de cada librería. En el enunciado se especifica que cada librería tiene a lo más D dependencias. Por lo tanto, cada elemento $l \in L$, tiene a lo más D dependencias. Dado que hay $|L|$ librerías, el número total de dependencias sería a lo más $|L|D$, y tenemos que $|E| = |L|D$. Por lo tanto, la complejidad del algoritmo corresponde a $O(|L| + |L|D)$, o en una notación más flexible, $O(L + LD)$.

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

En esta pregunta, según el enunciado y las issues, se puede asumir que si una librería se encuentra instalada, entonces TODAS sus dependencias también se encuentran instaladas.

Sea `estaInstalada(u)` la función con complejidad $O(1)$ que detecta si una librería se encuentra instalada, que recibe un nodo y retorna verdadero o falso según corresponda.

```
function dfs(startNode, V, alpha, array, i)
    let tiempo = 1
    for each (u in V) do
        if (estaInstalada(u)) then
            u.color = "negro"
        else
            u.color = "blanco"
        end if
    end for each
    tiempo = dfsVisit(alpha, startNode, tiempo, array, i)
end function
```

Complejidad:

Se usa un algoritmo muy parecido al anterior, la única diferencia estaría en la función `dfs`, la cual asigna un color negro a aquellos nodos ya instalados, es decir, a aquellos nodos $u \in L - R$. De esta manera,

al iterar en `dfsVisit`, estos nodos no serían visitados, por lo tanto, la cantidad de nodos a visitar se reduciría de un total de L, a sólo R, que serían los que faltan por instalar.

Ahora, V son los nodos del grafo, los cuales corresponden a las librerías R. E son las aristas del grafo, los cuales corresponden a las dependencias de cada librería. Igual que antes, cada librería tiene a lo más D dependencias. Por lo tanto, cada elemento $r \in R$, tiene a lo más D dependencias. Dado que hay $|R|$ librerías que faltan por instalar, el número total de dependencias sería a lo más $|R|D$, y tenemos que $|E| = |R|D$. Por lo tanto, la complejidad del algoritmo corresponde a $O(|R| + |R|D)$, o en una notación más flexible, $O(R + RD)$.

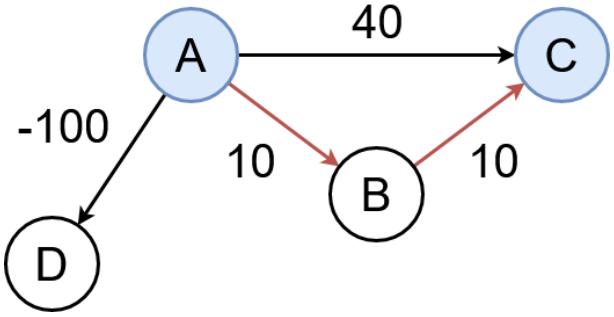
Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta"
– Matías Duhalde 12/11/19

FECHA: 16 de Noviembre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Pregunta 3

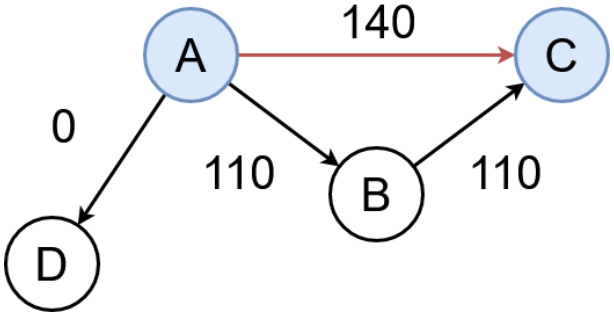
a) El grafo no contiene aristas de costo negativo

Supongamos que originalmente se tiene el siguiente grafo, el cual contiene una arista negativa:



Si nos concentramos en la ruta entre A y C (marcados en azul), tenemos que el camino más corto sería pasar por el nodo B, y por las dos aristas que tienen costo 10 (marcadas en rojo).

Si intentamos "arreglar" este diagrama usando lo propuesto en el enunciado, deberíamos restarle a cada arista el menor valor de costo (en este caso, -100), resultando el siguiente diagrama:

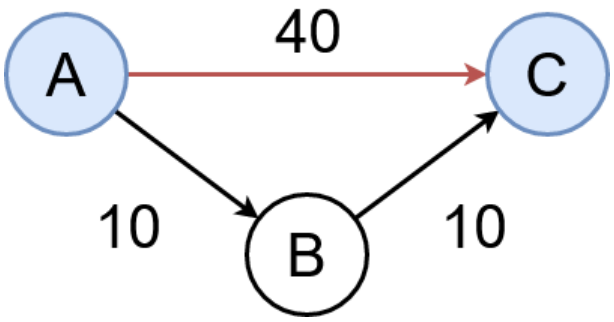


Ahora, volviendo a mirar la ruta entre A y C, tenemos que el camino más corto corresponde a la arista directa entre A y C (marcada en rojo), la cual tiene valor 140. El otro camino posible (que pasa por B), ahora tiene un valor de 220, por lo que ya no corresponde al camino más corto.

De esta manera se demuestra que las rutas encontradas por *Dijkstra* no se mantendrán en todos los casos, y que esta forma de intentar solucionar los casos con aristas negativas es incorrecta.

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

Se puede usar un ejemplo casi idéntico al primero. Considere el siguiente diagrama:



Al igual que en la sección anterior, usando el algoritmo original, la ruta más corta entre A y C sería aquella que pasa por B (en negro), la cual suma un valor de 20, en comparación con la otra directa entre A y C (marcada en rojo), que suma un valor de 40.

Sin embargo si definimos la ruta entre dos nodos como la multiplicación de los costos de cada arista, tenemos que el camino más corto ahora sería el directo entre A y C (arista roja), con valor 40. Si analizamos el otro camino, tenemos que su valor será la multiplicación del peso de la arista entre A y B (10), y la arista entre B y C (10), resultando un total de 100. Dado que $100 > 40$, la primera arista correspondería al camino más corto con esta definición.

De esta manera se demuestra que las rutas encontradas por *Dijkstra* no se mantendrán al cambiar la distancia de suma a multiplicación.

Nota: Como se discutió en las issues y se corrigió posteriormente en el enunciado, el valor inicial de d[s] en caso de la multiplicación corresponde a 1 (elemento neutro de la multiplicación).

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo 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"
– Matías Duhalde 12/11/19

FECHA: 16 de Noviembre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Pregunta 4

Nombre completo: Matías Patricio Duhalde Venegas
"Me comprometo 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"
– Matías Duhalde 12/11/19