PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC2133— ESTRUCTURAS DE DATOS Y ALGORITMOS 2020-2

FECHA: 11 de Diciembre de 2020
NOMBRE: Matías Patricio Duhalde Venegas

Interrogación 3

Pregunta 1

a) Demuestra que este problema tiene subestructura óptima.

Este problema es muy similar al problema de las tareas con ganancias que vimos en clases de programación dinámica. Cada elección de letrero se puede comparar a una tarea, en que si se elige un letrero a distancia d_i, entonces no podré elegir otro hasta $d_i + k$ (si estoy eligiendo los letreros en dirección de Arica a Punta Arenas, $d_i - k$ en la dirección contraria), y este letrero tendría una ganancia de w_i. Haciendo el paralelo, d_i sería la hora de comienzo de la tarea, $d_i + k$ la de término, y w_i la ganancia que tendría hacer la tarea. Igual que las tareas, estas distancias de los letreros no se pueden solapar. Tal como en el problema de las tareas, se puede separar al problema de los letreros en subestructuras óptimas.

Sea el L_i el letrero de publicidad a una distancia d_i de Arica, y que es visto por w_i personas al día, y sea Ω_i el conjunto que representa la solución óptima al problema para los letreros $1, ..., i$.

Tenemos que si L_j es parte de Ω_j, entonces, Ω_j puede calcularse a partir de la solución al subproblema de los letreros $1, ..., b(j)$, donde $b(j)$ representa al letrero con mayor i tal que este tenga una distancia menor o igual a $d_j - k$ de Arica. Por otro lado, si el L_j no es parte de Ω_j, este se puede calcular a partir de la solución al subproblema de los letreros $1, ..., j - 1$.

Lo anterior se puede demostrar por contradicción: Supongamos que Ω_j contiene efectivamente a L_j. También, supongamos por contradicción que cualquier L_i tal que $i < j$ puede pertenecer a Ω_j. Esto implica que se puede elegir un letrero L_r, tal que $b(j) < r < j$. Sin embargo, esto último posee una contradicción, debido a que cualquier letrero L_r tendría una distancia d_r, y $d_r > d_j - k$, lo que rompe la restricción del enunciado.

Es por esto, que el problema se puede solucionar a partir de sus mismos subproblemas, y se dice que tiene subestructura óptima.

b) Supón la siguiente estrategia codiciosa para resolver el problema: recorrer la lista de ubicaciones de letreros de Arica a Punta Arenas y poner un letrero en cada ubicación que cumpla con la restricción de las autoridades con respecto a la distancia con el último letrero escogido. Demuestra que, si todos los letreros tienen el mismo w, esta estrategia es óptima.

Nuevamente podemos hacer un paralelo con el problema de las tareas, en la cual si todas estas tenían el mismo valor, se podía resolver el problema con una estrategia codiciosa. Sin embargo, a diferencia de las tareas, cada letrero tendría una "duración" constante, o en otras palabras, la zona de exclusión de un letrero es igual para todos los otros letreros posibles. Gracias a esto, se puede partir colocando el primero que encontremos sin desviarse de la solución óptima.

Supongamos que elegimos el primer letrero posible (el más cercano a Arica). Usando el algoritmo codicioso, no se podrá elegir un nuevo letrero para nuestra solución óptima hasta que se cumpla la restricción $d_1 + k <= d_i$. De la misma manera y generalizando, si se elige un letrero d_i, no se podrá volver a elegir un letrero para la solución óptima hasta que se cumpla $d_i + k <= d_j$, o se llegue a Punta Arenas y no se puedan elegir más letreros.

Dado que elegimos L_1 para partir, y sea L_a el letrero que elegiremos después en nuestra solución óptima (es decir, el primero que encontremos después de L_1 que cumpla con la restricción), con $a > 1$, no podemos elegir ningún letrero r, con $1 < r < a$. Si en lugar de 1, eligiéramos un letrero L_r, no encontraremos una solución mejor a la inicial, debido a que en el mejor caso, el primer letrero que podremos elegir será L_a (mientras se cumpla que $d_r + k <= d_a$).

Entonces, considerando que comenzamos desde el primer letrero, todos los letreros tienen exactamente el mismo rango de exclusión, y todos los letreros poseen el mismo valor w, este algoritmo codicioso encontrará la solución óptima.

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

Pregunta 2

a) Dada una lista F de pares de forma (a, b) que indican amistad entre la persona a y la persona b, describe un algoritmo lineal en el número de pares que calcule la cantidad de grupos de amigos distintos que existen en Omashu.

Se asume que F se puede trabajar como una lista ligada, siendo `head` el primer elemento (nodo) de la cola,`value` su valor (que sería la tupla o el par), `next` el siguiente nodo de la lista, y `prev` el anterior. De esta manera se permite sacar nodos (hacer pop) de manera constante. También, se permite sacar el largo de la lista en tiempo lineal (o constante si se lleva cuenta de una referencia, pero no afecta nuestra complejidad).

```
// Se incluye la implementación de pop por si acaso
function pop(nodo)
    let nodo_anterior = nodo.prev
    let nodo_siguiente = nodo.next

    nodo_anterior.next = nodo_siguiente
    nodo_siguiente.prev = nodo_anterior
end function

function numeroDeGrupos(F)
    let numero_personas = 0
    let nodo_actual = F.head
    while (nodo_actual != NULL) do
        let par_actual = nodo_actual.value
        if (par_actual[0] == par_actual[1]) then
            // Se elimina el nodo de la lista ligada
            pop(nodo_actual)
            numero_personas++
        end if
        // La referencia aún existe
        nodo_actual = nodo_actual.next
    end while

    // Solo nos quedan en la lista original pares del tipo (a, b), (b,a).
    // La cantidad de pares restantes en F siempre será par
    let numero_grupos = numero_personas - (length(F) / 2)
    return numero_grupos
end function
```

El algoritmo cuenta la cantidad de personas

Esta función solamente tiene un ciclo, que sería el `while`, el cual itera solo una vez por todos los elementos. Todas las operaciones del ciclo tienen complejidad constante, por lo que todo el ciclo, y todo el algoritmo posee complejidad lineal $O(|F|)$.

Al salir del ciclo, se sacan los pares de tipo (a, a) de la lista inicial F. Al mismo tiempo, se van contando este tipo de nodos, los cuales representan la cantidad de personas que hay en F (esto debido a que si existe el par (a, b) en F, debe existir también (a, a) y (b, b)). Inicialmente, todas las personas representan un grupo independiente. Dado que sacamos todos los pares (a, a) de la lista, sólo nos quedan los pares del tipo $(a, b), (b, a)$ en esta. Hay un número par de pares dentro de esta, por lo que podemos dividir el largo en 2. Si existe el par (a, b), significa que estos dos forman un grupo, por lo que debemos restar uno al número de grupos. Siguiendo la misma lógica, por cada par de pares de elementos $(a, b), (b, a)$, se debe restar 1 a la cantidad de grupos inicial. De esta manera, se puede obtener la cantidad de grupos que tiene F con un algoritmo lineal.

b) Además de la lista F anterior, se te da una lista U con tríos de la forma (a, b, w). Cada trío indica que a y b no son amigos, pero podrían serlo si se les paga una cantidad positiva w de dinero. Describe un algoritmo a lo más *linearítmico* (es decir, del tipo $n \log n$) que calcule el costo mínimo necesario para que todos los habitantes de Omashu formen un solo gran grupo de amigos.

La idea general de este algoritmo es obtener un representante por cada grupo de amigos. De esta manera, se puede reducir la lista U a sólo las conexiones que tienen el costo mínimo, tal que se conectan dos grupos. De esta manera, no quedan conexiones triviales, tal que dos grupos se conecten entre sí por más de un camino, y para resolver el problema sólo basta con encontrar el MST, que se puede hacer con los algoritmos vistos en clases.

```
function contarNodos(F)
    let numero_personas = 0
    let nodo_actual = F.head
    while (nodo_actual != NULL) do
        let par_actual = nodo_actual.value
        if (par_actual[0] == par_actual[1]) then
            // Se elimina el nodo de la lista ligada
            pop(nodo_actual)
            numero_personas++
        end if
        // La referencia aún existe
        nodo_actual = nodo_actual.next
    end while
    return numero_personas
end function

function crearListaAdyacencia(F)
    let numero_nodos = contarNodos(F)
    // lista_adyacencia es una array de largo numero_nodos
    // cada elemento es un puntero a una linked_list inicialmente vacía
    let lista_adyacencia = array[numero_nodos]
    for (let i = 0; i < numero_nodos; i++) do
        lista_adyacencia[i] = LinkedList()
    end for
    for each (par in F) do
        lista_adyacencia[par[0]].append(par[1])
    end for each
    return lista_adyacencia
end function
```

```

function listaAGrafo(lista_adyacencia)
    let G = Grafo()
    for (let i = 0; i < |lista_adyacencia|; i++) do
        nodo = Nodo(i)
        nodo.vecinos = lista_adyacencia[i]
        G.V.append(nodo)
    end for each
    return G
end function

function listaRepresentantes(G)
    let numero_nodos = contarNodos(F)
    let lista_representantes = array[numero_nodos]
    for each (v in G.V) do
        // v.Rep contiene el representante de cada grupo asignado
        // por Kosaraju
        lista_representantes[v.value] = v.Rep
    end for
    return lista_representantes
end function

function reducirLista(lista_representantes, U)
    for (let i = 0; i < |U|; i++) do
        tupla = U[i]
        U[i] = (lista_representantes(tupla[0]),
                lista_representantes(tupla[1]),
                tupla[2])
    end for
end function

// -- Implementación --
// Preprocesamiento
lista_adyacencia = crearListaAdyacencia(F)
G = listaAGrafo(lista_adyacencia)
// Aplicar el algoritmo de kosaraju y obtener representantes
Kosaraju(G)
lista_representantes = listaRepresentantes(G)
// Reducir lista de costos
reducirLista(lista_representantes, U)
// Encontrar MST y sumar los costos
MST = Kruskal(U)
```

Primero, se crea una lista de adyacencia para representar el grafo a partir de F. Esto se hace en la función `crearListaAdyacencia`, la cual tiene una complejidad de $O(|F| + \mathit{numero\_nodos}) = O(|F|)$, debido que $F \geq \mathit{numero\_nodos}$. Posteriormente, la lista de adyacencia se convierte en un grafo en complejidad $O(|F|)$ (que usa estructuras de datos) para efectuar el resto de los algoritmos.

Luego, se usa el algoritmo de Kosaraju visto en clases para detectar las CFC, y asignar un representante a cada una de estas. Cada CFC representaría un grupo de amigos. De esta manera, se asignan representantes en cada grupo para manejar mejor a U posteriormente. La complejidad de Kosaraju, como se vio en clases, es de $O(|V| + |E|)$, que nuevamente estaría acotado por $O(|F|)$.

Posteriormente, luego de obtener los representantes, se "procesa" la lista de costos U. Por ejemplo, para una tupla (a, b, w) de U, si x es el representante de a y y el de b, la tupla resultante sería (x, y, w). Esta operación itera sobre U, por lo que su complejidad temporal es $O(|U|)$.

Finalmente, para encontrar el MST se usa el algoritmo de Kruskal. Dado que las aristas se ordenan de menor a mayor por costos, si hay dos aristas que conectan los mismos grupos, por ejemplo (x, y, w_1) y (x, y, w_2), siempre se tomará primero la que tiene menor w, y por lo tanto, la estrategia codiciosa no se rompe. Kruskal retorna las aristas que representan el MST. Como se vió en clases, la complejidad de Kruskal en el peor caso es de $O(|E| \log |V|)$, que en este caso sería $O(|U| \log |F|)$.

Después de esto, sólo bastaría con sumar el valor de cada Arista para obtener el valor final esperado. Dado que $|MST| \leq |U|$, obtener este valor final está acotado por $O(|U|)$.

Por lo tanto, la complejidad final será:

$$3 \cdot O(|F|) + 2 \cdot O(|U|) + O(|U| \log |F|) = O(|U| \log |F|)$$

La cual es de forma *linearítmica*.

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 [illegible]

Pregunta 3

a) Definimos L como el máximo $f(v)$ entre todos los posibles v. Modifica el algoritmo de Bellman-Ford para que su complejidad sea $\Theta(L \cdot E)$ para cualquier grafo.

```
1  function bellman-ford(s)
2      for each (u in V) do
3          d[u] = inf
4          pi[u] = null
5      end for each
6      d[s] = 0
7      for (k = 1 .. |V| - 1) do
8          let changed = false
9          for each ((u,v) in E) do
10             if (d[v] > d[u] + w(u,v)) then
11                 d[v] = d[u] + w(u, v)
12                 pi[v] = u
13                 changed = true
14             end if
15         end for each
16         if (not changed) then
17             break for
18         end if
19     end for
20 end function
```

Los cambios realizados fueron agregar las líneas 8, 16 y 17. Básicamente corresponde a agregar una condición que detecta si hubo algún cambio en la iteración actual, y si no hubo ninguno, se rompe el loop `for (k = 1 .. |V| - 1)`. Si no ocurre un cambio en ninguno de los nodos, significa que ya se alcanzó el camino más corto para todos los nodos, y por consecuencia, el número definitivo de aristas de la ruta de menor costo s a v de todos los v (es decir, $f(v)\ \forall v \in V$). De esta manera, no es necesario seguir iterando porque no ocurrirán cambios posteriormente, y se puede simplemente hacer un break (romper el loop).

Es por esto, que se puede decir que se alcanzó el máximo valor posible L de $f(v)$, y la complejidad queda acotada por $\Theta(L \cdot E)$.

A pesar del cambio, tenemos que en el peor caso para ciertos grafos $L = |V| - 1$, por lo que no reduciría el tiempo de ejecución en esos casos (aunque de igual manera, por como está definido L, continuaría estando acotado por este).

b) Sea $g(v)$ la cantidad de aristas que llegan a v. Modifica el algoritmo de Bellman-Ford para que el tiempo que tome esté dado por la siguiente expresión:

$$T(V, E) = \Sigma_{v \in V} f(v) \cdot g(v)$$

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 [illegible]

FECHA:	11 de Diciembre 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 13/10