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

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

Interrogación 3

Pregunta 1

La agencia publicitaria para la que trabajas ha sido contratada por XIAOMI MEJOR RELACIÓN PRECIO CALIDAD para posicionar letreros publicitarios en la ruta de Arica a Punta Arenas. Para esto tienes una lista de **n** ubicaciones de letreros disponibles, con sus distancias d_1, d_2, ... , d_n medidas desde Arica y ordenadas crecientemente, y una estimación de que el i-ésimo letrero es visto por w_i personas al día.

El cliente quiere maximizar la cantidad de personas que ven los letreros al día. Tú debes tener en cuenta que las autoridades no permiten que haya dos letreros del mismo producto o compañía a una distancia menor a k entre ellos.

¿Cuáles ubicaciones de letreros debes utilizar para esto?

a) Demuestra que este problema 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.

Solución Pregunta 1.a)

Una subestructura óptima significa que la solución óptima del problema original contiene soluciones óptimas de problemas más pequeños, pero similares.

Supongamos que tenemos la solución óptima, y tenemos que la posición del primer letrero es en **m**, donde **m** $<$ **n**. Si consideramos el sub problema, similar al original, donde existen $]\boldsymbol{m}, \boldsymbol{n}]$ ubicaciones, con distancias $]\boldsymbol{d_m}, \boldsymbol{d_n}]$, deberíamos tener una solución óptima del sub problema que pertenezca a la original, maximizando la cantidad de personas que ven los letreros al día y considerando la restricción que debe existir una distancia

k entre ellos.

De no ser así, tendríamos una solución que permita que más personas vean letreros de XIAOMI entre la posición **m** y **n**, la cual se podría utilizar considerando el problema desde Arica desde la posición **m** en adelante. Contradiciendo así la suposición de que la solución que tenemos es óptima.

Solución alternativa: Demostración por contradicción

Asumamos que tenemos una solución óptima del problema $\boldsymbol{S}$, con las ubicaciones de los letreros que maximizan las vistas por días y cumplen la restricción de distancias.

Ahora elijamos arbitrariamente un letrero (que llamaremos $\boldsymbol{s_i}$) de S para hacer un "corte" en nuestro camino óptimo. Este letrero está a una distancia $\boldsymbol{d_i}$ de Arica. Por la restricción de distancia, el letrero anterior (en S) a $\boldsymbol{s_i}$ debe estar a una distancia menor a $\boldsymbol{d_i - k}$ y el letrero siguiente a una distancia mayor a $\boldsymbol{d_i + k}$.

Claramente el valor estimado de visitas total de nuestro camino óptimo S es igual a la suma entre; la suma de las visitas de cada letrero anterior a $\boldsymbol{x_i}$ (llamaremos a este conjunto $\boldsymbol{S_0}$), $\boldsymbol{w_i}$ y la suma de los letreros que siguen a $\boldsymbol{x_i}$ (llamaremos a este conjunto $\boldsymbol{S_1}$).

Si existiera un conjunto de letreros del conjunto original, cada uno con una distancia menor a $\boldsymbol{d_i - k}$, cuya suma de visitas fuera mayor a la suma de las visitas de $\boldsymbol{S_0}$, entonces uniendo tal conjunto con $\boldsymbol{s_i}$ y $\boldsymbol{S_1}$, obtendramos una solución con mayor número de visitas total que $\boldsymbol{S}$, contradiciendo la suposición de que $\boldsymbol{S}$ es una solución óptima. Por lo tanto $\boldsymbol{S_0}$ debe ser solución óptima del subproblema entre las distancias **0** y $\boldsymbol{d_i - k}$. Por un argumento muy similar, concluimos que $\boldsymbol{S_1}$ también es solución óptima del problema entre las distancias $\boldsymbol{d_i + k}$ y $\boldsymbol{d_n}$.

Por lo tanto demostramos que una solución óptima $\boldsymbol{S}$ está compuesta de dos subestructuras óptimas, más el letrero $\boldsymbol{s_i}$ del corte, por lo que $\boldsymbol{S}$ tiene subestructura óptima.

Solución Pregunta 1.b)

Demostración por contradicción.

Supongamos que todos los pesos de las ubicaciones son iguales y que la estrategia codiciosa de recorrer la lista y asignar las ubicaciones que cumplan las restricciones de las autoridades no es óptima.

Esto significa que, a medida que yo recorro la lista y voy asignando ubicaciones, aquellas que cumplan con la restricción de las autoridades no siempre serán óptimas.

Como la ubicación escogida no es óptima, significa que en algún momento se escogió una ubicación en vez una con mayor utilidad o se escogieron menos ubicaciones que las que podrían haberse escogido.

Como este método de recorrer la lista de ubicaciones escogiendo la primera que cumple con las restricciones, entonces siempre escoge la mayor cantidad de ubicaciones posibles desde el origen (Arica)*.

Por lo tanto nos queda solo la segunda opción. Es decir, que en algún momento, se escogió una ubicación por sobre otra que aportaba con mayor utilidad. Sin embargo, esto contradice nuestra suposición inicial.

Es por esto que queda demostrado que, dado que los pesos de utilidad son iguales, recorrer la lista de ubicaciones y escoger aquellas que cumplan con las restricciones de las autoridades es una estrategia óptima de asignación.

*Demostración de que siempre se escoge la mayor cantidad de ubicaciones posibles. Este es un problema de programación dinámica con sub-estructura óptima. Por lo tanto, se demostrará por inducción.

[BI]

Si hay una sola ubicación, trivialmente se cumple que escogerla (también es la más cercana al punto de origen) cumple con las restricciones y maximiza la cantidad de ubicaciones asignables posible. Por lo tanto, se cumple el caso base.

[HI]

Supongamos que tenemos las ubicaciones ordenadas por distancia al origen (Arica) en una lista de la forma U = $\boldsymbol{\{u_0, u_1, u_2, ..., u_n\}}$. Supongamos también que estamos en la ubicación $\boldsymbol{u_i}$ que cumple las restricciones y que agregar $\boldsymbol{u_i}$ a la solución final S maximiza la distancia restante. Es decir $\boldsymbol{D - d(u_j) < D - d(u_i)\ \forall i < j}$ con $\boldsymbol{D}$ la distancia de Arica a Punta Arenas.

[TI]

Ahora, tomamos la siguiente ubicación factible que cumple con las restricciones, $\boldsymbol{u_{i+q}}$, tendremos que $\boldsymbol{D - d(u_{i+q}) < D - d(u_{i+q-e})}$ con $\boldsymbol{1 < e < q + 1}$, pero estas no son soluciones factibles porque incumplen las restricciones de distancia mínima entre ubicaciones. Por otro lado, $\boldsymbol{D - d(u_{i+q+h}) < D - d(u_{i+q})}$ con $\boldsymbol{1 < h < n - i - q}$ porque $\boldsymbol{i + q < i + q + h}$ y las ubicaciones están ordenadas de menor a mayor. Por lo tanto, $\boldsymbol{D - d(u_{i+q})}$ maximiza la distancia restante.

Queda entonces demostrado que escoger la ubicación más cercana al punto de origen que cumpla las restricciones maximiza la utilidad total.

[2pts] Si hay problemas de formalidad

[3pts] Si está correcta la demostración

Pregunta 2

La gente de la tierra de Omashu se toma los grupos de amigos muy en serio. Tan en serio, que podemos describirlos matemáticamente (son clases de equivalencia):

- Si a es amigo de b entonces b es amigo de a
- Cada persona es amiga de sí misma y cada persona pertenece a un solo grupo de amigos
- Si a forma parte del grupo X, y b es amigo de a, entonces necesariamente b forma parte de X.

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.

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 $nlogn$) que calcule el costo mínimo necesario para que todos los habitantes de Omashu formen un solo gran grupo de amigos.

Solución Pregunta 2a)

Se puede apreciar que cada grupo de amigos corresponde a un conjunto disjunto, donde nunca va existir un amigo que este en dos grupos distintos. Para la resolución del problema, se cuenta con un grafo donde cada nodo es una persona y las aristas son los pares $\boldsymbol{(a, b)}$ de la lista $\boldsymbol{F}$

Podemos notar que se pide calcular la cantidad de conjuntos disjuntos, los cuales se representan por los grupos de amigo, es por esto que se puede utilizar el algoritmo visto en clases **Kruskal** con modificaciones, tal como se muestra a continuación

```
1:  procedure OMASHU(F)
2:      groups := 0
3:      for (a, b) ∈ F do
4:          if a = b then
5:              make_set(a)
6:              groups := groups + 1
7:          end if
8:      end for
9:      for (a, b) ∈ F do
10:         if find_set(a) ≠ find_set(b) then
11:             union(a, b)
12:             groups := groups − 1
13:         end if
14:     end for
15:     return groups
16: end procedure
```

En primer lugar, como cada persona es amiga de si misma, se inicializa a cada nodo como su propio representante y junto con esto, se tiene un contador que se le va sumando uno por cada vez que se encuentra una nueva persona, de esta forma, después de haber recorrido todos los pares pertenecientes a $\boldsymbol{F}$, el contador ***groups*** tiene el total de nodos que tiene el grafo, que inicialmente van a ser nuestros conjuntos disjuntos

Luego, se vuelve a recorrer todos los pares de $\boldsymbol{F}$ y si se encuentra que el representante de $\boldsymbol{a}$ es distinto al de $\boldsymbol{b}$, se une los conjuntos, ya que pertenecen al mismo grupo, dejando al mismo representante para ambos. Como se unen los conjuntos, disminuye la cantidad de conjuntos disjuntos, por lo que hay que restarle uno a ***groups***

Finalmente, después de haber recorrido todos los pares de $\boldsymbol{F}$, se tiene la cantidad total de grupos de amigos distintos que existen en Omashu

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

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

[1 pt] Por justificar la complejidad lineal

En caso de plantear un algoritmo que no sea a lo más lineal, el puntaje máximo a obtener es 1 pto

Solución Pregunta 2b)

Este es similar al anterior. Solo que ahora primero se unirán ambas listas antes de seguir con el resto del algoritmo. Ahora en vez de calcular el número de grupos, se irá sumando el costo cada vez que haya una unión, retornando finalmente el costo total. Este algoritmo entrega complejidad $\boldsymbol{n \log n}$ dada la demostración vista en clases.

```
1:  procedure OMASHU(F, U)
2:      for (a, b) ∈ F do
3:          union(U, (a, b, 0))
4:      end for
5:      sort(U) de menor a mayor w
6:      cost := 0
7:      for (a, b, w) ∈ U do
8:          if a = b then
9:              make_set(a)
10:         end if
11:     end for
12:     for (a, b, w) ∈ U do
13:         if find_set(a) ≠ find_set(b) then
14:             union(a, b)
15:             cost := cost + w
16:         end if
17:     end for
18:     return cost
19: end procedure
```

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

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

[**1 pt**] Por justificar la complejidad a lo más linearítmica

En caso de plantear un algoritmo que no sea a lo más linearítmico, el puntaje máximo a obtener es 1 pto

Pregunta 3

Considere el siguiente algoritmo de Bellman-Ford para un grafo dirigido, con costos y sin ciclos de costo acumulado negativo:

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    for k = 1..|V| − 1:
        for each (u, v) ∈ E:
            if d[v] > d[u] + w(u, v):
                d[v] ← d[u] + w(u, v)
                π[v] ← u
```

Este algoritmo calcula las rutas de menor costo desde el vértice s a todos los demás vértices del grafo en tiempo $\Theta(V \cdot E)$. Queremos mejorar su rendimiento.

Sea $f(v)$ el número de aristas de la ruta de menor costo de s a v.

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

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:

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

Solución Pregunta 3a)

Recordemos que luego de la i-ésima iteración del algoritmo, todas las rutas tienen a lo más i aristas. Si $f(v) < i$, entonces las siguientes iteraciones no modificarán la ruta del vértice v.

En particular, la cantidad de iteraciones que será necesario hacer está dada por el mayor $f(v)$, L.

Podemos modificar el algoritmo para que detecte si hubo cambios en la ruta más corta de cualquier vértice. Si en una iteración no hubo cambios, significa que en todas las siguientes tampoco habrá cambios.

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    for k = 1..|V| − 1:
        changed ← false
        for each (u,v) ∈ E:
            if d[v] > d[u] + w(u,v):
                d[v] ← d[u] + w(u,v)
                π[v] ← u
                changed ← true
        if not changed, break
```

Y este punto se produce cuando hacemos la L-ésima iteración, ya que la ruta de ese vértice es la última en terminar de calcularse: de ahí en adelante las iteraciones no cambian las rutas calculadas.

Es decir, el *for* termina luego de la L-ésima iteración, y dentro del *for* se recorren todas las aristas cada vez, para una complejidad de $O(L \cdot E)$ en total (ignorando el $O(V)$ de la preparación del grafo)

Aclaración importante: Responder con SPFA es incorrecto porque, si bien este está acotado por arriba por $\mathcal{O}(L * E)$, no lo está por abajo (no se acota por $\Omega(L * E)$), y la notación pedida era en Big Theta, por lo que ambas condiciones debían cumplirse.

Solución Pregunta 3b)

Aclaración importante: A continuación se muestra una opción de respuesta. Esta, sin embargo, fue analizada y no sirve para todos los casos. Por lo tanto, se tomará como correcta cualquier propuesta de algoritmo que cumpla con los tiempos, aunque no sirva para todos los casos. También, se aceptará cualquier mejora sobre el algoritmo de la parte (a) de la pauta (que tenga mejor caso con complejidad menor a $\Omega(L * E)$ y sea correcto siempre), aunque no cumpla específicamente con el tiempo de ejecución pedido en todos los casos (como *Shortest Path Faster Algorithm* (SPFA)).

Como se dijo antes, luego de la i-ésima iteración del algoritmo, todas las rutas tienen a lo más i aristas. Si $f(v) < i$, entonces las siguientes iteraciones no modificarán la ruta del vértice v.

Para un vértice v, las únicas aristas que pueden modificar su ruta son las de la forma (u, v). Si en algún minuto del algoritmo la ruta de v ya es la más corta, no tiene sentido revisar ninguna de estas aristas.

Tal como $\alpha[v]$ son las aristas que salen de v, definimos $\beta[v]$ como las aristas que llegan a v.

En primer lugar, reformulamos el algoritmo para definirlo en función de β

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    for k = 1..|V| − 1:
        for each v ∈ V:
            for each u ∈ β[v]:
                if d[v] > d[u] + w(u, v):
                    d[v] ← d[u] + w(u, v)
                    π[v] ← u
```

Y ahora agregamos la parte de no revisar las aristas que llegan a vértices que ya están listos.

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    C ← V                                   // Vértices que no están listos
    while C ≠ ∅:
        C' ← ∅                              // Aquí guardaremos los que no están listos
        for each v ∈ C:
            for each u ∈ β[v]:
                if d[v] > d[u] + w(u, v):
                    d[v] ← d[u] + w(u, v)
                    π[v] ← u
                    C' ← C' ∪ {v}           // Si no entra al if ni una vez es porque está listo
        C ← C'
```

Por lo tanto, para cada vértice se repite su $for\ u \in \beta[v]$ unas $f(v)$ veces. Como $g(v) = |\beta[v]|$, entonces cada vértice aporta $f(v) \cdot g(v)$ pasos al algoritmo.

Distribución Puntaje P3)

IMPORTANTE: el puntaje máximo de esta pregunta son 6 puntos, es decir, si se obtienen 8 puntos entre la a y la b, el puntaje final serán 6 ptos

[P3a] Máximo 6 puntos

- A1.1: No llega a la condición y no justifica apropiadamente que L iteraciones son suficientes **(0 ptos**

parte a)

- A1.2: No llega a la condición, pero sí justifica apropiadamente que L iteraciones son suficientes **(3ptos parte a)**
- A2.1: Incluye SPFA en a, pero justifica su correctitud o valor con errores. **(0ptos parte a)**
- A2.2: Incluye SPFA en a y justifica apropiadamente su correctitud y valor **(0ptos parte a, 2 ptos bonus B)**
- A3: Justifica apropiadamente que L iteraciones son suficientes y llega a la condición de término **(6 ptos)**

[P3b] Máximo 2 puntos de bonus para complementar a.

- A2.2: Incluye SPFA **en a** y justifica apropiadamente su correctitud y valor **(0 ptos parte a, 2 ptos bonus b)**
- B1: No responde b (0 ptos parte b)
- B2: Propone algoritmo incorrecto o incorrectamente justificado (puede ser SPFA con correctitud o valor mal justificado) **(0 ptos bonus b)**
- B3: Propone SPFA (o algortimo con mejor caso mejor a $\mathbf{\Omega}$(LE)) correctamente justificado (2 ptos bonus b)
- B4: Algoritmo que no sirve para todos los casos, pero cumple con complejidad **(2 ptos)**

Pregunta 4

Se acercan las fiestas, y el generoso Krampus te presenta $\boldsymbol{n}$ regalos, de los cuales debes escoger $\boldsymbol{k}$. Cada regalo trae la etiqueta con su precio, y como eres una persona materialista, quieres maximizar la suma de los precios de los regalos que elijas. Los regalos están dispuestos en la forma de un árbol binario de navidad (no de búsqueda), donde cada nodo corresponde a un regalo. Hay una sola restricción para los regalos que puedes escoger: si escoges el regalo $\boldsymbol{u}$, necesariamente debes escoger el padre de $\boldsymbol{u}$ en el árbol, y así sucesivamente.

Escribe un algoritmo de programación dinámica que indique los $\boldsymbol{k}$ regalos que debes escoger de manera de maximizar el precio total.

Solución Pregunta 4)

A continuación se propone un algoritmo que resuelve el problema. Este algoritmo no es el único que lo resuelve. Se evalúa que el algoritmo que propongan cumpla los criterios establecidos en el desglose de puntaje.

```
function ProgramacionDinamica(u, k):
    Aux = diccionario inicialmente vacío
    maximum, set = C(u, k)
    return set

function C(u, k):
    if Aux[u, k] not null:
        return Aux[u, k]
    if k == 0:
        Aux[u, k] = (0, {})
    else if k == 1:
        Aux[u, k] = (u.precio, {u})
    else if left(u) is null and right(u) is null and k > 1:
        Aux[u, k] = (-Inf, {})
    else:
        max = 0, set = {u}
        for i in 0..k-1:
            price_left, set_left = C(left(u), i)
            price_right, set_right = C(right(u), k - 1 - i)
            if price_left + price_right > max:
                set = {u} union set_left union set_right
                max = price_left + price_right
        Aux[u, k] = (max + u.price, set)
    return Aux[u, k]
```

- 2pt Por definir la función recursiva que resuelva el problema en base a la resolución de subproblemas del mismo índole (Tipo **DFS**) y que recorra solo hasta $\boldsymbol{k}$. 1pt si no recorre hasta $\boldsymbol{k}$.
- 1pt Por utilizar programación dinámica, implementando la estructura de datos auxiliar (o bien creándola en el mismo nodo) que almacene resultados anteriores (puede ser tabla de hash, diccionario, arreglo, etc).
- 1pt Por utilizar la estructura auxiliar adecuadamente.

- 1pt Por definir correctamente todos los casos bases y casos bordes del problema. 0,5pts si define los casos bases pero no casos bordes o vice versa.
- 1pt Por crear correctamente el set y retornarlo. 0pts si no retorna el conjunto o si el conjunto es incorrecto.