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

IIC2133 — Estructuras de Datos y Algoritmos 1'2025

Examen 8 de julio de 2025

Condiciones de entrega: Para quienes rindieron la I1 y la I2 deben entregar solo 3 de las 4 preguntas. Para quienes tienen interrogación recuperativa, deben entregar 3 preguntas, donde una de ellas debe ser la correspondiente a la materia de la interrogación faltante

Tiempo: 2 horas

Entrega: Al final de la prueba tienen 10 minutos para subir la imagen de la prueba a Canvas,**cada pregunta y el torpedo** por separado en vertical

Evaluación: Cada pregunta tiene 6 puntos (+1 punto base). La nota es el promedio de las 3 preguntas entregadas. La nota de la I recuperativa es el promedio de las preguntas recuperativas y la pregunta correspondiente del examen.

1. Materia I1: Ordenamiento y Dividir para Conquistar

En una carrera de tortugas participan n de ellas. Antes de largar, cada tortuga ocupa una posición de salida de acuerdo al ranking del campeonato anual. La posición 1 en la largada la ocupa la primera tortuga en el ranking, la posición 2 la segunda tortuga (que larga un poco más atrás que la primera), y así sucesivamente. Cada tortuga lleva su número pegado en el caparazón. Al finalizar la carrera, se registra la posición de llegada de cada tortuga en un arreglo $P[1..n]$, donde $P[i]$ indica la posición de largada de la tortuga que llegó en la posición i. Por ejemplo, si $P[3] = 2$, significa que la tortuga que partió segunda terminó llegando tercera. Un **adelantamiento** se define como un par de tortugas (i, j) tal que $i < j$ y $P[i] > P[j]$.

1. (4 pts.) Diseñe un algoritmo eficiente de tipo *dividir para conquistar* que calcule la cantidad total de adelantamientos producidos, dado el arreglo $P[1..n]$ como entrada.
   Ejemplo: Si $P = [3, 1, 5, 4, 2]$, la cantidad de adelantamientos es 5.

   Solución:

Input: Arreglo $P[1..n]$ de posiciones de largada de los caracoles.
Output: Número total de adelantamientos (inversiones).
Function `CountInversions`$(P,\ izq,\ der)$**:**
    if $izq \geq der$ **:**
        return 0
    else:
        $m \leftarrow \lfloor (izq + der)/2 \rfloor$
        $i_1 \leftarrow$ `CountInversions`(P, izq, m)
        $i_2 \leftarrow$ `CountInversions`$(P, m+1, der)$
        $invMerge \leftarrow$ `MergeCount`(P, izq, m, der)
        return $i_1 + i_2 + invMerge$

Function `MergeCount`(P, izq, m, der)**:**
    Crear arreglos temporales L y R
    Copiar $P[izq..m]$ en L, $P[m+1..der]$ en R
    $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow izq$, $invCount \leftarrow 0$
    while $i \leq |L|$ y $j \leq |R|$ **:**
        if $L[i] \leq R[j]$ **:**
            $P[k] \leftarrow L[i]$; $i \leftarrow i + 1$
        else:
            $P[k] \leftarrow R[j]$; $j \leftarrow j + 1$
            $invCount \leftarrow invCount + (|L| - i + 1)$
        $k \leftarrow k + 1$
    Copiar elementos restantes de L y R (si los hay) en P
    return $invCount$

Pauta de evaluación:

- Caso base: 0.3 puntos.
- Dividir: $(m \leftarrow \lfloor (izq + der)/2 \rfloor)$, 0.6 puntos.
- Conquistar: llamadas recursivas a `CountInversiones`, 0.6 puntos.
- Combinar: `MergeCount` correcto (2.0 puntos) y **return** $i_1 + i_2 + invMerge$ (0.5 puntos)

2. (2 pts.) Justifique la complejidad de su algoritmo. ¿Qué complejidad tendría una solución ingenua que verifica todos los pares posibles?
   Solución: El tiempo de ejecución es $O(n \lg n)$ porque el algoritmo es básicamente un MergeSort modificado. Una solución ingenua tendría complejidad $O(n^2)$, básicamente por cada posición i del arreglo P hay que contar cuántas inversiones respecto al elemento en $P[i+1..n]$.

2. Materia I2: Árboles Rojo-negro

Como vimos en clase, la inserción de un nodo en un árbol rojo-negro se puede dividir en tres casos; en todos los casos, el nodo x recién insertado se pinta inicialmente de rojo:

1. el padre p de x es negro → estamos listos

2. el padre p de x es rojo, pero el hermano q de p es negro (si p no tiene hermano, lo suponemos negro) → hacemos una o dos rotaciones y le cambiamos el color a dos nodos [ustedes tienen que saber los detalles de este caso]

3. el padre p de x es rojo, y el hermano q de p también es rojo (es decir, el nodo recién insertado, su padre y su tío son todos rojos) → como el abuelo r de x necesariamente es negro, cambiamos los colores de r y de sus hijos p y q; ahora r es rojo (y sus hijos p y q son negros), por lo que hay que revisar el color del padre s de r :

   - s es negro → estamos listos
   - s es rojo → repetimos el caso 2) o el caso 3), según corresponda, pero ahora con s en lugar de p, y el hermano t de s en lugar de q

El caso 3) significa que potencialmente podemos llegar de vuelta hasta la raíz del árbol. Una manera de evitar tener que hacer este recorrido "de vuelta" es ir preparando el árbol a medida que vamos bajando (desde la raíz, cuando estamos buscando el punto en que hay que hacer la inserción), de modo de garantizar que q no será rojo —algoritmo de inserción top-down :
Al ir bajando, cuando encontramos un nodo U que tiene dos hijos rojos, cambiamos los colores de U (a rojo) y de sus hijos (a negro); esto producirá un problema solo si el padre V de U también es rojo, en cuyo caso aplicamos el caso 2) anterior.
Muestra la ejecución de este algoritmo de inserción top-down al insertar la clave 22 en el siguiente árbol rojo-negro: la raíz tiene la clave 35; sus hijos, las claves 30 (rojo) y 42 (negro); los dos hijos negros de 30 tienen las claves 25 y 33; los dos hijos rojos de 42 tienen las claves 40 y 45; y los dos hijos rojos de 25, tienen las claves 20 y 27.

Árboles rojo-negros

Como vimos en clase, la inserción de un nodo en un árbol rojo-negro se puede dividir en tres casos; en todos los casos, el nodo *x* recién insertado se pinta inicialmente de rojo:

1) el padre *p* de *x* es negro → estamos listos
2) el padre *p* de *x* es rojo, pero el hermano *q* de *p* es negro (si *p* no tiene hermano, lo suponemos negro) → hacemos una o dos rotaciones y le cambiamos el color a dos nodos [**ustedes tienen que saber los detalles de este caso**]
3) el padre *p* de *x* es rojo, y el hermano *q* de *p* también es rojo (es decir, el nodo recién insertado, su padre y su tío son todos rojos) → como el abuelo *r* de *x* necesariamente es negro, cambiamos los colores de *r* y de sus hijos *p* y *q*; ahora *r* es rojo (y sus hijos *p* y *q* son negros), por lo que hay que revisar el color del padre *s* de *r* :

   s es negro → estamos listos

   s es rojo → repetimos el caso 2) o el caso 3), según corresponda, pero ahora con *s* en lugar de *p*, y el hermano *t* de *s* en lugar de *q*

El caso 3) significa que potencialmente podemos llegar de vuelta hasta la raíz del árbol. Una manera de evitar tener que hacer este recorrido "de vuelta" es ir preparando el árbol a medida que vamos bajando (desde la raíz, cuando estamos buscando el punto en que hay que hacer la inserción), de modo de garantizar que *q* no será rojo —algoritmo de inserción *top-down* :

> Al ir bajando, cuando encontramos un nodo *U* que tiene dos hijos rojos, cambiamos los colores de *U* (a rojo) y de sus hijos (a negro); esto producirá un problema solo si el padre *V* de *U* también es rojo, en cuyo caso aplicamos el caso 2) anterior.

Muestra la ejecución de este algoritmo de inserción *top-down* al insertar la clave 22 en el siguiente árbol rojo-negro: la raíz tiene la clave 35; sus hijos, las claves 30 (rojo) y 42 (negro); los dos hijos negros de 30 tienen las claves 25 y 33; los dos hijos rojos de 42 tienen las claves 40 y 45; y los dos hijos rojos de 25, tienen las claves 20 y 27.

Respuesta (estos son, más o menos, los pasos importantes):

Al bajar desde la raíz, con clave 35, no encontramos ningún caso especial hasta que llegamos al nodo negro con clave 25, que tiene ambos hijos rojos: 20 y 27. [**1 pt**]

Entonces aplicamos la nueva regla: cambiamos los colores de 25 a rojo y de sus dos hijos, 20 y 27, a negros. [**1.5 pts**]

Como el padre de 25 también es rojo (el nodo con clave 30), aplicamos el caso 2: hacemos una rotación simple a la derecha de la arista 35–30 —con lo que queda 30 como raíz (roja), con hijos 25 y 35 negros— e intercambiamos colores entre padre e hijos, dejando a 30 negro, y a 25 y 35 ambos rojos. [**2 pts**]

Ahora el nodo con clave 25 tiene ambos hijos —con claves 20 y 27— negros; y como 20 es una hoja, insertamos allí el nuevo nodo con clave 22, que pintamos de rojo. [**1.5 pts**]

Treaps

3. Grafos

Considera una institución de educación superior que ofrece sus **programas** en la modalidad "on-line." Un programa consiste en múltiples cursos (o asignaturas), cada uno de los cuales tiene como prerrequisitos cero o más cursos del mismo programa. Un/una estudiante puede inscribir todos los cursos que quiera en un mismo período académico; la única restricción es que al momento de inscribir un curso al inicio del período académico, tenga aprobados los prerrequisitos correspondientes.
El programa de Inteligencia Artificial Ética se ha vuelto muy popular, por lo que la institución le ha ido agregando más y más cursos a medida que aparecen nuevos temas en el debate público. De pronto, tú te das cuenta de que te encantaría estudiar este programa, y, estando dispuesto/a a hacer el esfuerzo que sea necesario, tu plan es terminarlo lo antes posible. Por ello, consultas a la institución, "¿Cuál es el mínimo número de períodos académicos en que puedo terminar el programa?" La institución se complica un montón para darte la respuesta —hay demasiados cursos y demasiados prerrequisitos— y, por lo tanto, distintas personas te dan distintas respuestas. Por lo que tú ofreces solucionar el problema eficientemente —para este programa y cualquier otro que tenga la institución— a cambio de un descuento en la matrícula. Explica de manera clara y precisa cómo resolverías el problema aplicando las estructuras de datos y algoritmos estudiados este semestre en iic2133.

Rutas mínimas

Considera una institución de educación superior que ofrece sus programas en la modalidad "on-line." Un programa consiste en múltiples cursos (o asignaturas), cada uno de los cuales tiene como prerrequisitos cero o más cursos del mismo programa. Un/una estudiante puede inscribir todos los cursos que quiera en un mismo período académico; la única restricción es que al momento de inscribir un curso al inicio del período académico, tenga aprobados los perrequisitos correspondientes.

El programa de Inteligencia Artificial Ética se ha vuelto muy popular, por lo que la institu-ción le ha ido agregando más y más cursos a medida que aparecen nuevos temas en el debate público. De pronto, tú te das cuenta de que te encantaría estudiar este programa, y, estando dispuesto/a a hacer el esfuerzo que sea necesario, tu plan es terminarlo lo antes posible. Por ello, consultas a la institución, "¿Cuál es el mínimo número de períodos académicos en que puedo terminar el programa?"

La institución se complica un montón para darte la respuesta —hay demasiados cursos y demasiados prerrequisitos— y por lo tanto distintas personas te dan distintas respuestas. Por lo que tú ofreces solucionar el problema eficientemente —para este programa y cual-quier otro que tenga la institución— a cambio de un descuento en la matrícula.

Explica de manera clara y precisa cómo resolverías el problema aplicando las estructuras de datos y algoritmos estudiados este semestre en iic2133.

Respuesta :

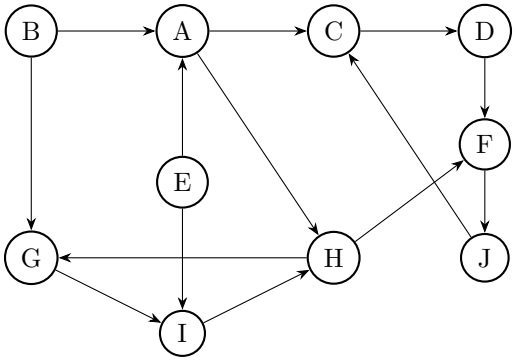
[**2 pts**] Lo primero es plantearse el problema aplicando las estructuras de datos estudiadas; en este caso, un grafo direccional (acíclico, o *DAG*): Un curso es un nodo [**0.7**], y hay una arista direccional del nodo *a* al nodo *b* si *a* es prerrequisito de *b* [**0.7**]; de esta manera, cada programa de la institución es un DAG [**0.6**].

[**2 pts**] Como los estudiantes pueden inscribir todos los cursos que quieran en un mismo período —si cumplen con todos los prerrequisitos correspondientes— entonces el mínimo número de períodos en que puedo terminar un programa es el número de cursos en la secuencia más larga de cursos en que un curso es prerrequisito de otro y éste es prerrequisito de otro y así sucesivamente [**1**]. En la representación del programa como un DAG, la secuencia más larga de cursos correspon-de a la ruta (direccional) con el mayor número de nodos, que también es la ruta con el mayor número de aristas [**1**] (que también es la ruta más larga cuando todas las aristas tienen el mismo costo).

[**2 pts**] Resolvemos el problema ejecutando el algoritmo BFS (estudiado en clase, sólo que ahora es aplicado a un grafo direccional) a partir de cada nodo en el DAG que no tenga aristas que lleguen a él (es decir, a partir de cada nodo que representa a un curso que no tiene prerrequisitos) [**1**]. En cada caso, registramos el valor final del campo δ. Finalmente, nos quedamos con el mayor valor de δ (entre todos estos casos) y le sumamos 1 [**1**] (como δ parte en 0, su valor final en cada caso representa el número de aristas en cada ruta más larga).

4. Grafos

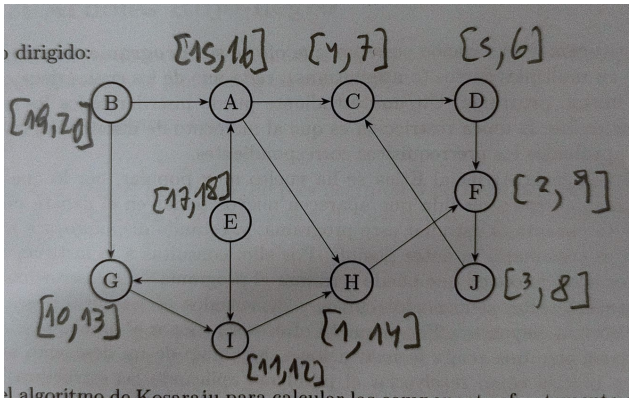
Dado el siguiente grafo dirigido:



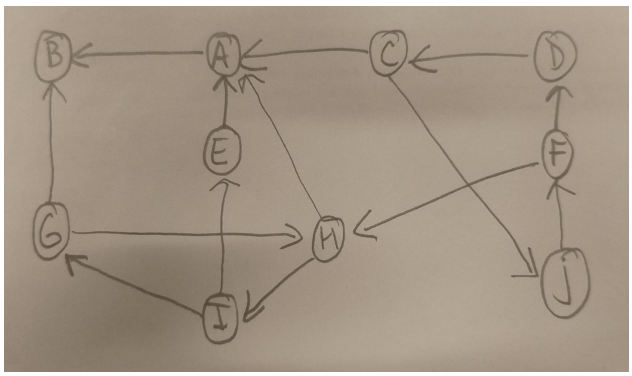
a) (3 pts.) Ejecute el algoritmo de Kosaraju para calcular las componentes fuertemente conectadas. Muestre detalles como: intervalos DFS de los nodos, grafo transpuesto, orden topológico de los vértices, y los representantes (arbitrarios) de cada componente.

Solución:

Los intervalos luego de hacer un BFS comenzando desde H (podían comenzar desde donde sea):



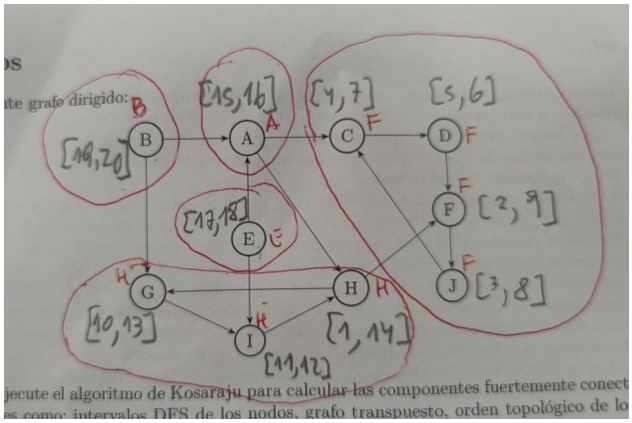
El grafo transpuesto:



El orden de los vértices de acuerdo a valor decreciente del extremo superior de cada intervalo DFS es:

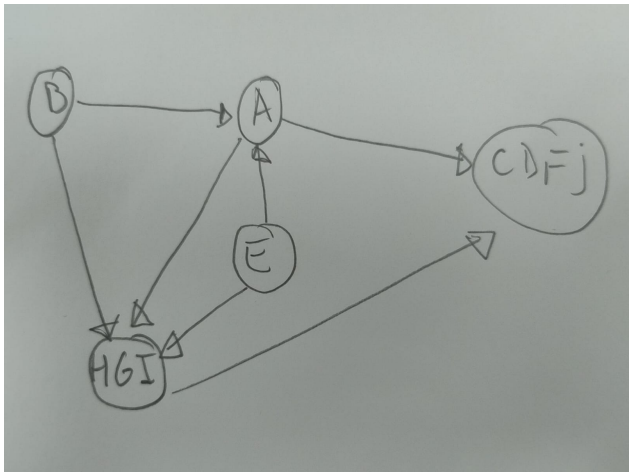
$$B, E, A, H, G, I, F, J, C, D.$$

Siguiendo ese orden y usando el algoritmo visto en clases, genera las siguientes componentes (marcadas en rojo). Junto a cada vértice, en rojo también, se marca el representante de cada componente.



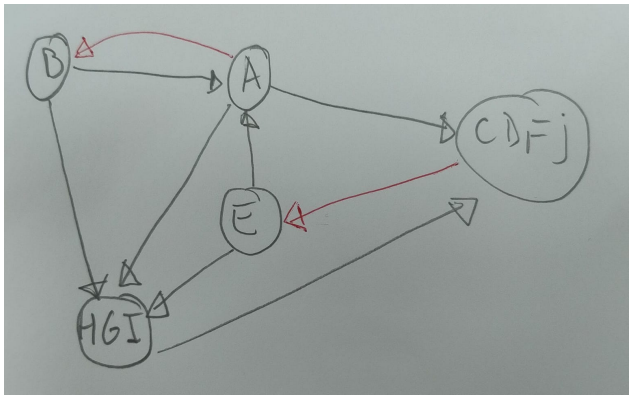
b) (1 pt.) Muestre el grafo de componentes G^{CFC} del grafo.

Solución: El grafo de componentes obtenido a partir del punto anterior es el siguiente (en cada vértice pongo los vértices del grafo original que forman la componente, pero podrían poner cualquier nombre, ojalá el nombre del representante del punto anterior):



c) (2 pts.) ¿Cuál es la cantidad mínima de aristas que se deben agregar al grafo dirigido original para que tenga una única componente fuertemente conectada? Indique qué aristas agregaría y justifique.

Solución: El número mínimo de aristas que deben agregarse para que el grafo tenga una única componente fuertemente conectada (es decir, para que todo par de vértices sea mutuamente alcanzable entre sí) es 2. Esto se podía analizar mejor con el grafo de componentes, en donde hay dos vértices fuente (B y E) y un vértice sumidero (CDFJ). Para que CDFJ deje de ser sumidero y E una fuente, hay que agregar la arista CDFJ → E (alternativamente podían agregar CDFJ → B, también es correcto). Para que B deje de ser una fuente, hay que agregar una arista que llegue a B desde cualquier otro vértice, por ejemplo A → B. Si agrega aristas sin justificar, el puntaje máximo es de 1 punto. Las aristas en rojo son las agregadas:


B
A
CDFJ
E
HGI