Ayudatón 2025-1

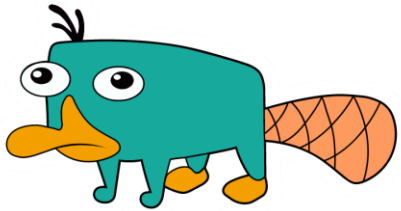
Joaquín Peralta - Elías Ayaach - Alex Infanta - Amelia Gonzalez - Paula Grune

Ordenamiento

I1 2023-1

Se tiene un arreglo A[0 . . . n−1] originalmente ordenado, tal que varios de sus elementos fueron desordenados. Se sabe que en este minuto, al menos un 80 % de sus elementos están en su posición correcta ordenada. En su empresa se piensa usar Quicksort para ordenar nuevamente A.

a) Un ingeniero de software (SI) propone que la elección del pivote sea la mediana entre los valores en los índices 0, n/2 y n − 1 del arreglo A. Indique en qué líneas o zona debe hacer la modificación en la versión de Quicksort vista en clases y especifique el fragmento de pseudocódigo nuevo para incorporar el cambio propuesto.



Solución

El siguiente método se llama en lugar de aquel que genera el pivote aleatorio en Partition (línea 1). El llamado se hace con GetPivot(A, i, f) cuando se hace Partition(A, i, f).

input : Secuencia $A[0, \ldots, n-1]$, índices i, f
output: $\varnothing$

`QuickSort` (A, i, f)**:**

```
1    if i ≤ f :
2        p ← Partition(A, i, f)
3        Quicksort(A, i, p − 1)
4        Quicksort(A, p + 1, f)
```

input : Secuencia $A[0, \ldots, n-1]$, índices i, f
output: Índice del pivote aleatorio en la secuencia ordenada

`GetPivot` (A, i, f)**:**

```
1    p1 ← i, p2 ← ⌊(i + f)/2⌋, p3 ← f
2    if A[p1] ≤ A[p2] ≤ A[p3] ∨ A[p3] ≤ A[p2] ≤ A[p1] :
3        p ← p2
4    elif A[p2] ≤ A[p1] ≤ A[p3] ∨ A[p3] ≤ A[p1] ≤ A[p2] :
5        p ← p1
6    else:
7        p ← p3
8    return p
```

b) Un segundo SI propone que cuando el subarreglo a ordenar sea de tamaño 20 o menos se utilice InsertionSort, ya que si está ordenado, este es su mejor caso. Indique en qué líneas o zona debe hacerse la modificación en la versión de Quicksort vista en clases y especifique el fragmento de pseudocódigo nuevo para incorporar el cambio propuesto

Solución: Se modifica el caso base de Quicksort para que en lugar de revisar el caso base con $f \leq i$, se verifique si $(f - i) > 20$. En tal caso, se ejecuta Quicksort de forma usual. En caso contrario, se hace un llamado a InsertionSort(A, i, f)

b) Un tercer SI propone reemplazar Quicksort por InsertionSort para todo el proceso, afirmando que si el arreglo A está 80 % ordenado, el desempeño de InsertionSort se debe parecer más a su mejor caso, que es mejor que Quicksort. ¿Es correcto lo propuesto? Justifique.

Solución: Si se reemplaza por InsertionSort es importante notar que la fracción de elementos desordenados (invertidos) es proporcional a n, a saber, ≈ 0,2·n. De esta forma, para aquellos elementos, InsertionSort ejecuta una cantidad lineal de veces y en consecuencia, tiene una ejecución O(n^2) de forma global. Dado que esta complejidad se corresponde con la de peor caso de Quicksort, no presenta una ventaja del punto de vista asintótico. Pero cabe notar que en la práctica, dado que la posición final de los pivotes en Quicksort no se puede anticipar, InsertionSort puede presentar una ventaja en los tiempos empíricos.

Tablas de Hash + Heaps

2019-1-C3– Tabla de Hash, Heap

Una tienda quiere premiar a sus k clientes más rentables. Para ello cuenta con la lista de las n compras de los últimos años, en que cada compra es una tupla de la forma (ID cliente, Monto). La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras.
Explique qué estructuras de datos utilizar para resolver este problema en tiempo esperado O(n + n log k)

Solución

Queremos obtener el set de tuplas (ID cliente, Rentabilidad), donde la rentabilidad para el ID Cliente = i es la suma de todos los montos de las tuplas de la forma (i, Monto). Para eto creamos una tabla de hash T donde se almacenan tuplas (ID Cliente, Monto), cada vez que se inserte un ID Cliente:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado
- Sino, se guarda en la tabla con el monto recién insertado como monto inicial

Al hacer esto para las n tuplas, hemos encontrado la rentabilidad para cada cliente

La inserción en esta tabla tiene tiempo esperado O(1) como se ha visto en clases. Como se realizan n inserciones, esta parte tiene tiempo esperado O(n)

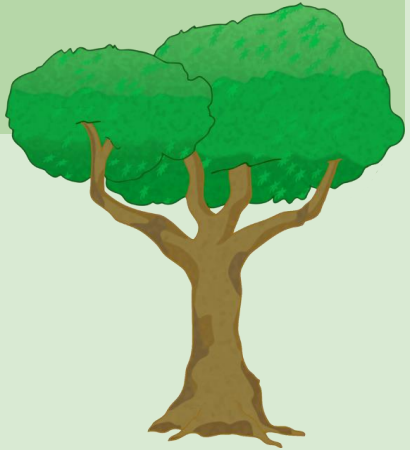
Buscar k clientes más rentables

Sea m el total de clientes distintos. Creamos in min-heap de tamaño k que contendrá los k clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente menos rentable de los k más rentables encontrados hasta el momento.

Iteramos sobres las tuplas en T, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre T. Para cada elemento que veamos, que sea mayor a la raiz, lo intercambiamos con esta. Luego hacemos shift-down para restaurar la propiedad del heap.

Cada inserción en el heap toma O(log k). En el peor caso insertamos los m elementos en el heap, por lo que esta parte es, O(m log k). Pero como en el peor caso m = n, esta parte es O(n log k)

Árboles

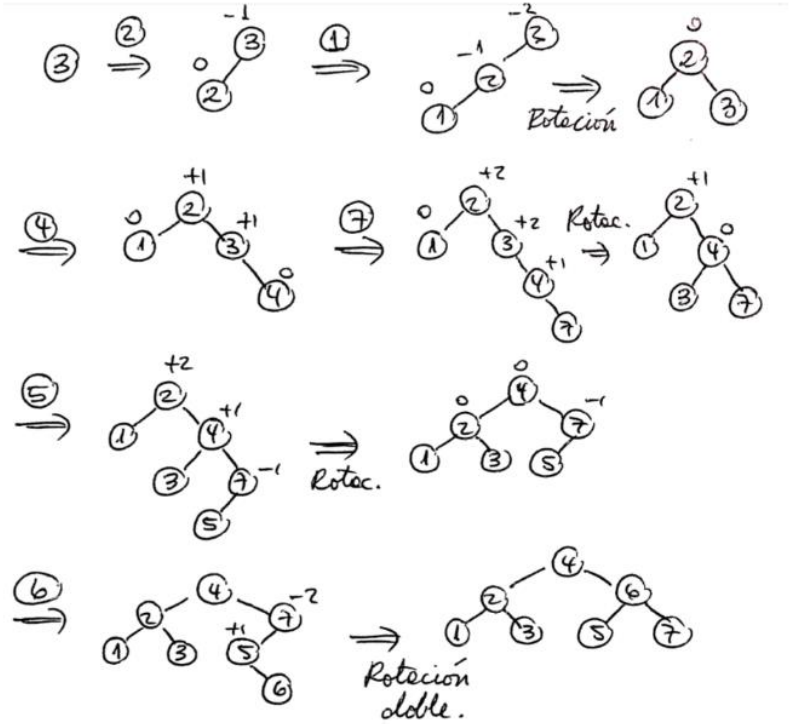


I2 2022-2

A) Muestre un diagrama del proceso de inserción y balance, al insertar las siguientes llaves en un árbol AVL inicialmente vacío:

3 – 2 – 1 – 4 – 7 – 5 - 6

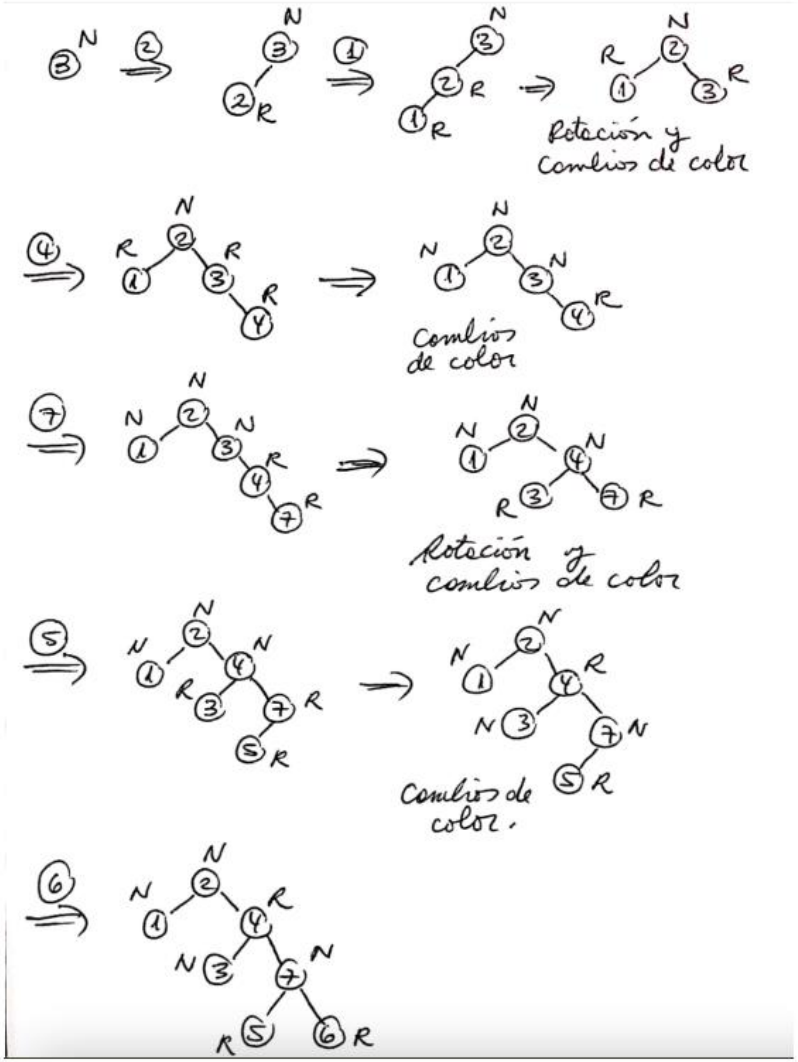
Solución



I2 2022-2

B) Repita el proceso para un árbol rojo-negro inicialmente vacío

Solución


Rotación y
cambios de color
cambios
de color
Rotación y
cambios de color
cambios de
color.

I2 2022-2

c) Sin cambiar aristas, ¿es posible dar una coloración del árbol en (a) de forma que cumpla ser rojo-negro? ¿es posible considerar el árbol en (b) como AVL? Justifique

Solución: El árbol obtenido en (a) admite la coloración en que cada nodo es negro, satisfaciendo las condiciones de rojo-negro. Esto debido a que es perfectamente balanceado y todos los caminos desde un nodo a sus hojas negras descendientes pasa por la misma cantidad de nodos negros.

El árbol obtenido en (b) no es AVL pues la raíz tiene un subárbol izquierdo con altura 1, mientras que el hijo derecho tiene altura 3. La diferencia en mayor a 1 y por lo tanto no se cumple la propiedad AVL.

I2 2022-2

d) si bien la complejidad asintótica de la búsqueda en ambos tipos de árboles es la misma, el tiempo en la práctica puede ser diferente. Dada una misma secuencia de inserciones iniciales, argumente en qué tipo de árbol (AVL o rojo-negro) es más probable que la búsqueda de una llave tarde más.

Solución: Un árbol rojo-negro usa una noción de balance menos exigente que el criterio AVL, pues permite que los sub-árboles hijos de un nodo cualquiera tengan alturas que difieren en más de 1 unidad. Esto causa que la altura del árbol en la práctica pueda ser mayor en rojo-negro que en AVL, lo que hace que la búsqueda pueda ser más lenta en la práctica

Backtracking

2019-1-C4-2–Modelamiento, podas, heurísticas

Sea G(V, E) un grafo no direccional y C ⊆ V un subconjunto de sus vértices. Se dice que C es un kclique si |C| = k y todos vértices de C están conectados con todos los otros vértices de C. Dado un grafo cualquiera G(V, E) y un número k, queremos determinar si existe un k-clique dentro de G. Esto se puede resolver usando backtracking.

2019-1-C4-2–Modelamiento, podas, heurísticas

a)Describe la modelación requerida para aplicar backtracking a este problema: explica cuál es el conjunto de variables, cuáles son sus dominios, y describe en palabras cuáles son las restricciones sobre los valores que pueden tomar dichas variables

b) Propón una poda y explica la modelación a nivel de código requerida para implementarla eficientemente.

c) Propón una heurística para el orden de las variables y explica la modelación a nivel de código requerida para implementarla eficientemente.

Solución

a)Describe la modelación requerida para aplicar backtracking a este problema: explica cuál es el conjunto de variables, cuáles son sus dominios, y describe en palabras cuáles son las restricciones sobre los valores que pueden tomar dichas variables

- Las variables son los vértices de G.
- El dominio es binario: 1 si el vértice está presente en el k-clique y 0 si no.
- Las restricciones son que el vértice asignado como presente en el k-clique (con valor 1 en la variable) tiene que estar conectados a todos los otros vértices que han sido asignados como presente en el k-clique, y el número de vértices asignados como presentes en el grafo debe ser igual a k.

Pseudocódigo

$X = V$

$D = \{0, 1\}$

$R = \{(v, v') \in E \ \forall \ v' \in C\}$

```
is_solvable(X, D, C, k):
        si k = 0 return True
        si X = {} return False
        x ← alguna variable de X
        para v ∈ D:
                si v = 1:
                        si v no cumple R, continuar
                        si is_solvable(X - {x}, D, C ∪ {x} , k  - 1):
                                retornar True
                        C = C - {x}
        si v = 0:
                si is_solvable(X - {x}, D, C, k):
                        retornar True

retornar False
```

¿Como deberíamos hacer la llamada a la función?

```
is_solvable(X, D, C, k):
        si k = 0 return True
        si X = {} return False
        x ← alguna variable de X
        para v ϵ D:
                si v = 1:
                        si v no cumple R, continuar
                        si is_solvable(X - {x}, D, C ∪ {x} , k  - 1):
                                retornar True
                        C = C - {x}
                si v = 0:
                        si is_solvable(X - {x}, D, C, k):
                                retornar True

        retornar False
```

$X = V$

$D = \{0, 1\}$

$R = \{(v, v') \in E \ \forall \ v' \in C\}$

¿Como deberíamos hacer la llamada a la función?

```
is_solvable(X, D, C, k):
        si k = 0 return True
        si X = {} return False
        x ← alguna variable de X
        para v ϵ D:
                si v = 1:
                        si v no cumple R, continuar
                        si is_solvable(X - {x}, D, C ∪ {x} , k  - 1):
                                retornar True
                        C = C - {x}
                si v = 0:
                        si is_solvable(X - {x}, D, C, k):
                                retornar True

        retornar False
```

$X = V$

$D = \{0, 1\}$

$R = \{(v, v') \in E \ \forall \ v' \in C\}$

- **is_solvable({V, D, {}, k)**

Solución

b) Propón una poda y explica la modelación a nivel de código requerida para implementarla eficientemente.

Solución

b) Propón una poda y explica la modelación a nivel de código requerida para implementarla eficientemente.

- Si el vértice que estamos revisando tiene un número de aristas tiene menos de k - 1 aristas que se conectan con él, es imposible que éste pertenezca al k-clique. Para implementarla, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta acceder a esta variable y si es menor a k - 1, se detiene la ejecución.

- Si quedan menos variables a asignar, que el valor de k, podemos terminar esa ejecución ya que es imposible agregar suficientes elementos a C para que pertenezcan al kclique. Para esto, podemos llevar un contador de cuántas variables nos quedan por asignar, y cuántos elementos llevamos en nuestro k-clique. Si la cantidad de variables que nos queda por asignar es menor a (k - (N° elementos asignados que llevamos en k-clique)), detenemos la ejecución.

Solución

c) Propón una heurística para el orden de las variables y explica la modelación a nivel de código requerida para implementarla eficientemente.

Solución

c) Propón una heurística para el orden de las variables y explica la modelación a nivel de código requerida para implementarla eficientemente.

- Podemos ordenar los vértices de mayor a menor en función de la cantidad de aristas que poseen. De esta manera, es más probable que éstos sean parte de algún k-clique. Para esto, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta con ordenar este arreglo de vértices en función de la cantidad de aristas hacia otros nodos, mediante algún algoritmo eficiente de los que se han visto en clases (como MergeSort, QuickSort, etc.)

Greedy

2018-2-Ex-P3–Demostración de estrategia codiciosa

Supongamos que quieres hacer una caminata por un parque nacional, que te va a tomar varios días. Tu estrategia es caminar lo más posible de día, pero acampar cuando oscurece. El mapa proporcionado por la oficina de turismo muestra muchos buenos lugares para acampar, y tú has decidido que cada vez que llegues a uno de estos lugares vas a calcular (correctamente) si alcanzas a llegar al próximo antes de que oscurezca; en caso afirmativo, sigues; de lo contrario, te detienes y acampas. Argumenta rigurosamente que esta estrategia minimiza el número de detenciones para acampar que vas a tener que hacer.

Solución

Por contradicción

Primero, algunos supuestos y definiciones. Sean L la longitud total del camino, d la distancia que puedes caminar en un día, y x1, x2, … xn los puntos de detención que muestra el mapa (distancias desde la entrada del camino). Decimos que un conjunto de puntos de detención es válido si:

- la distancia entre cualquier par de puntos adyacentes (en el conjunto) es a lo más d,
- la distancia desde la entrada al primer punto es a lo más d, y
- la distancia desde el último punto a la salida del camino es a lo más d.

Solución

Sean $R = \{x_{p1}, x_{p2}, \ldots, x_{pk}\}$ el conjunto (válido) de puntos de detención elegidos por tu estrategia codiciosa, y $S = \{x_{q1}, x_{q2}, \ldots, x_{qm}\}$, con $m < k$, un conjunto válido de menos puntos que R.

Primero, los puntos de R están más lejos que los puntos de S; es decir, para cada $j = 1, 2, \ldots, m$, tenemos que $x_{pj} \geq x_{qj}$, lo que demostramos por inducción sobre j.

- Para $j = 1$, sale de la definición de la estrategia codiciosa: tú viajas lo más posible el primer día antes de detenerte.
- Para $j > 1$ y suponiendo que la afirmación es válida para todo $i < j$: $x_{qj} - x_{qj-1} \leq d$, y también $x_{qj} - x_{qj-1} \geq x_{qj} - x_{pj-1}$, lo que implica que $x_{qj} - x_{pj-1} \leq d$. Es decir, una vez que tú sales de x_{pj-1}, podrías caminar hasta x_{qj} en un día, por lo que x_{pj}, que es donde finalmente vuelves a detenerte, solo puede estar más lejos que x_{qj}.

Segundo, como $m < k$, entonces $x_{pm} < L - d$; de lo contrario no tendría sentido que te detengas en x_{pm+1}. Como además $x_{qm} \leq x_{pm}$, por la demostración anterior, resulta que $x_{qm} < L - d$, lo que contradice que S sea un conjunto válido de puntos de detención.

Programación Dinámica

I2 2024-1

Suponga que en una sala se han programado n charlas c1,...,cn, cada una de las cuales dura exactamente una hora y tal que cada charla c_{i + 1} comienza apenas termina la charla anterior c_i. La asistencia a una charla c_i (1 <= i <= n) entrega un puntaje p_i (representado por un entero positivo) pero produce tanto cansancio que es imposible asistir a las siguientes d_i charlas, teniendo que descansar. Alice quiere asistir a las charlas y sumar la máxima cantidad de puntaje posible, respetando las reglas de descanso indicadas.

I2 2024-1

a) Dados los valores pi,...,pn y di,...,dn para las n charlas, escriba una ecuación de recurrencia para la función S(i), la que calcula (de forma exhaustiva) el puntaje máximo que se puede obtener con las charlas ci,...,cn. Indique claramente los casos base necesarios para la correcta definición de la función.

Solución: Para $1 \leq i \leq n$ definimos:

$$S(i) = \begin{cases} 0, & i > n \\ \max\{S(i+1), S(i+d_i+1)+p_i\}, & i \leq n. \end{cases}$$

Claramente, la solución al problema original se obtiene con $S(1)$. La versión pseudocódigo es:

Algoritmo 1: $S(i)$

```
if i > n then
    return 0
else
    return max{S(i+1), S(i+d_i+1)+p_i}
```

I2 2024-1

b) Implemente el algoritmo del punto anterior usando programación dinámica, tal que evite recalcular subproblemas antes calculados

Solución: En este caso se mantiene un arreglo M[1...n] con todas sus entradas inicializadas con ⌀

Algoritmo 2: $SPD(i)$

```
if i > n then
    return 0
else
    if M[i] = ∅ then
        M[i] ← máx {SPD(i + 1), SPD(i + d_i + 1) + p_i}
    return M[i]
```

Dijkstra + Bellman Ford

I3 2024-1

Un agujero de gusano es un túnel que atraviesa el espacio y el tiempo y conecta dos sistemas estelares. Los agujeros de gusano tienen algunas propiedades peculiares:

- Cada agujero de gusano es de un único sentido y tiene dos puntos extremos, uno llamado extremo origen (que es donde comienza el agujero de gusano) y el otro llamado extremo destino (que es donde termina el agujero de gusano). Ambos extremos están situados en sistemas estelares distintos.
- El tiempo que se tarda en atravesar un agujero de gusano es insignificante.
- Un sistema estelar puede ser el punto extremo (origen o destino) de más de un agujero de gusano.
- Por alguna razón desconocida, a partir de nuestro sistema solar, siempre es posible terminar en cualquier sistema estelar siguiendo una secuencia de agujeros de gusano (quizás la Tierra sea el centro del universo).
- Entre cualquier par de sistemas estelares, hay como máximo un agujero de gusano en cualquier dirección.

I3 2024-1

Todos los agujeros de gusano tienen una diferencia de tiempo constante entre sus puntos extremos, medido en años usando un valor entero. Por ejemplo, un agujero de gusano específico puede hacer que la persona que lo atraviese viaje X años en el futuro. Otro agujero de gusano puede hacer que la persona viaje Y años en el pasado. Dado que se asume que el universo se originó con el Big Bang, diremos que ése es el tiempo 0. Ningún agujero de gusano puede hacernos retroceder a un instante anterior al Big Bang: si lo intentamos, quedaremos en el tiempo 0 . Alice, una brillante doctora en física, quiere utilizar agujeros de gusano para viajar en el tiempo y estudiar diversos fenómenos.

I3 2024-1

(a) Modele esta realidad usando grafos, indicando todas las características que considere necesarias, de manera de poder resolver el problema planteado en el item (b).

Solución: Deben modelar la situación usando un grafo dirigido. Los nodos son los sistemas estelares, las aristas dirigidas son los agujeros de gusano. Las aristas deben tener peso representado por un número entero, indicando la diferencia en años que se produce al seguir un agujero de gusano. El valor del entero será positivo si el agujero de gusano permite viajar en el futuro, o negativo si se viaja al pasado.

I3 2024-1

(b) Escriba un algoritmo eficiente que, a partir de la representación de un conjunto de sistemas estelares y sus correspondientes agujeros de gusano, le permita a Alice determinar si es posible (o no) viajar a ver el Big Bang. Además, en caso afirmativo, su algoritmo debe indicar al menos una secuencia de agujeros de gusano tal que si se los atravieza, permitan ese viaje.

Solución: La idea es determinar si el grafo tiene un ciclo negativo o no, usando el algoritmo de Bellman-Ford, modificado como a continuación para poder obtener el ciclo negativo:

Solución

```
BigBang(s):
    for u ∈ V :
        d[u] ← ∞;  π[u] ← ∅
    d[s] ← 0
    for k = 1 ... |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + c(u, v) then
                d[v] ← d[u] + c(u, v)
                π[v] ← u
    if d[v] > d[u] + c(u, v) then
        v' ← u
        L ← ∅
        while v' ≠ v :
            L.append(v')
            v' ← π[v']
        return (true, L)
    return (false, ∅)
```

I3 2024-1

(c) ¿Cómo modificaría lo hecho en los items anteriores para obtener un algoritmo que permita determinar si es posible o no avanzar de forma indefinida en el futuro? Indique los detalles (aunque no es necesario escribir pseudocódigo).

Solución: Bellman-Ford no sirve en este caso, porque lo que hay que hacer es detectar un ciclo positivo. Sin embargo, se pueden modificar los pesos de las aristas, multiplicando los valores originales por -1 , y así aplicar el algoritmo del item (b).

MST

2023-2-I3-P3–Árboles de cobertura

(a) Considere un grafo no dirigido G y un árbol de cobertura de costo mínimo T para G. Suponga que se añade un conjunto de k nodos adicionales con ciertas aristas, resultando en un grafo conexo G′ . Explique cómo obtener un MST para G′ a partir de T e indique la complejidad de su propuesta. No necesita entregar pseudocódigo.

Solución

(a) Considere un grafo no dirigido G y un árbol de cobertura de costo mínimo T para G. Suponga que se añade un conjunto de k nodos adicionales con ciertas aristas, resultando en un grafo conexo G′ . Explique cómo obtener un MST para G′ a partir de T e indique la complejidad de su propuesta. No necesita entregar pseudocódigo.

- Sean E′ y V′ los conjuntos de aristas y nodos nuevos para agregar a G. Para decidir qué aristas de E′ se deben incluir para extender el árbol T, podemos usar el algoritmo de Prim, escogiendo la arista de menor costo que conecta con un nodo no conectado previamente a T. En definitiva, es continuar el proceso normal del algoritmo de Prim. La complejidad de este algoritmo es entonces O(E′ log(V′)) si consideramos que la cardinalidad de los conjuntos mencionados es arbitraria.

2023-2-I3-P3–Árboles de cobertura

(b) Se propone el siguiente algoritmo con la intención de obtener un MST de un grafo G. Observe que la línea 2 toma aristas de E en orden arbitrario.
(i) Describa qué estructuras de datos utilizar para implementar este algoritmo de forma eficiente. Indique la complejidad de CasiKruskal al usar dichas estructuras.

```
CasiKruskal(G):
1    T ← lista vacía
2    for e ∈ E :
3        if Agregar e a T no forma ciclo :
4            T ← T ∪ {e}
5    return T
```

Solución

(i) Describa qué estructuras de datos utilizar para implementar este algoritmo de forma eficiente. Indique la complejidad de CasiKruskal al usar dichas estructuras.

```
  CasiKruskal(G):
1     T ← lista vacía
2     for e ∈ E :
3         if Agregar e a T no forma ciclo :
4             T ← T ∪ {e}
5     return T
```

- La línea 3 corresponde a la operación más costosa de este algoritmo. Para determinar eficientemente si agregar una arista forma o no un ciclo, podemos usar la estructura de conjunto disjunto para almacenar los conjuntos de nodos que están conectados, tal como en Kruskal.

2023-2-I3-P3–Árboles de cobertura

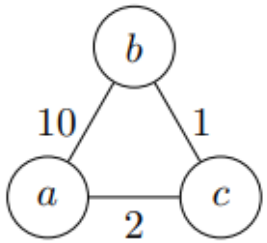
(ii) Demuestre que CasiKruskal no retorna un MST para todo grafo G.

```
CasiKruskal(G):
1    T ← lista vacía
2    for e ∈ E :
3        if Agregar e a T no forma ciclo :
4            T ← T ∪ {e}
5    return T
```

Solución

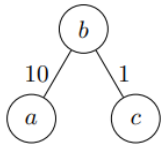
(ii) Demuestre que CasiKruskal no retorna un MST para todo grafo G.

Consideremos el grafo $G = (\{a, b, c\}, \{\{a, b\}, \{b, c\}, \{a, c\}\})$ con los siguientes costos



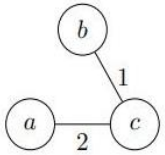
Solución

Si el orden de iteración de aristas en la linea 2 del algoritmo es {a, b}, {b, c}, {a, c} el árbol obtenido por CasiKruskal es



costo 11

Pero sabemos que el único árbol de cobertura de costo minimo para G es



costo 3

Kosaraju - CFC

2024-2-I2

Queremos saber si estando en una estación del metro, puedo ir a cualquier otra estación usando únicamente el metro, para esto representamos la red del metro como un grafo direccional de la siguiente manera: cada estación es un vértice y si en una misma linea del metro, la estación v es la próxima estación a la estación u, entonces (y solo entonces) incluimos una arista direccional.

a) Describe un algoritmo eficiente en pseudocódigo que permita responder la pregunta original. El algoritmo sólo puede hacer usa de DFS estudiado en clases

Solución

Si representamos la red del metro como se describe arriba, entonces la respuesta a la pregunta inicial es "sí" si y sólo si el grafo es una sola gran componente fuertemente conexa.

Por lo tanto, el algoritmo que se pide es el algoritmo de componentes fuertemente conexas (CFCs) con la única particularidad de que al terminar hay que ver cuántas CFCs encontró: si encontró exactamente una, entonces el algoritmo imprime "sí", en cualquier otros caso, "no".

Cuando el enunciado dice "tu algoritmo puede hacer uso sólo del algoritmo DFS estudiado en clases," quiere decir que hay que escribir el algoritmo con un nivel de detalle similar al usado en clases (que hace uso de DFS), y hay que agregarle la condición final; no se puede poner como respuesta, "el algoritmo CFC estudiado en clases."

Solución

```
// Función auxiliar para llenar la pila
DFS_FillStack(G, u, visitado, pila):
visitado[u] ← true
for cada vértice v adyacente a u en G:
if not visitado[v]:
DFS_FillStack(G, v, visitado, pila)
pila.push(u) // Agregar después de procesar adyacentes
```

```
// Función auxiliar para crear grafo transpuesto
CreateTranspose(G):
G_transpuesto ← grafo vacío con los mismos vértices que G
for cada vértice u en V(G):
for cada vértice v adyacente a u en G:
// Invertir la dirección de la arista
agregar arista (v, u) a G_transpuesto
return G_transpuesto
```

```
// Función auxiliar para procesar una componente
DFS_PrintComponent(G_transpuesto, u, visitado, numero_CFC):
visitado[u] ← true
print("Estación " + u + " pertenece a CFC " + numero_CFC)
for cada vértice v adyacente a u en G_transpuesto:
if not visitado[v]:
DFS_PrintComponent(G_transpuesto, v, visitado, numero_CFC)
```

Solución

Algoritmo CFC para Red de Metro

```
// Input: grafo dirigido G que representa la red del metro
// Output: número de CFCs encontradas
MetroStronglyConnectedComponents(G):
    Paso 1: Realizar DFS en el grafo original
    visitado ← arreglo de booleanos inicializado en false
    pila ← pila vacía
    for cada vértice u en V(G):
        if not visitado[u]:
            DFS_FillStack(G, u, visitado, pila)
    Paso 2: Crear el grafo transpuesto
    G transpuesto ← CreateTranspose(G)
    Paso 3: DFS en grafo transpuesto
    visitado ← reinicializar a false
    contador_CFCs ← 0
    while pila no está vacía:
        u ← pila.pop()
        if not visitado[u]:
            contador_CFCs ← contador_CFCs + 1
            DFS_PrintComponent(G_transpuesto, u, visitado, contador_CFCs)
    return contador_CFCs
```

Solución

```
// Algoritmo principal para responder la pregunta
AlgoritmoMetroCFC(G):
num_CFCs ← MetroStronglyConnectedComponents(G)
if num_CFCs == 1:
return "sí" // Se puede ir a cualquier estación
else:
return "no" // No se puede ir a todas las estaciones
```

Complejidad: O(V + E) - lineal en el tamaño del grafo
Espacio: O(V) para las estructuras auxiliares

2024-2-I2

b) Justifica que tu algoritmo es eficiente

Solución: El algoritmo básicamente hace dos recorridos BFS del mismo grafo, primero, sobre la versión original del grafo, y luego sobre la versión transpuesta. Luego es O(V + E), es decir, es lineal en el "tamaño" del grafo: números de vértices y números de aristas.

Éxito en el examen