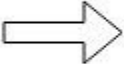


1. Backtracking

Un tablero de *kakuro square* es una grilla de $N \times M$ donde cada posición tiene una celda que puede ser rellenada por un número natural distinto de 0. Además, cada fila y columna tiene un número natural que indica el valor que debe tener la suma de los números correspondientes a la fila o columna. P.ej.:

	23	15	12
23			
17			
10			



	23	15	12
23	9	8	6
17	8	4	5
10	6	3	1

El problema de rellenar la grilla cumpliendo con las restricciones se puede resolver utilizando la estrategia de backtracking.

- Identifica las variables del problema y sus respectivos dominios.
- Explica con tus palabras cómo se podría revisar en tiempo $O(1)$ si la asignación de una variable rompe una restricción.
- Explica con tus palabras una poda o una heurística que se pueda aplicar para resolver este problema más eficientemente.

Respuesta:

- a) Para resolver con backtracking este problema se debe asignar a cada celda un número, por lo que las variables son las celdas [1pto].

En teoría cada celda puede tener cualquier número natural, pero ya que no puede haber un dominio infinito para resolverlo con backtracking es necesario acotar los dominios. Un dominio acotado correcto puede ser los números del 1 al MAX(restricción). Ya que se sabe que nunca un número puede ser mayor al valor de su restricción [1pto].

- b) Se puede mantener un contador por cada restricción del problema que cuente la suma actual de cada fila o columna. De esta manera se inmediatamente si rompo la restricción cuando el contador supera la restricción [2pts].

- c) [2pts]

Podas: Ver que los contadores descritos en b) no superen el máximo antes de tener completa la fila o columna. Ver que la suma de una fila o columna actual más el número de casillas vacías de la fila o columna no sea mayor a la restricción (ya que no se pueden asignar ceros). Probablemente hay mas podas.

Heurísticas: Probar primero las celdas que pertenecen a filas o columnas con más valores ya asignados (o menos valores por asignar). También pueden haber muchas más.

3. Búsqueda en profundidad (DFS)

La siguiente es una versión más general del algoritmo visto en clases para recorrer un grafo G a partir de un vértice v :

```
DFS( $G, v$ ):
  marcar  $v$ 
   $S \models v$       —inicializa un stack de vértices,  $S$ , con  $v$ 
  while  $S$  no está vacío :
     $w \models S$  —extrae el vértice en el top de  $S$  y asígnalo a  $w$ 
    “visitar”  $w$ 
    for all  $x$  tal que  $(w, x)$  es una arista de  $G$  :
      if  $x$  no está marcado:
        marcar  $x$ 
         $S \models x$ 
```

a) Supongamos que G es no direccional (las aristas no tienen dirección). Decimos que G es **biconectado** si no hay ningún vértice que al ser sacado de G desconecta el resto del grafo. Por el contrario, si G no es biconectado, entonces los vértices que al ser sacados de G desconectan el grafo se conocen co-mo **puntos de articulación**.

Describe un algoritmo eficiente para encontrar todos los puntos de articulación en un grafo conectado. Explica cuál es la complejidad de tu algoritmo.

Respuesta.

Para obtener un algoritmo eficiente que resuelve este problema, utilizamos una versión adaptada de DFS que pueda detectar aristas hacia atrás en el grafo no direccional. El principio sobre el que se basa la solución es que luego de hacer DFS en un grafo no dirigido hay dos posibilidades para que un vértice sea punto de articulación:

1. La raíz del árbol DFS generado es un punto de articulación: esto ocurre cuando la raíz tiene más de un hijo, pues significa que la única forma de descubrir todos estos hijos, fue partir una nueva rama de la búsqueda desde la raíz. Si no fuera así y la raíz no es punto de articulación, entonces al sacarla los hijos deberían tener otro camino que los una. *Este será el primer chequeo: verificar si la raíz tiene más de un hijo en el árbol DFS.* [1 pt. Por caso raíz]
2. Un vértice que no es raíz es punto de articulación: esto ocurre si para un nodo u todos sus descendientes no cuentan con un camino que llegue a un antecesor de u en el árbol DFS. Esto se puede verificar en tiempo constante si se agrega un atributo a cada nodo que se define según

$$u.low = \min\{u.disc, w.low \text{ para algún } w \text{ descendiente de } u\}$$

Este atributo guarda el tiempo de descubrimiento de u o el de un ancestro alcanzable con alguna arista hacia atrás. Con esta estrategia, los puntos de articulación (no raíz) son aquellos que tienen el atributo *low* menor que el de sus descendientes directos. [1 pt. Por estrategia general]

Con esto, el algoritmo DFS que incorpora el atributo *low* a cada vértice seguido de una revisión para cada vértice permite entregar los puntos de articulación de todo el grafo. En resumen:

1. Ejecutar DFS que registra tiempos de descubrimiento y *low* (paso completo en $O(V+E)$)
2. Revisar cada vértice en el árbol DFS generado y comparar *low* con sus descendientes. Agregar a una lista aquellos vértices que no cumplan la condición buscada (paso completo en $O(V+E)$).
3. Revisar cuántos hijos tiene la raíz del árbol y agregarla si tiene más de uno (paso completo en $O(1)$).

Por lo tanto, la complejidad del algoritmo es $O(V+E)$ para un grafo representado con listas de adyacencia.

[1 pt. Por explicación de complejidad]

b) Considera el siguiente grafo G direccional (las aristas tienen dirección), representado mediante sus listas de adyacencias:

[0]: 5-1	[1]:	[2]: 0-3	[3]: 5-2	[4]: 3-2	[5]: 4	
[6]: 9-4-0	[7]: 6-8	[8]: 7-9	[9]: 11-10	[10]: 12	[11]: 4-12	[12]: 9

Ejecuta el algoritmo DFS anterior a partir del vértice $v = 0$. Muestra el orden en que los vértices van siendo marcados y el contenido del stack S cada vez que cambia.

Respuesta (posible solución).

Partiendo de 0, lo marcamos y lo ponemos en S ; $S = [0]$.

[A partir de ahora, cada paso = **0.5 pts.**]

Al sacar 0 ($S = []$) y asignarlo a w , vemos que sus vecinos son 5 y 1; los marcamos y los ponemos en S ; $S = [5, 1]$.

Al sacar 5 ($S = [1]$) y asignarlo a w , vemos que su único vecino es 4; lo marcamos y lo ponemos en S ; $S = [4, 1]$.

Al sacar 4 ($S = [1]$) y asignarlo a w , vemos que sus vecinos son 3 y 2; los marcamos y los ponemos en S ; $S = [3, 2, 1]$.

Al sacar 3 ($S = [2, 1]$) y asignarlo a w , vemos que sus vecinos son 5 y 2, ambos ya marcados; $S = [2, 1]$.

Al sacar 2 ($S = [1]$) y asignarlo a w , vemos que sus vecinos son 0 y 3, ambos ya marcados; $S = [1]$.

Al sacar 1 ($S = []$) y asignarlo a w , vemos que no tiene vecinos; $S = []$ y terminamos.

4. Dos grafos G_1 y G_2 se dicen *isomorfos* si existe una función biyectiva f tal que $f(x) = y$ si y sólo si x y y son adyacentes en G_1 si y sólo si $f(x)$ y $f(y)$ son adyacentes en G_2 . Escribe un algoritmo que determine si dos grafos son isomorfos. Considera que si dos grafos son isomorfos, al eliminar un vértice x de G_1 (y las aristas de x) y su vértice correspondiente (y aristas) en G_2 , los grafos que quedan también son isomorfos.

Respuesta

F = diccionario inicialmente vacío

backtracking(Grafo1, Grafo2):

// Caso base 1 pto

If (grafo1.nodos = grafo2.nodos = vacío) return true

// Asignar los nodos de un grafo los nodos del otro 2 pts

n = grafo1.nodos.pop(0)

For nodo in grafo2.nodos:

F(n) = nodo

// Restriccion 2 pts

If (cumple_restriccion(n, nodo))

// Hacer bien caso recursivo y UNDO 1 pto

If (backtracking(grafo1 sin n, grafo2 sin nodo) return true

F(n) = null

Return false

cumple_restricciones(nodo1, nodo2):

If (|nodo1.vecinos| != |nodo2.vecinos|) return false

For v in nodo1.vecinos:

If (v esta en F):

If (F(v) esta en nodo2.vecinos):

Return false

Return true

EX 2017-1

2. Dadas una secuencia de números enteros positivos no necesariamente distintos $\mathbf{x} = x_1, \dots, x_n$ y una secuencia de bits $\mathbf{b} = b_1, \dots, b_n$, se define la función:

$$\mathbf{x} \otimes \mathbf{b} = \sum_{i=1}^n (-1)^{b_i} x_i$$

a) Dé el pseudocódigo de una función que utilice backtracking y que reciba una secuencia de números \mathbf{x} y un número N , y retorne *true* cuando existe un \mathbf{b} tal que $\mathbf{x} \otimes \mathbf{b} = N$, y que retorne *false* en caso contrario.

Respuesta: 1 function $f(i, s)$

b) Dé un pseudocódigo iterativo para la función de a) que sea **mucho más** eficiente que su implementación de b). Analice su tiempo de ejecución.

I1 2016-2

1. Dado un mapa, queremos colorear las regiones de manera tal que nunca dos regiones adyacentes (límites) tengan el mismo color, usando a lo más cuatro colores. Para esto, conviene representar el mapa como un grafo, en que los nodos corresponden a regiones y hay una arista entre dos nodos si y sólo si las dos regiones correspondientes son adyacentes. Generalizando, el problema de colorear un grafo consiste en determinar todas las formas diferentes en que un grafo dado puede ser coloreado usando a lo más m colores. Escribe un algoritmo de **backtracking** para resolver este problema.

Suponiendo que el grafo tiene n nodos, representamos los nodos simplemente por los números $1, 2, \dots, n$; y representamos las aristas por una matriz G tal que $G[i][j] = 1$ si y sólo si hay una arista entre los nodos i y j , y $G[i][j] = 0$ en otro caso. Además, representamos los m colores por los números $1, 2, \dots, m$, y las soluciones por las tuplas (x_1, \dots, x_n) , en que x_i es el color del nodo i .

Se sugiere escribir un ciclo principal, en que asignan todos los colores válidos para el nodo k ; cada vez que se asigna un nuevo color al nodo k , este ciclo principal se llama recursivamente para asignar todos los colores válidos para el nodo $k+1$. Y se sugiere escribir otro ciclo para hacer la asignación de un color a un nodo; este ciclo asigna el "color" 0 si no puede asignar un color válido.

Respuesta:

Como se deduce del enunciado, sigue el patrón del problema de las 8 reinas.

Inicializamos el vector x en ceros.

Primero, asignamos el color 1 al vértice 1: $x[1] = 1$; y tratamos de asignar, recursivamente y uno por uno, todos los colores válidos para el vértice 2. En este proceso, cada vez que asignamos un color válido al vértice 2, tratamos de asignar, recursivamente y uno por uno, todos los colores válidos para el vértice 3; etc. Si no podemos asignar un color válido al vértice k , entonces tenemos que cambiar la última asignación de color que hicimos al vértice $k-1$. Cuando asignamos un color válido al vértice n , imprimimos el vector x .

En código:

```
colorear(k):
    repeat:
        proxColor(k)
        if x[k] == 0: return
        if k == n: print(x)
        else: colorear(k+1)
    until False

proxColor(k):
    repeat:
        x[k] = x[k]+1 % m+1
        if x[k] == 0: return
        for j = 1 ... n:
            if G[k,j] != 0 and x[k] == x[j]: break
        if j == n+1: return
    until False
```

I2 2016-2

1. Un *punte* en un grafo no direccional es una arista que, si se saca, separaría el grafo en dos subgrafos disjuntos. Describe un algoritmo eficiente que encuentre todos los puentes de un grafo no direccional; justifica la eficiencia y la corrección de tu algoritmo.

Respuesta:

Una arista (u, v) es un puente $\hat{U}(u, v)$ no pertenece a un ciclo; dfs encuentra los ciclos. Por lo tanto, ejecutemos dfs y miremos el bosque dfs producido. Recordemos que dfs en un grafo no direccional sólo produce aristas de árbol y hacia atrás; **no** produce aristas hacia adelante ni cruzadas.

Una arista de árbol (u, v) en el bosque dfs es un puente \hat{U} no hay aristas hacia atrás que conecten un descendiente de v a un ancestro de u . ¿Cómo determinamos esto? Recordemos el tiempo de descubrimiento $v.d$ que dfs asigna a cada vértice v : una arista hacia atrás (x, y) conectará un descendiente x de y a un ancestro y de u si $y.d < x.d$.

Luego, si $v.d$ es menor que todos los tiempos d que pueden ser alcanzados mediante una arista hacia atrás desde cualquier descendiente de v , entonces (u, v) es un puente.

Es decir, basta modificar un poco `dfsVisit` visto en clase: asignar a cada vértice v un tercer número $v.min$ que sea el menor de todos los $y.d$ alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar `dfsVisit(v)`, $v.d = v.min$, entonces (u, v) es un puente. Esta modificación no cambia la complejidad de `dfsVisit`.

El puntaje distribuye así:

- 4 ptos por el algoritmo (Se toma en cuenta cuán eficiente es).
- 1 pto por justificar correctamente la eficiencia, independiente que tan bueno sea el algoritmo y suponiendo que este es correcto.
- 1 pto por justificar la correctitud del algoritmo.

I2 2015-2

2. a) Considera un grafo no direccional. Queremos pintar los vértices del grafo, ya sea de azul o de amarillo, pero de modo que **dos vértices conectados por una misma arista queden pintados de colores distintos**.

Da un algoritmo eficiente que pinte los vértices del grafo según la regla anterior, o bien que se dé cuenta de que no se puede; justifica la corrección de su algoritmo.

La idea es usar DFS, que es $O(V+E)$:

- 1) *Recordemos que al aplicar DFS a un grafo no direccional sólo aparecen aristas de árbol y hacia atrás; no hay aristas hacia adelante ni cruzadas.*
- 2) *Por lo tanto, en el árbol DFS resultante, si a partir de un vértice v surgen varias ramas, significa que en el grafo original cualquier ruta que va de los vértices en una de esas ramas a los vértices en otra necesariamente pasa por v .*
- 3) *Por lo tanto, a partir de la raíz del árbol DFS podemos pintar los vértices alternando colores a medida que bajamos por cada rama.*
- 4) *Finalmente, hay que revisar las aristas hacia atrás del árbol DFS, cuya presencia refleja la existencia de ciclos en el grafo original:*
 - *Si alguna arista hacia atrás conecta (directamente) vértices pintados del mismo color, entonces el grafo **no se puede pintar** como se pide en el enunciado.*
 - *Pero si el árbol DFS no tiene aristas hacia atrás, o si ninguna arista hacia atrás conecta vértices pintados del mismo color, entonces es posible pintar los vértices del grafo original como se pide en el enunciado; la asignación de colores definida en el paso 3) es una forma concreta de hacerlo.*

b) Considera el siguiente grafo direccional G , representado mediante listas de vértices adyacentes:

$[0] \rightarrow 5, 1$	$[4] \rightarrow 3, 2$	$[8] \rightarrow 7, 9$	$[12] \rightarrow 9$
$[1] \rightarrow$	$[5] \rightarrow 4$	$[9] \rightarrow 11, 10$	
$[2] \rightarrow 0, 3$	$[6] \rightarrow 9, 4, 0$	$[10] \rightarrow 12$	
$[3] \rightarrow 5, 2$	$[7] \rightarrow 6, 8$	$[11] \rightarrow 4, 12$	

Determina las componentes fuertemente conectadas de G **ejecutando el algoritmo estudiado en clase**:

- 1) Realizamos DFS de G , para calcular los tiempos de finalización, $u.f$, de cada vértice
- 2) Determinamos G^T
- 3) Realizamos DFS de G^T , pero en el ciclo principal consideramos los vértices en orden decreciente de $u.f$ calculado antes

El DFS del **paso 1** se puede realizar a partir de cualquier vértice de G ; el resultado del paso 1, en términos de los tiempos $u.f$ asignados a cada vértice, va a depender de cuál vértice partimos. Supongamos que partimos de 0, luego de 6, y finalmente de 7:

$0.d = 1$ $5.d = 2$ $4.d = 3$ $3.d = 4$ $2.d = 5$ $2.f = 6$ $3.f = 7$ $4.f = 8$ $5.f = 9$ $1.d = 10$ $1.f = 11$ $0.f = 12$	$6.d = 13$ $9.d = 14$ $11.d = 15$ $12.d = 16$ $12.f = 17$ $11.f = 18$ $10.d = 19$ $10.f = 20$ $9.f = 21$ $6.f = 22$ $7.d = 23$ $8.d = 24$ $8.f = 25$ $7.f = 26$
---	--

El **paso 2** consiste en transponer G , es decir, invertir la dirección de las aristas; vale tanto el dibujo del grafo, como la representación mediante listas de vértices adyacentes:

$[0] \rightarrow 2, 6$	$[4] \rightarrow 5, 6, 11$	$[8] \rightarrow 7$	$[12] \rightarrow 10, 11$
$[1] \rightarrow 0,$	$[5] \rightarrow 0, 3$	$[9] \rightarrow 6, 8, 12$	
$[2] \rightarrow 3, 4$	$[6] \rightarrow 7$	$[10] \rightarrow 9$	
$[3] \rightarrow 2, 4$	$[7] \rightarrow 8$	$[11] \rightarrow 9$	

En el **paso 3** hacemos DFS del grafo transpuesto, pero en el ciclo principal de DFS consideramos los vértices en orden decreciente de $u.f$ calculado en el paso 1, es decir, en este caso:

partimos con el vértice 7 ($7.f = 26$), a partir del cual sólo podemos llegar al 8, de modo que estos dos vértices forman una componente;

seguimos con el vértice 6 ($6.f = 22$), a partir del cual no podemos ir a ningún otro vértice (excepto 7, pero ya fue considerado), de modo que 6 forma una componente por sí solo;

seguimos con 9 ($9.f = 21$), a partir del cual podemos llegar a 12, 11 y 10 (también a 6, pero ya lo consideramos), de modo que 9, 10, 11 y 12 forman otra componente;

... similarmente encontramos la componente 0, 2, 3, 4 y 5;

... y por último, la componente formada sólo por 1.

I2 2015-1

4. Considera el siguiente grafo no direccional, representado mediante sus listas de adyacencias:

[0]: 6 – 2 – 1 – 5	[1]: 0	[2]: 0	[3]: 5 – 4	[4]: 5 – 6 – 3
[5]: 3 – 4 – 0	[6]: 0 – 4	[7]: 8	[8]: 7	[9]: 11 – 10 – 12
[10]: 9	[11]: 9 – 12	[12]: 11 – 9		

Determina las **componentes conectadas** de este grafo, ejecutando el algoritmo basado en DFS estudiado en clase. En particular, muestra el orden en que se van produciendo cada una de las llamadas recursivas, si la primera llamada es DFS(0); marca los puntos de retorno de las llamadas en que se completa la detección de una componente conectada; y para cada componente conectada detectada lista sus vértices.

```
dfs(0)
  dfs(6)
    0 ✓ – ya lo había descubierto
    dfs(4)
      dfs(5)
        dfs(3)
          5 ✓
          4 ✓
          3 fin – terminé de descubrir todo lo que podía desde 3
          4 ✓
          0 ✓
        5 fin
        6 ✓
        3 ✓
      4 fin
    6 fin
    dfs(2)
      0 ✓
    2 fin
    dfs(1)
      0 ✓
    1 fin
    5 ✓
  0 fin → aquí se completa una componente conectada, formada por 0, 1, 2, 3, 4, 5, 6
  dfs(7)
    dfs(8)
      7 ✓
    8 fin
  7 fin → aquí se completa otra componente conectada: 7, 8
  dfs(9)
    dfs(11)
      9 ✓
    dfs(12)
      11 ✓
      9 ✓
    12 fin
    11 fin
    dfs(10)
      9 ✓
    10 fin
    12 ✓
  9 fin → aquí se completa la última componente conectada: 9, 10, 11, 12
```


EX 2015-2

1. Considera un grafo direccional $G = (V, E)$ con costos y **acíclico** (y que, por lo tanto, puede ser ordenado topológicamente):

a) [3 pts.] Da un algoritmo de tiempo $O(V+E)$ para encontrar las rutas más cortas en G desde un vértice de partida s .

1. *ordenar G topológicamente* —visto en clase; necesario para poder hacer un algoritmo eficiente
2. *para cada vértice v de G : —este es el procedimiento $init(s)$ visto en clase*
 $d[v] = \infty$
 $\pi[v] = nil$
 $d[s] = 0$
3. *para cada vértice u de G , en el orden producido por la ordenación topológica: —esta condición es esencial para la corrección del algoritmo*
para cada vértice v adyacente a u : —este es el procedimiento $reduce(u, v)$ visto en clase
if ($d[v] > d[u] + w(u, v)$)
 $d[v] = d[u] + w(u, v)$
 $\pi[v] = u$

b) [3 pts.] Justifica que tu algoritmo encuentra las rutas más cortas y que toma tiempo $O(V+E)$.

[1.5 pts.] El algoritmo es de tiempo $O(V+E)$, como se justifica a continuación:

- ordenar G topológicamente (paso 1) es $O(V+E)$, ya que esencialmente es hacer un recorrido DFS
- el ciclo "para" que sigue (paso 2) es $O(V)$, por lo que no agrega complejidad
- el ciclo "para" que sigue (paso 3) es $O(V+E)$, ya que mira cada vértice u y, para cada uno, mira cada arista que sale de u (los vértices v adyacentes a u); y, en cada caso, hace una operación de complejidad constante (un *reduce*)

[1.5 pts.] El algoritmo efectivamente calcula las rutas más cortas desde s a todos los otros vértices (alcanzables desde s); en el paso 3:

- cada arista (u, v) es reducida exactamente una vez, cuando u es procesado (cuando la iteración "pasa" por u), dejando $d[v] \leq d[u] + w(u, v)$
- esta desigualdad se mantiene de ahí en adelante, hasta que el algoritmo termina, ya que $d[u]$ no cambia: como los vértices son procesados en orden topológico, ninguna arista que apunte a u va a ser reducida después de procesar u