Estructuras de Datos y Algoritmos - IIC2133
Control 4
6 de mayo, 2019

1) Un *puente* en un grafo no direccional es una arista (*u, v*) tal que al sacarla del grafo hace que el grafo quede desconectado -o, más precisamente, aumenta el número de componentes conectadas del grafo; en otras palabras, la única forma para ir de *u* a *v* en el grafo es a través de la arista (*u, v*). **[4 pts.]** Explica cómo usar el algoritmo DFS para encontrar eficientemente los puentes de un grafo no direccional; y **[2 pts.]** justifica qué tan eficientemente. Recuerda que DFS asigna tiempos de descubrimiento (y finalización) a cada vértice que visita.

R: Para hallar los puentes en el grafo lo que se hará, básicamente, es buscar aquellas aristas que no pertenezcan a algún ciclo dentro de este. Es posible encontrar dichas aristas utilizando el algoritmo DFS. Teniendo nuestro bosque, generado por el algoritmo, supongamos que tenemos una arista (*u, v*) y que en el bosque no es posible llegar desde un descendiente de *v* hasta un ancestro de *u*, entonces diremos que dicha arista no pertenece a ningún ciclo y en consecuencia, es un puente. Para detectar de manera eficiente si una arista es un puente, lo que se hará es guardar, en cada nodo, el menor de los tiempos de descubrimiento de cualquier nodo alcanzable (no necesariamente en un paso) desde donde me encuentre, llamémoslo *v.low*. Si la arista es la (*u, v*), esto es:

$$u.low = min\{u.d, v.low\}$$

Ahora, al momento de retornar el método *dfsVisit*, lo que se hará es comparar dicho valor con el tiempo de descubrimiento del nodo en el que estoy parado. Digamos estoy parado en el nodo *u* y visité *v*, si se tiene que $u.d < v.low$ entonces no se puede alcanzar ningún ancestro de *u* desde algún descendiente de *v*, en consecuencia dicha arista no pertenece a ningún ciclo y es un puente.

Esta forma de detectar puentes en el grafo posee la misma complejidad que el algoritmo DFS, O(|*V*| + |*E*|). Esto ya que lo único que se está haciendo es actualizar un valor y compararlo, lo que no suma mayor complejidad (se hace en tiempo constante).

- **[4 pts]** Se explica un algoritmo o bien las modificaciones que se le deben realizar a DFS para lograr el objetivo de manera clara y concisa. Solución debe ser eficiente. Si no se cumple con los puntos anteriores no hay puntaje.
- **[2 pts]** Solo si el algoritmo es correcto (efectivamente encuentra los puentes) y se justifica el por qué de la eficiencia mostrada.

2) Sea *G*(*V, E*) un grafo no direccional y $C \subseteq V$ un subconjunto de sus vértices. Se dice que *C* es un *k-clique* 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*.

a) [2pts.] 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.

R: Cualquier descripción que describa correctamente la modelación y explique de manera clara las variables, los dominios de las variables y restricciones sobre los valores que pueden tomar dichas variables, de tal forma que se pueda resolver con backtracking, tendrá el puntaje correspondiente. Recordar que al ser no direccional, si (v, v') ϵ E $\rightarrow$ (v', v) ϵ E. A continuación, se plantean dos posibles formas de solucionar el problema:

Propuesta 1:

Las **variables** son nodos pertenecientes al k-clique. El **dominio** son los posibles vértices que puede ir asociado al nodo. Las **restricciones** son que cada vértice que se escoge del dominio tiene que estar conectado a todos los otros nodos que se han escogido anteriormente.

Ejemplo en pseudocódigo (para guía del alumno; no era necesario implementar):

```
X = nodos 1, …, k
D = V
R = {(v, v') ϵ E ∀ v' ϵ C}

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

Luego, podemos llamar a is_solvable({1, …, k}, D, {}).

Propuesta 2:

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.

Ejemplo en pseudocódigo (para guía del alumno; no era necesario implementar):

```
X = V
D = {0, 1}
R = {(v, v') ϵ E ∀ v' ϵ C}

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
```

Luego, podemos llamar a is_solvable({V, D, {}, k).

- **[1 pt]** Explica el conjunto de variables y su dominio.
- **[1 pt]** Explica las restricciones sobre los valores que pueden tomar dichas variables.

b) [2 pts.] Propón una **poda** y explica la modelación a nivel de código requerida para implementarla eficientemente.

R: Si el alumno menciona una poda acorde a su modelación que sea efectiva, y explica cómo implementarla, tendrá el puntaje correspondiente. Algunas podas posibles:

- 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.
- (Propuesta 2) 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 k-clique. 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.

- **[1 pt] Propone** una poda que efectivamente sea útil para el problema a resolver
- **[1 pt] Explica** cómo implementarla eficientemente

c) [2 pts.] Propón una **heurística para el orden de las variables** y explica la modelación a nivel de código requerida para implementarla eficientemente.

R: Si el alumno menciona una heurística acorde a su modelación que sea efectiva, y explica cómo implementarla, tendrá el puntaje correspondiente. Algunas heurísticas posibles:

- (Propuesta 2) 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.)

- (Propuesta 1) Se puede ordenar el dominio en función de la cantidad de aristas que tienen las variables. De esta manera, es más probable que al asignar a la variable su valor, éste pertenezca al k-clique. El procedimiento es similar a la heurística anterior: 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.), y utilizar esto como el dominio de la función de backtracking.

- **[1 pt] Propone** una heurística que efectivamente sea útil para el problema a resolver
- **[1 pt] Explica** cómo implementarla eficientemente