Representación del Conocimiento

Práctica 2: Separación Gráfica

Arahí Fernández Monagas

Iker Martínez Gómez

Carlos Velázquez Fernández

Grado en Ingeniería
Informática
Mención en Computación
Curso 2023-2024

1. Fundamentos teóricos

El objetivo de la práctica consiste en implementar un algoritmo que compruebe si dos variables son separables gráficamente dado el grafo de una red bayesiana. Primero, debemos comprender bien el significado de estos conceptos de manera individual:

Una **red bayesiana** es un grafo dirigido acíclico $G = (V, E)$ en el que cada $v_i \in V$ se asocia con una variable aleatoria $X_i \in X$ y cada X_i, con distribuciones condicionales $p(x_i \mid pa(X_i))$. En otras palabras, cada vértice representa una variable cuya distribución de probabilidad consiste en la probabilidad de que ocurra esa variable dados sus padres. Podemos escribir esta red como una factorización:

$$p(x) = \prod\ p(x_i \mid pa(X_i))$$

donde cada variable discreta X_i puede tomar k valores.

En múltiples ocasiones nos interesa saber si dos variables son independientes la una de la otra. Al añadir nuevas variables a la red bayesiana la complejidad crece de forma exponencial, pero no todas nos importan para todas las consultas, por lo que podemos tratar de simplificarla. La factorización desarrollada previamente nos ayuda a definir esta **independencia condicional** como:

$$X \perp_p Y \mid \mathbf{Z} \iff \exists\phi_1\ \exists\phi_2\ [p(x, y, \mathbf{z}) = \phi_1(x, \mathbf{z}) \cdot \phi_2(y, \mathbf{z})]$$

siendo p una distribución y ϕ_1 y ϕ_2 dos funciones no negativas.

Sin embargo, existe una forma mucho más sencilla de decidir si dos variables son independientes y es mediante la **separación gráfica**. Si dos variables son separables gráficamente, entonces son independientes.

$$X \perp_G Y \mid \mathbf{Z} \implies X \perp_p Y \mid \mathbf{Z}$$

$X \perp_G Y \mid \mathbf{Z}$ si no hay caminos activos entre X e Y dado $\mathbf{Z}$. Siguiendo este teorema podemos programar un algoritmo que, tomando como entrada un grafo, compruebe si dos variables son separables dado un conjunto de variables observadas. Si el resultado es afirmativo, entonces sabemos con certeza que éstas son independientes. En caso contrario, únicamente sabemos que son dependientes en alguna distribución probabilística.

2. Algoritmo de separación gráfica

Para decidir $X \perp_G Y \mid \mathbf{Z}$ seguimos los siguientes pasos:

1. Eliminar vértices hojas del grafo (evitando $\{X, Y\} \cup \mathbf{Z}$).
2. Moralizar el grafo (unir padres con hijos en común y eliminar la direccionalidad de $E(G)$).
3. $X \perp_G Y \mid \mathbf{Z}$ si no existen caminos entre X e Y sin ningún $Z \in \mathbf{Z}$.

El algoritmo implementado (representado en la figura 1) lleva a cabo estas tareas. La función *graphicSeparation(G, X, Y, Z)* recibe como parámetros un grafo *G*, las variables X e Y de las cuáles queremos saber si son independientes y una lista de vértices observados *Z*. El valor de retorno será *True* en caso de que dichas variables sean independientes y *False* en caso contrario.

En las próximas secciones se desarrollan cada uno de estos pasos en detalle, manteniendo la nomenclatura anterior de G, X e Y, y siendo V y E los vértices y las aristas de G respectivamente. Para representar a los padres de un nodo v se emplea la función $pa(v)$, así como $ch(v)$ para sus hijos.

2.1. Eliminación de vértices hoja

La eliminación de los vértices hoja (nodos los cuales no tienen hijos) en la separación gráfica resulta una estrategia útil para simplificar y mejorar la interpretación de las redes bayesianas. Facilita la identificación de dependencias, pues estas redes pueden representar relaciones complejas entre diferentes variables, y a medida que aumenta el número de nodos y aristas en el grafo, la representación de este se vuelve más difícil de comprender. Es por esto que al eliminar vértices hoja, se logra reducir nodos y aristas innecesarios para comprobar la independencia de dos variables dadas.

Para eliminar los vértices hoja se ha implementado una función llamada *deleteLeaves(G, leaves, cantDelete)*. Este método recibe como input un grafo, una lista que contiene los vértices hoja iniciales que deben ser eliminados y una lista de vértices que no deben ser eliminados, es decir, aquellos vértices observados o de los cuales queremos comprobar su independencia, siguiendo la expresión $X \perp Y \mid Z$.

El algoritmo explora iterativamente todos los vértices hoja que se encuentran en la lista *leaves*[1] y los eliminará siempre y cuando no estén contenidos en *cantDelete*. Antes de eliminar cada vértice se obtiene información acerca de sus respectivos padres. Una vez eliminado, debemos verificar si estos padres se han convertido en nodos hoja como consecuencia de la eliminación de sus descendientes, En caso afirmativo, se agregan dichos padres a la lista de vértices hoja a eliminar para que también puedan ser borrados en futuras iteraciones. Una vez finalizado el bucle while, finaliza el programa devolviendo el nuevo grafo modificado.

Es importante comprobar si los padres de los nodos eliminados pasan a ser vértices hoja para intentar reducir el grafo al máximo posible. Esto puede ser fundamental en situaciones donde se necesita un grafo limpio y sin nodos innecesarios para determinadas consultas, lo que facilita análisis y visualización de datos, además de simplificar su propia estructura.

2.2. Moralización

La moralización de un grafo consiste en la unión mediante aristas de nodos con hijos en común y la eliminación de la direccionalidad.

La función *moralize(G)* (representada en el algoritmo 4) recibe como su único argumento un grafo y retorna un nuevo grafo moralizado. Primero se identifica qué nodos comparten hijos, guardando un diccionario *parents* (llave: vértice, valor: lista de padres). Para cada valor del diccionario, es decir, para cada conjunto de nodos con hijos en común, se añaden tantas aristas nuevas a *newEdges* como combinaciones de sus elementos sean posibles (siempre y cuando las aristas no estén ya en el grafo). Finalmente, se crea un nuevo grafo sin direccionalidad cuyos vértices serán los mismos que en el grafo original, y cuyo conjunto de aristas será la unión de las nuevas con las antiguas.

2.3. Búsqueda de camino

Para comprobar si existe un camino entre dos nodos de un grafo acíclico no dirigido, podemos emplear distintas técnicas. En la práctica se ha optado por una búsqueda en anchura.

La función *existsPath* (descrita en 5) recibe un grafo G, y dos nodos X e Y; y devuelve True o False en función de si existe un camino que conecte ambos vértices. Comenzamos el algoritmo con una lista de nodos a explorar *queue*, la cual contendrá inicialmente a *X*. Mientras haya elementos en la lista, realizaremos los siguientes pasos:

1. Sacamos el nodo de *queue*.

[1] **leaves** se obtiene con ayuda del método auxiliar *leaves(G)*, cuyo pseudocódigo se encuentra en 2.

2. Si ya lo habíamos visitado, pasamos a la siguiente iteración. En caso contrario seguimos.

3. Si dicho nodo es Y, significa que hemos alcanzado el destino, por lo que existe un camino. Retornamos *True*.

4. Añadimos el nodo a *visited* y todos sus vecinos a *queue* para que se exploren en próximas iteraciones.

Si el bucle while termina, significa que hemos explorado todos los vértices sin haber encontrado un camino entre X e Y, por lo que deberemos de retornar *False*.

3. Pruebas

Para comprobar de manera efectiva que el algoritmo de separación gráfica funciona correctamente, se ha desarrollado un fichero de prueba llamado **test.py** en el cual se presentan 3 diferentes grafos, cada uno de ellos con 4 consultas distintas (dos casos donde sabemos que la separación gráfica es *True*, es decir, que sí existe independencia entre X e Y dado Z, y otros dos casos en los que la separación gráfica es *False*).

Para una mejor diferenciación de los tests, se ha optado por separar en diferentes funciones cada grafo, presentado en cada uno de ellos los cuatro casos. Por último, al final del programa, se imprime el mensaje "TODOS LOS TESTS HAN PASADO DE FORMA CORRECTA PARA CADA GRAFO", únicamente si ha ocurrido así.

El motivo principal para realizar estas comprobaciones es asegurarse de que el algoritmo de separación gráfica funcione correctamente en todas las situaciones. Variar el input del programa (los grafos de entrada) ayuda a identificar y corregir posibles errores en el código, que no hayan surgido durante el desarrollo.

4. Complejidad Temporal

Veamos el cálculo de la complejidad temporal para las distintas funciones del algoritmo de separación gráfica (por simplicidad, las operaciones que se realicen en tiempo constante serán despreciadas para los cálculos):

1. **Función leaves:** Al recorrer en un bucle *for* todos los nodos del grafo comprobando si tienen descendencia, la complejidad de esta función será $\mathbf{O(V)}$, siendo V el número de nodos del grafo. Los métodos *DiGraph.predecessors* y *DiGraph.successors* se basan en consultas a diccionarios por lo que su complejidad temporal es constante.

2. **Función deleteleaves:** Comenzamos creando una copia en profundidad del grafo para no editar aquel qué se recibe como argumento. Esto tiene un coste computacional aproximado de $O(V+E)$, ya que se deben recorrer tanto vértices (V) como aristas (E) para crear el nuevo objeto. El bucle *while*, en el peor de los casos (todos los vértices del grafo (a excepción de X e Y) son hoja y no se observa ninguno, es decir, $Z = \emptyset$), constará de una complejidad de $O(V)$. Por otro lado tenemos un bucle *for* que recorre todos los predecesores de cada nodo a eliminar. El peor caso que podemos contemplar aquí es aquel en el que siendo los vértices $v_0..., v_i..., v_n$, cada nodo v_i tenga i padres (el grafo debe ser acíclico). El número de elementos de cada lista del bucle en este caso variaría en cada iteración. En total, se recorrerían $\sum_{i=0}^{V}(V-(V-i))$ elementos, lo que podemos representar como $O(V-(V-\frac{V-1}{2}))$, o más sencillamente, $O(V)$, puesto que al final es la única variable de la que se depende y además es lineal. Teniendo en cuenta ambos bucles, la complejidad resultante será $O(V+E+E\times V) \approx \mathbf{O(V^2)}$.

3. **Función moralize:** Comenzamos con la búsqueda de los padre e hijos. Para ello se itera a través de todos los vértices en el primer bucle *for*, que presenta una complejidad temporal en el peor de los casos $O(V)$, siendo V el número de vértices. La complejidad temporal de este algoritmo se define a la hora de unir los vértices con sus hijos en común. Para realizar esto tenemos tres bucles anidados, que recorren la lista de padres de cada nodo y añaden las aristas correspondientes. Esto presentaría una complejidad temporal, en el peor de los casos, de $O(V^3)$. Por lo tanto, la complejidad temporal de la función viene dada por esta unión de padres con hijos en común, siendo $\mathbf{O(V^3)}$.

4. **Función existsPath:** Al ejecutar esta función estamos realizando una búsqueda en anchura en el grafo dado. Siguiendo el pseudocódigo 5 podemos comprobar que cada nodo se visitará una única vez (el número de iteraciones del *while* será ligeramente superior puesto que encolamos todos los nodos vecinos cada vez que exploramos uno nuevo, independientemente de si se ha visitado, aunque luego no lo exploremos si se cumple esta condición). El número de iteraciones del *for*, teniendo en cuenta las de cada exploración, dependerá del grado de cada nodo, aunque lo podemos simplificar como número de aristas del grafo. Por todo ello podemos aproximar la complejidad temporal de esta función a $\mathbf{O(V + E)}$.

Si hacemos una vista general del algoritmo siguiendo el pseudocódigo 1, podemos comprobar que la complejidad temporal resultante vendrá dada por la mayor de todas las funciones anteriores. La complejidad de la separación gráfica será $O(V + V^2 + V^3 + (V + E)) \approx \mathbf{O(V^3 + E)}$.

A. Anexo: Algoritmos

Algorithm 1 Separación Gráfica

function GRAPHICSEPARATION(G, X, Y, Z)

$G' \leftarrow deleteLeaves(G, leaves(G), \{X, Y\} \cup Z)$ ▷ Remove leaf nodes
$G'' \leftarrow moralize(G')$ ▷ Moralize the graph

for $z \in Z$ **do**
$G''.remove(z)$ ▷ Remove observed nodes
end for

$path \leftarrow existsPath(G'', X, Y)$

return not $path$ ▷ No path means variables are separable
end function

Algorithm 2 Nodos Hoja

procedure LEAVES(G)
$leaf \leftarrow list()$
for $v \in V$ **do**
if $ch(v) = \emptyset$ **then**
$leaf.append(v)$
end if
end for
return $leaf$
end procedure

Algorithm 3 Eliminación de Nodos Hoja

procedure DELETELEAVES($G, leaves, cantDelete$)

$G' \leftarrow G$

while *leaves not empty* **do**
$l \leftarrow leaves.pop()$ ▷ l can not be X or Y, nor be in Z

if $l \notin cantDelete$ **then**
$G'.remove(l)$

for $p \in pa(l)$ **do**
if $ch(p) = \emptyset$ **then** ▷ Chek if parent is now a leaf
$leaves.append(p)$
end if
end for

end if

end while

return G'
end procedure

Algorithm 4 Moralización

function MORALIZE(G)

$parents \leftarrow dictionary()$ ▷ (key: child, value: [parents])
$newEdges \leftarrow set()$ ▷ Edges matching parents with common children
for $v \in V$ **do**
$parents[v] \leftarrow pa(v)$
end for

for $P \in parents.values()$ **do** ▷ Match parents with common children
for $v_1 \in P$ **do**
for $v_2 \in P \backslash \{v_1\}$ **do**
if $edge(v_1, v_2) \notin E$ **then**
$newEdges.append(edge(v_1, v_2))$
end if
end for
end for
end for

$G' \leftarrow undirectedGraph(V, E \cup newEdges)$ ▷ New graph with new edges and no directionality

return G'
end function

Algorithm 5 Existe Camino

```
function existsPath(G, X, Y)

    queue ← list(X)
    visited ← list()

    while queue not empty do                          ▷ Breadth-first search
        v ← queue.pop()

        if v ∉ visited then
            visited.append(v)

            if v = Y then                             ▷ There is a path
                return True
            end if

            for e ∈ E do                              ▷ Append neighbor nodes
                if v ∈ e then
                    queue.append(v)
                end if
            end for

        end if
    end while

    return False                                      ▷ There is not any path
end function
```