Ficha 05

Grafos – Implementación Matricial

1.] Introducción.

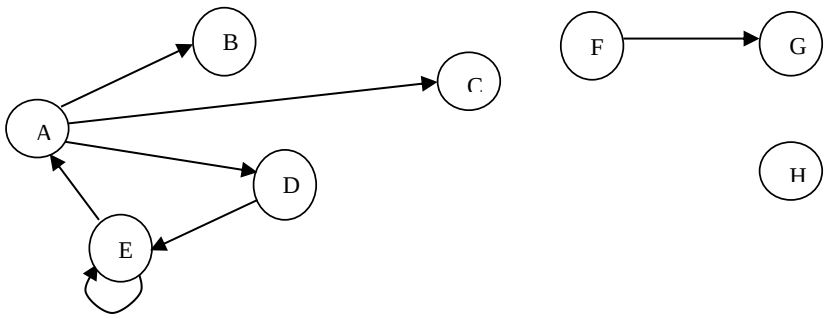
Hasta aquí las estructuras de datos estudiadas permitían almacenar elementos pero no nos decían mucho respecto de las relaciones entre ellos, más allá de la relación de "sucesor" o "antecesor" que surje de la propia disposición de los objetos en cada estructura.

Sin embargo, muchos problemas se centran en colecciones de objetos u elementos y *también* en las relaciones existentes entre ellos. Por ejemplo, un sociólogo querrá almacenar la información sobre los individuos que integran los grupos humanos que estudia, pero también (y con la misma importancia) querrá almacenar la información sobre las relaciones que se dan entre esos individuos (¿habla el individuo ***a*** con el ***b***? ¿comparten momentos fuera del trabajo los individuos ***c*** y ***d***?, etc.) Por otra parte, los directivos de una empresa de servicios viales querrán almacenar información acerca de cada ciudad existente en la región en que prestan servicios, pero también querrán almacenar información acerca de qué caminos (o sea, relaciones) existen entre qué ciudades, para luego poder analizar la forma de optimizar los procesos de transporte entre ellas. Y sólo hemos citado dos ejemplos conocidos del mundo real... que pueden representarse muy bien mediante *grafos*.

Los *grafos* son estructuras de datos que permiten modelar con relativa sencillez las *relaciones* que existen entre los diversos *objetos* del dominio de un problema. Muchos de los problemas referidos a grafos se han estudiado muy bien, existiendo soluciones eficientes muy conocidas y aplicadas. Otros problemas están bien estudiados, pero no se conocen soluciones eficientes (o sea, soluciones que no consistan en analizar por *fuerza bruta* todas las combinaciones de datos posibles, lo cual lleva a algoritmos que resultan inaplicables cuando el número de datos es grande). Mucha de la investigación que se realiza en el mundo de las ciencias de la computación tiene que ver con problemas referidos grafos.

Un ***grafo*** *es una estructura de datos compuesta por un conjunto de n nodos (también llamados vértices) y por un conjunto de m arcos (también llamados aristas), de tal forma que un arco une dos vértices cualesquiera*. Notar que en la definición no se imponen reglas en cuanto a qué vértices deben unirse por cuáles arcos, ni cuántos arcos pueden salir de o llegar a un vértice. Esos elementos dependen del problema que se está modelando. El siguiente es un ejemplo de un grafo, compuesto por un conjunto de ocho vértices *V = {A, B, C, D, E, F, G, H}* y por un conjunto de siete arcos *W = {(A,B), (A,C), (A,D), (D,E), (E,E), (E,A), (F,G)}*:

Figura 1: un grafo dirigido

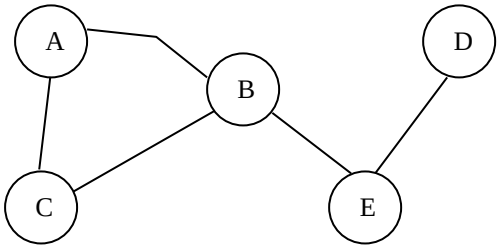


2.] Tipos de grafos.

En la *Figura 1* anterior se muestra la representación gráfica de un *grafo* compuesto por ocho vértices y siete arcos. Por otra parte, dependiendo del problema, pueden darse distintas modalidades de grafos. Si en un grafo importa indicar el *sentido* de cada arco (es decir que cada arco tiene un *vértice de partida* o *inicio* y un *vértice de llegada*), entonces el grafo se dice *dirigido* y se dibuja a cada arco terminado en punta de flecha. Se entiende que el vértice desde el cual parte la flecha es el vértice del partida del arco, y el vértice al cual apunta la flecha es el vértice de llegada. El grafo de la *Figura 1* es un *grafo dirigido*: los arcos indican relaciones entre nodos, y el sentido de cada arco indica cual es el nodo de partida y cual el nodo de llegada de la relación. Por ejemplo, un *grafo dirigido* como el de la *Figura 1* puede usarse para modelar un *sociograma*: cada vértice puede representar a un individuo de un grupo, y cada arco puede representar algún tipo de relación entre esos individuos que esté siendo estudiada por un sociólogo o un psicólogo. Así, por ejemplo, si un arco parte del vértice que representa al individuo *A* y llega a un individuo *B*, podría significar que para que *A* ha elegido a *B* para algún tipo de tarea que se hubiese propuesto, y así modelar una red de preferencias entre los individuos del grupo.

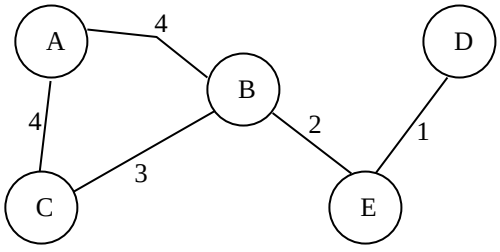
Si el grafo no se requiere que los arcos sean dirigidos (es decir, sólo importa indicar que el arco existe pero no cual de sus vértices es el origen y cual el de llegada), entonces el grafo se dice *no dirigido* y sus arcos se representan simplemente como una línea que une dos vértices, pero sin punta de flecha en ninguno de sus extremos: el arco que conecta dos vértices puede entenderse como que puede recorrerse indistintamente en un sentido como en el otro. El grafo de la *Figura 2* es un *grafo no dirigido*. Los grafos no dirigidos pueden usarse, como ejemplo, para modelar una red de caminos que vinculan a distintas ciudades o pueblos de una zona: cada vértice será una ciudad o pueblo, y cada arco indicará la existencia de una ruta directa entre ambas. Es claro que si hay una ruta que va de la ciudad *A* hacia la *B*, entonces la misma ruta existe saliendo de *B* y viajando hacia *A*...

Figura 2: un grafo no dirigido



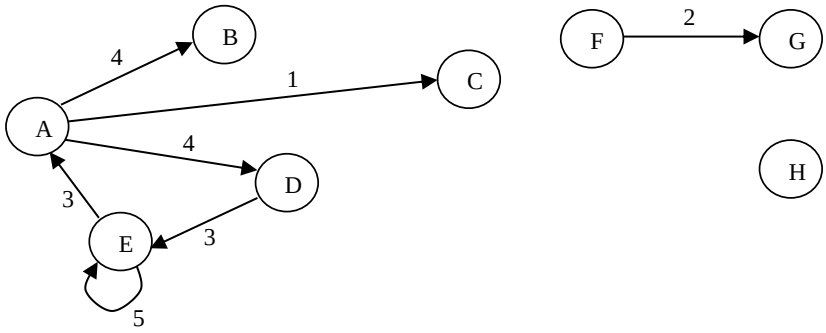
También puede ocurrir que en algunos problemas se requiera que cada arco tenga asociado un *valor*, *peso* o *ponderación*. El grafo puede ser en este caso dirigido o no dirigido, pero se debe asociar a cada arco el *peso* del mismo. En estos casos, el grafo se dice *ponderado* o con *factores de peso*. El grafo de la *Figura 3* es un *grafo ponderado no dirigido*. Un grafo ponderado no dirigido podría usarse en el mismo caso anterior (un modelo de ciudades y las rutas que las conectan) pero agregando en cada arco la *distancia* entre las ciudades unidas por ese arco (o algún otro valor como podría ser el monto a pagar en concepto de peaje si se circula por esa zona, etc.)

Figura 3: un grafo ponderado no dirigido



Por cierto, un grafo ponderado también puede ser dirigido de acuerdo a los requerimientos del dominio del problema. El mismo grafo dirigido de la *Figura 1* podría tener factores de peso como se muestra en la *Figura 4*:

Figura 4: un grafo ponderado dirigido



3.] Definiciones y terminología esencial.

Más allá de la representación gráfica, un grafo puede ser descripto usando notación de conjuntos. Para el conjunto de vértices basta con una definición por enumeración de los valores contenidos en cada vértice. En cuanto al conjunto de arcos, la notación depende del tipo de grafo. Si el grafo es dirigido se enumeran los arcos usando la noción de *par ordenado* para cada uno. Así, el arco que parte del vértice *A* y llega al *B* puede representarse con el par ordenado *(A, B)* en el que el primer elemento del par indica el vértice de partida, y el segundo indica el de llegada. De esta forma, el par *(A, B) no representa* al mismo arco que el par *(B, A)* pues en este último el vértice de partida sería *B*, y el de llegada el *A*. Con estos criterios, el grafo *G = {V, W}* de la *figura 1* (donde *V* es el conjunto de vértices y *W* el de arcos) puede describirse así:

$$G \begin{cases} \text{conjunto de vértices} \quad V = \{A, B, C, D, E, F, G, H\} \\ \text{conjunto de arcos} \quad W = \{(A, B), (A, D), (A, C), (C, C), (D, E), (E, A), (F, G)\} \end{cases}$$

Y está claro que en un grafo no dirigido, el par que represente a un arco será un par libre en el cual no tendrá importancia cual sea el vértice partida y cual el de llegada. En ese sentido, en un grafo no dirigido el par *(A,B)* hará referencia indistintamente al mismo arco que el par *(B,A)*.

Las figuras que hemos mostrado en las secciones anteriores ilustran algunos conceptos interesantes, que enumeramos:

a.) Un grafo puede constar de varios subgrafos no unidos entre sí. La *figura 1* representa un *solo grafo*, el cual contiene tres partes o subgrafos no unidos por arcos a los otros subgrafos. Cada uno de estos subgrafos separados de un grafo, se suele llamar también *componente conexa*. En este grafo hay tres componentes conexas: la primera está formada por los vértices *A*, *B*, *C*, *D*, y *E*, que están relacionados entre sí de alguna manera. La segunda está formada por los vértices *F* y *G*, que se relacionan entre sí pero no con los otros vértices. Y la última, está conformada sólo por el vértice *H*, que no se relaciona con ningún otro.

b.) Si todos los vértices de un grafos están vinculados entre sí formando una sola gran componente conexa, el grafo es un *grafo conexo*. La *Figura 2* y la *Figura 3* representan grafos conexos.

c.) El caso del vértice *H* en la *figura 1*, muestra claramente que no es obligatorio que un vértice sea punto de partida o de llegada de algún arco. Un vértice puede estar completamente aislado del resto.

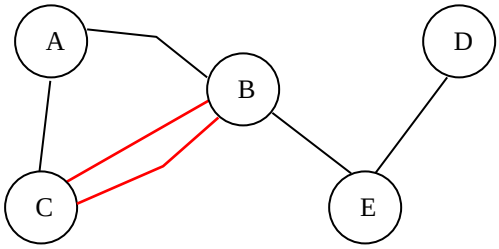
d.) En general, dos vértices se dicen *adyacentes* si existe un arco que los une, sin importar, en principio, el sentido de ese arco. En la *figura 1*, por ejemplo, los vértices *A* y *C* son adyacentes. Observar que el vértice *H* no es adyacente a ningún otro vértice del grafo.

e.) Se dice que entre dos vértices *A* y *B* hay un *camino de longitud k*, si existe una secuencia de *k* arcos que permitan llegar desde *A* hasta *B*. Es obvio que si dos vértices son adyacentes, entonces existe un camino de longitud 1 (uno) entre ellos. En la *Figura 2*, por ejemplo, existe un camino de longitud 3 entre los vértices *C* y *D*: el camino incluye los arcos *(C, B)*, *(B, E)* y *(E, D)*.

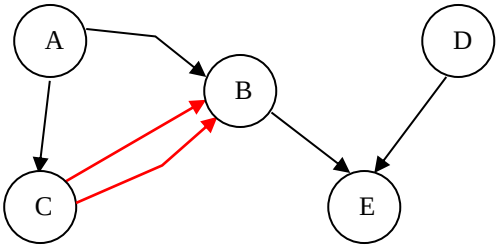
f.) Si existe un camino de cualquier longitud desde un vértice que lleve de retorno al mismo vértice, entonces el grafo tiene un *ciclo*, y se denomina *grafo cíclico*. En caso contrario, el grafo es *acíclico*. En la *figura 1* hay un ciclo que une los vértices *A*, *D* y *E*. Un arco que parte de un nodo y llega al mismo nodo, se denomina *auto ciclo*. En la *figura 1*, el vértice *C* tiene un *auto ciclo*.

g.) En general, dados dos vértices *A* y *B* de un grafo no dirigido podrá ser válido que haya uno o *más arcos* $(A,B)_1$, $(A,B)_2$, $(A,B)_k$ que los unan, teniendo así la noción de *arcos paralelos*. Esto puede incluso ser cierto para grafos dirigidos, en los que podría haber varios arcos con el mismo sentido direccional uniendo los mismos vértices. *La Figura 5* que sigue muestra ejemplos de grafos simples con arcos paralelos (marcados en color rojo). De todos modos, queda claro que la validez o no de contar con arcos paralelos en un grafo depende de la situación: para algunos problemas y algoritmos será válido y natural la presencia de más de un arco uniendo los mismos vértices, aunque para muchos problemas ese no sea el caso.

Figura 5: grafos con arcos paralelos



a.) un grafo no dirigido con dos arcos paralelos $(B,C)_1$ y $(B,C)_2$.



b.) un grafo dirigido con dos arcos paralelos $(C,B)_1$ y $(C,B)_2$.

h.) Si no se admiten arcos paralelos y se admiten auto ciclos, entonces es fácil comprobar que un grafo *G* con *n* vértices tendrá un máximo de n^2 arcos (un auto ciclo para cada vértice y un arco desde cada vértice hacia cada uno de los otros *n-1* vértices) aunque las posibilidades reales de contar con esa cantidad de arcos dependerá del tipo de grafo y del problema. Por razones que impactan en la forma y eficiencie de implementación de un grafo en un lenguaje de programación, conviene distinguir al menos dos casos: un grafo *G* de *n* vértices se dice *grafo denso* si su número de arcos *m* se aproxima a n^2 (o sea, *el número de arcos m es* $O(n^2)$). Si en cambio el número de arcos *m* se aproxima a *n* (es decir, $m = O(n)$) entonces tenemos un *grafo poco denso*. En general, un grafo *G* para el que existan todos los arcos para unir a todo par posible de vértices, se llama también *grafo completo* (y es claro que un grafo completo será *denso*).

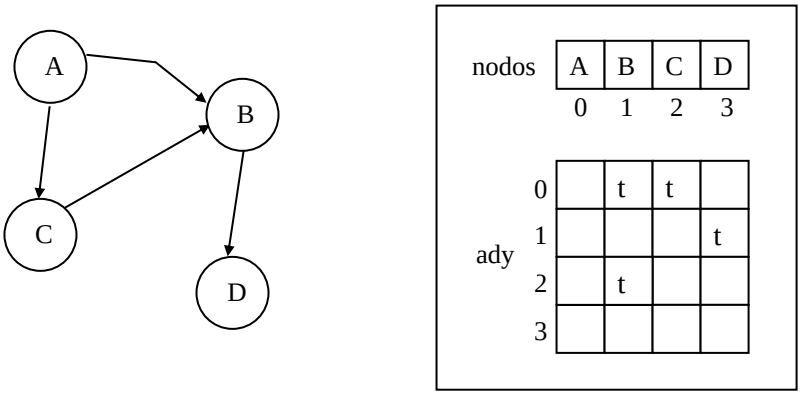
4.] Implementación matricial de grafos.

Existen diversas maneras de implementar un grafo en un lenguaje de programación y la decisión de usar una u otra alternativa dependerá de algunos factores típicos. La forma clásica (y más elemental) de representación de un grafo consiste en usar un *arreglo bidimensional* para representar los arcos. Asumiendo que el grafo tendrá un número de vértices no mayor a un cierto número *n* conocido, la información asociada a cada vértice podría almacenarse en un vector de *n* componentes (uno por cada vértice), y las relaciones o arcos podrían estar mantenidas en una matriz *ady*, cuadrada de orden *n*, que en principio podría ser booleana. El uso de esta matriz es la que justifica que la técnica se designe como *implementación matricial*.

Si el grafo es dirigido, la idea es que las filas de la matriz representarán a los vértices de partida de los arcos del grafo y las columnas representarán a los vértices de llegada. Si existe un arco entre los vértices *i*

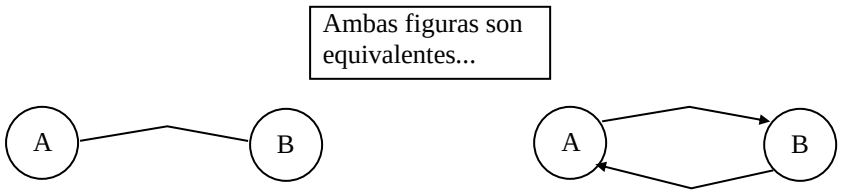
y *j*, entonces habrá un valor *true* en el componente *ady[i][j]* y si no existe tal arco entonces el valor del componente será *false*. En el siguiente ejemplo mostramos un grafo dirigido simple y su representación matricial (el valor *true* es representado con una *t*, y los casilleros en blanco se suponen valiendo *false*):

Figura 6: Implementación matricial de un grafo dirigido



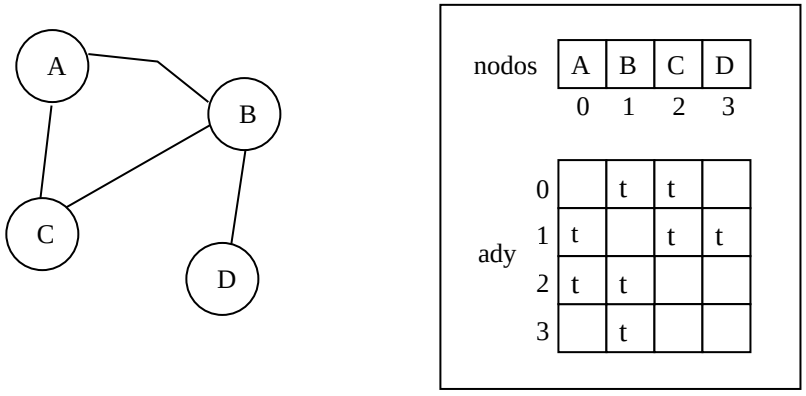
La matriz usada para representar los arcos del grafo se designa como *matriz de adyacencias* del grafo, ya que la misma efectivamente está repesentando todas las relaciones de adyacencia entre los vértices o nodos del grafo. El vector usado para contener los vértices nos indica también el número de fila o columna que le corresponde a cada nodo en la matriz. Así, el vértice *A* está asignado en el casillero cero del vector, y esto nos dice que en la matriz el nodo *A* será representado por la *fila 0* (cuando se lo tome como vértice de partida de un arco) o por la *columna 0* (cuando se lo tome como nodo de llegada de un arco).

Para implementar un *grafo no dirigido* en forma matricial, debemos hacer una observación simple: cada arco de un grafo no dirigido puede entenderse en realidad como *dos arcos dirigidos*: el que va del primer nodo al segundo y el que vuelve del segundo al primero:



En este sentido, un *grafo no dirigido* no es otra cosa que un *grafo dirigido simétrico*: un grafo dirigido en el cual cada vez que existe el arco *(A, B)* existe también el arco *(B, A)* y por lo tanto, la implementación matricial puede hacerse en la misma forma que ya vimos, pero tomando la precaución de hacer que cada vez que asigne un *true* en el casillero *ady[i][j]*, se asigne también en *ady[j][i]*:

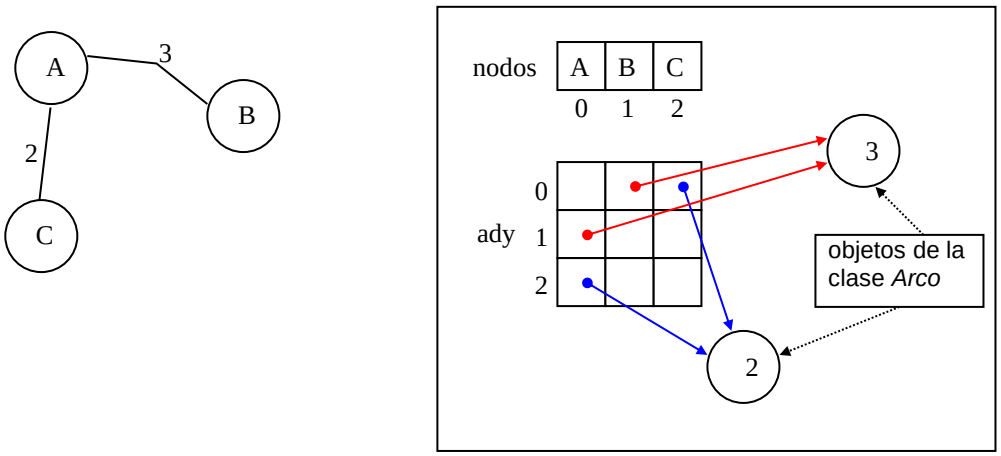
Figura 7: Implementación matricial de un grafo no dirigido



Si el grafo es *ponderado* (dirigido o no dirigido) entonces para su implementación debemos considerar la forma de almacenar el peso o valor de cada arco. Una idea sería que en la matriz de adyacencias se almacene directamente el peso del arco (si el arco existe) y un cero si el arco no existe. Pero esta idea trae el problema de la *ambigüedad del cero*: el valor cero podría interpretarse como ausencia de arco, o como un arco cuyo peso es cero (si el peso cero fuese admisible)... Además, si quisiéramos que un arco tenga asociados varios valores a modos de *peso múltiple*, esa idea no sería viable (aunque podría resolverse admitiendo arcos paralelos).

Lo normal es representar a *los arcos como objetos de una clase Arco*, la cual puede tener tantos atributos como sean necesarios, y almacenar en cada casilla de *ady* una referencia a un objeto de esa clase. En principio, si el arco no existe podríamos dejar en *null* ese casillero, pero a la larga esa idea también traería problemas a la hora de representar el grafo en forma de *String* (además de otros problemas molestos de control), por lo cual preferiremos que cada objeto de la clase *Arco* tenga un atributo *boolean* para indicar si ese objeto representa un arco válido en el grafo o no. Toda la matriz de adyacencias tendrá referencias no nulas a objetos *Arco*, y cada objeto dirá si el arco representado vale o no. En el ejemplo siguiente, mostramos la idea. Suponemos que los casilleros vacios de la matriz *ady* apuntan ellos también a objetos de la clase *Arco*, aunque no los mostramos. También suponemos que los objetos *Arco* que se muestran, tienen un atributo *boolean* que vale *true* (aunque tampoco lo mostramos aquí):

Figura 8: Implementación matricial de un grafo ponderado no dirigido



La implementación matricial tiene la ventaja de la sencillez, por cuanto el peso de la implementación recae casi totalmente sólo en una matriz cuadrada. Pero como contrapartida, el uso de esa matriz hace que se desperdicie espacio en memoria: la matriz usa una cantidad de exactamente n^2 casillas para representar a todo arco no paralelo posible, existan o no esos arcos. Es claro entonces que la representación matricial será aplicable sólo si el grafo es denso o si por alguna razón se prefiere mantener simplicidad (por ejemplo, por razones didácticas). Además, si el grafo admitiese arcos paralelos la implementación matricial no sería tan práctica ni tan obvia. La alternativa es implementar el grafo mediante *listas multiencadenadas*, lo cual puede hacerse en formas diversas, como veremos. En todo lo que resta del curso, asumiremos que la implementación a usar es la de listas multiencadenadas: la versión matricial se muestra aquí sólo a título informativo y para mantener una visión completa del tema, pero para casi todos los algoritmos que se pedirá implementar se asumirá la versión de listas enlazadas.

En el modelo *DLC-Grafos-Matricial* que acompaña a estas notas, mostramos la idea matricial en una implementación general muy básica (que repetimos, no es la que esperamos que se use luego en tareas y actividades). En ese modelo, la clase *Grafo* representa un *grafo dirigido y ponderado*. En el vector de nodos de esa clase almacenamos referencias polimórficas de la clase *Object*, para permitir generalidad (y en esta implementación simple el control de homogeneidad se hace en forma manual). Renunciamos a la idea primaria de almacenar directamente un *boolean* en cada casilla de la matriz *ady*, y en cambio almacenamos referencias a objetos *Arco*. Si en la práctica queremos que el grafo sea sin factores de peso, simplemente

ignoramos los factores de peso al crear un arco: la clase *Grafo* provee dos versiones sobrecargadas del método *unir()*: en la primera se nos piden los vértices a unir y un factor de peso para el arco, pero en la segunda sólo se nos piden los vértices a unir y se asume que el peso del arco será cero. También incluimos un método *cortar()* que elimina el arco entre dos vértices, simplemente haciendo que en ese arco se ponga en *false* el atributo *existe*. Hay algunos otros atributos en la clase, que serán de utilidad en algunos algoritmos que posteriormente veremos:

```
public class Grafo
{
   // atributos esenciales
   protected Object   []nodos;    // vector de nodos
   protected Arco  [][]ady;       // matriz de adyacencias

   // requerido por el algoritmo de Warshall
   protected boolean  [][]trans;

   public Grafo ()
   {
    this(10);
   }

   public Grafo ( int n )
   {
       // el vector de nodos, todas las casillas valiendo null...
       nodos = new Object[n];

       // la matriz de adyacencias...
       ady   = new Arco [n][n];
       for(int i = 0; i < n; i++)
       {
          for(int j = 0; j < n; j++)
          {
             //... llena de Arcos con "existe" valiendo false
             ady[i][j] = new Arco();
          }
       }

       // la matriz de cierre transitivo…
       trans = new boolean [n][n];
   }

   public boolean unir ( Object n1, Object n2 )
   {
      return unir(n1, n2, 0);
   }

   public boolean unir ( Object n1, Object n2, int p )
   {
       boolean out = false;
       int i1 = buscar (n1);
       int i2 = buscar (n2);
       if( i1 != -1 && i2 != -1 )
       {
         ady[i1][i2] = new Arco(p);
         out = true;
       }
```

```
        return out;
    }

    public boolean cortar ( Object n1, Object n2 )
    {
        boolean out = false;
        int i1 = buscar (n1);
        int i2 = buscar (n2);
        if( i1 != -1 && i2 != -1 )
        {
          ady[i1][i2].set(false);
          out = true;
        }
        return out;
    }
    // resto de la clase aquí....
}
```

En el mismo modelo, la clase *GrafoNoDirigido* deriva de la clase *Grafo* y representa *grafos ponderados no dirigidos*, basándonos en la idea ya vista: *un grafo no dirigido puede entenderse como un grafo dirigido simétrico*. La nueva clase simplemente redefine el método *unir()* para que cada vez que se asigna un arco en *ady[i][j]* lo haga también en *ady[j][i]*:

```
public class GrafoNoDirigido extends Grafo
{
   public GrafoNoDirigido()
   {
      this (10);
   }

   public GrafoNoDirigido (int n)
   {
     super(n);
   }

    public boolean unir ( Object n1, Object n2, int p )
    {
        boolean out = false;
        int i1 = buscar (n1);
        int i2 = buscar (n2);
        if( i1 != -1 && i2 != -1 )
        {
          ady[i1][i2] = new Arco(p);
          ady[i2][i1] = ady[i1][i2];
          out = true;
        }
        return out;
    }
    // resto de la clase aquí....
}
```

5.] Presentación de problemas esenciales.

Una vez definido el contexto de trabajo con una jerarquía de clases que permitan manejar grafos de distintos tipos, surgen algunos problemas que son clásicos del mundo de los grafos, y para los cuales hemos

dado una solución en el modelo *Grafos [matricial]*. Analizaremos con algún detalle esos problemas, uno a uno, aunque muchos de estos problemas volverán a ser analizados en ocasión de ver la forma de implementar un grafo en forma de listas multiencadenadas.

a. Problema del cierre transitivo (algoritmo de Warshall).

En numerosas ocasiones se necesita saber si entre dos nodos *A* y *B* de un grafo hay un camino, de cualquier longitud que sea. Por ejemplo, si el grafo representa un plan de estudios de una carrera universitaria, un estudiante querrá saber si puede cursar la asignatura *B*, sabiendo que aún no tiene cursada la materia *A*. Esta consulta es equivalente a preguntar si existe un camino de cualquier longitud que comience en *A* y termine en *B*: si tal camino existe, el alumno no puede cursar la materia *B* (pues *A* es previa de *B*).

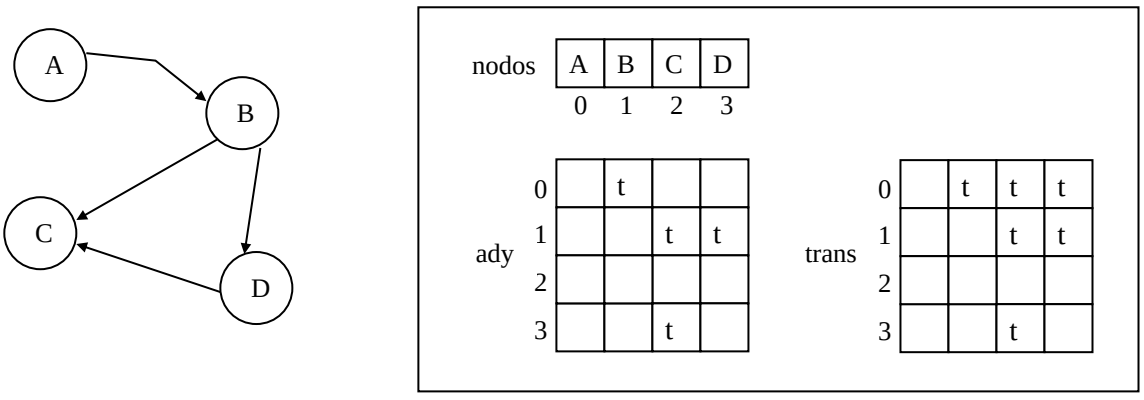
Nos interesa plantear un método que pueda "dejar la cosas planteadas" de tal modo que se permitan consultas de camino para cualquier par de nodos *A* y *B*, sin tener que recalcular la existencia del camino cada vez. En ese sentido, pensemos que la matriz de adyacencias *ady* resuelve el problema si la consulta fuera determinar la existencia de *caminos de longitud k = 1*: todos los caminos de longitud 1 están informados por *ady*, y sólo necesitamos saber el nodo de partida y el de llegada. Así, una consulta de caminos de longitud 1 puede hacerse en tiempo constante, cualesquiera sean *A* y *B*.

Siguiendo esa idea, la solución a nuestro problema más general es plantear una segunda matriz, a la que llamaremos *trans*, del mismo orden que *ady*, pero que se use para dejar indicada la existencia de caminos de *cualquier* longitud entre los nodos del grafo. En principio, esa matriz contendrá simplemente valores *boolean*. Así, si el casillero *trans[x][y]* vale *true*, eso significará que existe un camino que parte del nodo *x* y llega al nodo *y*, aunque no sabremos de qué longitud es ese camino (y en la mayoría de los problemas no importará). El cálculo de la matriz *trans* se conoce como *problema del cierre transitivo de un grafo*, y la matriz *trans* se designa como *matriz de cierre transitivo* del grafo dado.`

En el siguiente modelo (*Figura 9*) mostramos un grafo dirigido sencillo. En la matriz de adyacencias estamos mostrando simplemente valores *boolean* para indicar la existencia de arcos, y en la matriz *trans* mostramos su cierre transitivo:

Un conocido método para calcular la matriz de cierre, se conoce como *algoritmo de Warshall*, en honor a su descubridor (*Stephen Warshall*). La idea es simple: se comienza inicializando la matriz *trans* con todos los caminos de longitud 1 (que están ya informados en *ady*). Luego se trabaja sólo con *trans*, teniendo en cuenta una observación obvia: si existe un camino que conecte al nodo *x* con el nodo *y*, y también existe un camino que conecte a *y* con *z*, entonces existe el camino que conecta a *x* con *z*. Una relación de este tipo entre tres nodos, se conoce como una *relación transitiva* (y de allí el nombre de la matriz: se trata de almacenar en ella la información de todas las relaciones transitivas que hay en el grafo)

Figura 9: un grafo dirigido y su matriz de cierre transitivo



En la clase *Grafo* hemos incluido el método *cierre()* que aplica el *algoritmo de Warshall* para calcular el cierre transitivo, dejando el mismo en la matriz *trans* que está definida como un atributo de la clase. El primer par de ciclos, simplemente copia el contenido de *ady* en *trans*. Y la segunda parte del algoritmo busca las relaciones transitivas, actualizando el contenido de *trans*. Como ese proceso involucra tres ciclos anidados, cada uno con *n* repeticiones, es fácil ver que el tiempo de ejecución es $O(n^3)$. Note el uso del operador *|| (or)* para actualizar el valor de *trans*: se está haciendo una *suma lógica* entre las casillas *trans[i][j]* y *trans[k][j]*:

```
public void cierre ()
{
  int i, j, k, n = nodos.length;
  for (i = 0; i<n; i++)
  {
    for (j = 0; j < n; j++)
    {
      trans[i][j] = ady[i][j].exists();
    }
  }

  for (k = 0; k < n; k++)
  {
    for (i = 0; i < n; i++)
    {
        if ( trans[i][k] )
        {
          for (j = 0; j < n; j++)
          {
                trans[i][j] = trans[i][j] || trans[k][j];
          }
        }
    }
  }
}
```

b. Problema del recorrido de un grafo en el orden de sus arcos.

Otro problema de aparición frecuente en el mundo de los grafos, es el de recorrer el grafo y procesar sus vértices, pero en el orden indicado por sus arcos. Está claro que si sólo queremos mostrar los vértices sin importar como están relacionados, bastaría con recorrer el vector de nodos. Pero en la mayor parte de los problemas para los cuales se modela un grafo, lo que interesa es procesar los vértices siguiendo el orden que surge de sus relaciones.

Existen varios pequeños inconvenientes que debemos analizar: el primero es por dónde comenzar el recorrido. En teoría, un grafo no tiene lo que llamaríamos un vértice inicial, por lo que un algoritmo de recorrido deberá empezar por un vértice cualquiera. Y esto es correcto: no hay ningún inconveniente en comenzar un recorrido por cualquier vértice que querramos. En la práctica, y por obvias razones de sencillez de implementación, la mayoría de los recorridos comienza por el vértice que está en la casilla cero del vector de nodos.

Otro incoveniente es que el grafo podría tener varios arcos que pasen por el mismo vértice, y posiblemente no querramos procesar a cada vértice más de una vez. ¿Cómo evitar que un nodo que ya fue procesado, sea procesado nuevamente si otro camino nos lleva otra vez a él? Una solución simple consiste en usar un vector de valores *boolean*, cuyos casilleros se usen para marcar si un nodo ya procesado o visitado (valor *true*) o aún no lo fue (valor *false*). El vector tendrá tantos casilleros como vértices el grafo, y el casillero *i* se usará para marcar al vértice que está en la casilla *i* del vector de nodos.

El mismo vector puede usarse para resolver otro problema: el grafo podría tener varias componentes conexas (subgrafos no conectados entre sí). Si el recorrido comienza por un vértice particular y se sigue el orden de sus arcos, sólo serán alcanzados los vértices que estén en la misma componente conexa que el primero. ¿Cómo hacer para recorrer los vértices que están en otras componentes? Se usa el mismo vector para saber si algún vértice sigue sin ser procesado, y en ese caso el recorrido comienza otra vez, seleccionando otro vértice. Cuando todos los vértices del grafo estén marcados como ya procesados, el recorrido habrá terminado.

Son varios los problemas que requerirían del recorrido de un grafo, por lo cual la implementación puntual del recorrido se hace en cada método que resuelve cada problema. Sin embargo, mostramos aquí la idea esencial, a modo de plantilla o modelo general:

```
public void recorrer ()
{
   int k, j, n = nodos.length;

   // el vector de banderas de visita...
   boolean [ ] visitados = new boolean [ n ];

   // ningún nodo fue visitado aún...
   for (k=0; k<n; k++) visitados[k] = false;

   // controlamos uno a uno si algún nodo está sin visitar…
   for (k=0; k<n; k++)
   {
      // si k está sin visitar...
           if ( ! visitados[k] )
           {
             // ...un recorrido desde él hacia sus nodos conectados...
             visitar(k, visitados);
           }
   }
}

// método recursivo: avanza en profundidad usando la pila implícita...
private void visitar ( int k,  boolean [ ] visitados )
{
        int t, n = nodos.length;

        // marcamos al nodo k como ya procesado…
        // ... y aquí podríamos programar el proceso que hay que hacer con k...
        // ... como mostrarlo, contarlo, sumarlo, etc.
        visitados[k] = true;

        // ahora buscamos los arcos que salen del nodo k...
        for (t=0; t<n; t++)
        {
            // … si encuentro un arco hacia el nodo t...
            if( ady[k][t].exists() )
            {
                 // ... y t está sin visitar...
                if ( ! visitados[t] )
                {
                        // ... entonces comienzo un recorrido recursivo desde t
                        visitar(t, visitados);
                }
            }
        }
}
```

Note que el algoritmo general mostrado antes, produce lo que se conoce como un *recorrido en profundidad*: se comienza con un vértice cualquiera, y el mecanismo recursivo va tomando el próximo vértice a visitar de forma de alejarse lo más posible del inicial antes de procesar a otro vecino próximo. Un ejemplo de recorrido en profundidad son los conocidos algoritmos de recorrido en *preorden*, *entreorden* y *postorden* para un árbol binario: en todos ellos, se comienza por la raiz y el proceso se aleja hasta al nodo más profundo del lado izquierdo, para luego regresar por el camino inverso y procesar en la misma forma a cada subárbol. Este efecto de alejamiento se produce por el hecho de que la recursividad usa implícitamente una pila para ir almacenando las direcciones de retorno y los parámetros que el método de recorrido toma, lo cual es aprovechado por los programadores para "recordar" la secuencia de regreso.

Sin embargo, los recorridos no tendrían porqué ser obligatoriamente en profundidad. Podemos pensar en un esquema que en lugar de comenzar por un vértice y se vaya alejando, comience por ese vértice y procese a todos los vecinos próximos antes de alejarse. Este recorrido sería el equivalente a procesar un árbol binario por niveles (cosa que no hemos implementado...) y se denomina *recorrido en amplitud*.

Para implementar un recorrido en amplitud, la idea es usar una cola en lugar de una pila, y almacenar en esa cola los vértices que siguen en el recorrido, que es lo mismo que hacemos con la pila implícita en la recursión. Pero si terminamos haciendo lo mismo, ¿cuál es la diferencia? Al usar una cola, los nodos se procesan en el mismo orden en que se guardan, mientras que al usar una pila esos nodos se procesan en orden invertido (y es lo que provoca que el efecto de ida hasta el fondo y luego de regreso en orden inverso).

Un proceso recursivo usa un pila de manera automática e implícita, favoreciendo el trabajo de los programadores que no deben pensar en los detalles de cómo implementar y usar esa pila. El problema es que los lenguajes de programación actuales no implementan recursividad basada en colas... y entonces, si queremos usar una cola en lugar de una pila para un recorrido, debemos implementar la clase *Cola* y hacernos cargo explícitamente de manejar esa cola, guardar en ella los nodos a procesar en el orden que necesitemos, y extraer de esa cola esos nodos cuando les toque ser procesados... lo cual implica eliminar el proceso recursivo y transparentar el mecanismo cíclico que la recursión mantiene oculto.

Si suponemos que la clase *Cola* está implementada (cosa que efectivamente es así en nuestro modelo *Grafos [matricial]*, aunque en forma muy simple) entonces el siguiente esquema general muestra la forma de hacer un recorrido en amplitud (note que el método *recorrer()* que hace el control inicial del vector *visitados*, es estrictamente el mismo que en el recorrido en profundidad: ese método invocar al método *visitar()*, y es este último el que internamente usa una pila o una cola para avanzar...):

```
public void recorrer ()
{
   int k, j, n = nodos.length;

   // el vector de banderas de visita...
   boolean [ ] visitados = new boolean [ n ];

   // ningún nodo fue visitado aún...
   for (k=0; k<n; k++) visitados[k] = false;

   // controlamos uno a uno si algún nodo está sin visitar…
   for (k=0; k<n; k++)
   {
            // si k está sin visitar...
            if ( ! visitados[k] )
            {
                     // ...un recorrido desde él hacia sus nodos conectados...
                     visitar(k, visitados);
            }
   }
}
```

```
// método NO recursivo, y usa una cola para avanzar en amplitud...
private void visitar (int k, boolean [ ] visitados)
{
  int t;
  Soporte struct = new Queue();
  // Soporte struct = new Stack();

  struct.add(k);
  while(!struct.empty())
  {
          k = struct.remove();

          // System.out.print(k);

          visitados[k] = true;
          for(t = 0; t<nodos.length; t++)
          {
            if (ady[k][t].exists())
            {
              // no había sido encontrado aún...
              if (!visitados[t])
              {
                visitados[t] = true;
                        struct.add(t);
              }
            }
          }
  }
}
```

Si bien hemos dejado el análisis de los detalles para el alumno, es importante destacar un hecho notable: el método *visitar()* ha eliminado la recursión y reemplazó la pila implícita por una cola. Sin embargo, note que si en ese mismo método se deja todo como está pero se reemplaza la instancia de la clase Cola por una instancia de una clase Pila (ahora manejada en forma explícita), el algoritmo sigue funcionando... ¡pero recorrerá el grafo en profundidad!

c. Problema del conteo de componentes conexas.

En ocasiones será necesario poder determinar si un grafo es conexo o no. Es decir, querremos saber si todos sus nodos están unidos en una única gran componente conexa o no. Por ejemplo, dado un grafo no dirigido que representa ciudades y caminos de una región dada, querremos saber si todas esas ciudades son alcanzables desde cualquiera de ellas, o si por el contrario existen algunas que forman un subsistema separado. En definitiva, necesitamos saber cuántas componentes conexas tiene el grafo.

El hecho es que como ya sabemos recorrer un grafo en el orden de sus arcos, la solución al problema del conteo de componentes conexas es trivial: en el método no recursivo que lanza el recorrido controlando el vector visitados, usamos un contador que simplemente se incremente en uno cada vez que el ciclo encuentra un nodo que no haya sido visitado, y se retorna ese contador al final del proceso. La clase *GrafoNoDirigido* implementa esta idea en el método *contarConexasEnProfundidad()* (que como su nombre lo indica, recorre el grafo en profundidad):

```
public int contarConexasEnProfundidad ()
{
  int k, cx = 0, n = nodos.length;
  boolean [ ]visitados = new boolean[n];

  //aplica el recorrido...
  for (k=0; k<n; k++)
  {
```

```
    if ( ! visitados[k] )
    {
      // encontré una nueva componente conexa...
      // ... la cuento y la recorro en profundidad...
      cx++;
      visitarEnProfundidad(k, visitados);
    }
  }
  return cx;
}

private void visitarEnProfundidad (int k, boolean [ ] visitados)
{
  int t, n = nodos.length;
  visitados[k] = true;
  for (t=0; t<n; t++)
  {
    if( ady[k][t].exists() )
    {
      if ( ! visitados[t] ) visitarEnProfundidad(t, visitados);
    }
  }
}
```

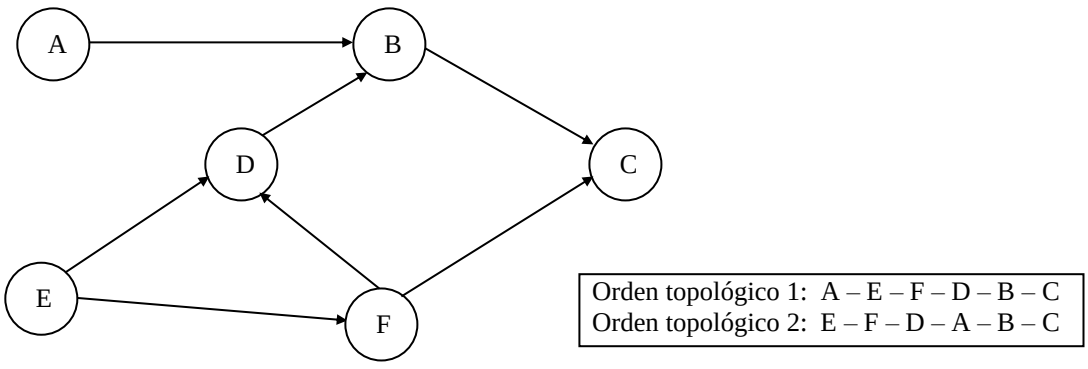
La misma clase *GrafoNoDirigido* incluye otro método *contarConexasEnAmplitud()*, que también cuenta las componentes conexas pero recorre el grafo en amplitud. Los resultados son, obviamente, los mismos.

d. Problema del orden topológico.

Cuando se usan grafos dirigidos es porque interesa el orden de los arcos. Es decir, en el problema analizado importa saber en qué dirección debe entenderse la relación entre dos vértices. Y en muchos de estos problemas, suele surgir la necesidad de saber si los vértices del grafo podrían ser procesados en un orden tal, que cuando se procese el nodo *x* se hayan procesado antes todos los nodos que se encontraban en cualquier camino que termine en *x*. Por ejemplo, si un grafo dirigido representa un plan estudios (o plan de correlatividades) de una carrera universitaria, podría surgir el requerimiento de saber un posible orden en que las materias podrían cursarse, sin violar reglas de correlatividad previa. Si tal orden es posible en un grafo dirigido, se dice que el grafo tiene un *orden topológico*.

Es evidente que si el grafo tiene al menos un ciclo, entonces no hay orden topológico posible para el grafo. Y note también que un grafo dirgido acíclico podría tener más de un orden topológico válido. En la siguiente figura se muestra un grafo dirigido acíclico, y dos órdenes topológicos válidos para el mismo:

Figura 10: un grafo dirigido y dos posibles recorridos topológicos para el mismo



Para determinar un posible orden topológico en un grafo, se puede usar un recorrido en profundidad implementado recursivamente, y prestar atención a un detalle sutil: cada vez que el método recursivo termina una de sus ejecuciones, desaloja de la pila implícita al vértice *k* que tomó parámetro. Para esa instancia de ejecución actual del método, ese vértice *k* era el principio de un camino. Pero si esa instancia de ejecución terminó, significa una de dos cosas posibles: 1) que el camino que comenzaba en *k* ya fue recorrido en forma completa, o 2) que ese nodo *k* era un vértice terminal y ningún camino salía de él. En ambos casos, significará que ya no encontraremos ningún vértice despúes de *k*... y por lo tanto *k* puede ser incluido en el orden topológico como *el final* de un camino.

Para hacer esto, usaremos un vector *sort* que contendrá a los vértices, pero en orden topológico. El vector comienza vacío. Cada vez que el método de recorrido recursivo esté por terminar, se almacenará en *sort* el vértice actual, pero comenzando desde la derecha del vector *sort* (debido a que el nodo actual, como dijimos, es *el final* de un camino). En otras palabras, el vector *sort* se llena de derecha a izquierda, mediante el procesamiento en *postorden* de los vértices en el método recursivo. El método que calcula el orden topológico finalmente retorna el vector *sort*, que puede ser mostrado o procesado normalmente, de izquierda a derecha para obtener el orden topológico. La clase *Grafo* implementa esa idea en el método *ordenTopologico()*:

```
public Object [ ] ordenTopologico ()
{
  // si existe un ciclo, el grafo no tiene orden topológico...
  if ( ciclos() ) return null;

  int k,j, n = nodos.length;
  indice = n - 1 ; // indice está definida como privada en la clase
  boolean [ ] visitados = new boolean [ n ];
  Object [ ] sort   = new Object[ n ]; // el orden topológico

  // ningún nodo fue visitado aún...
  for (k=0; k<n; k++) visitados[k] = false;

  for (k=0; k<n; k++)
  {
    if ( ! visitados[k] )
    {
       visitar1(k, visitados, sort);
    }
  }
  return sort;
}

private void visitar1 (int k, boolean [ ] visitados, Object [ ] sort)
{
  int t, n = nodos.length;
  visitados[k] = true;
        for (t=0; t<n; t++)
        {
            if( ady[k][t].exists() )
            {
                if ( ! visitados[t] ) visitar1(t, visitados, sort);
            }
        }

        //  k está por salir de la pila... guardarlo en el vector sort.
        sort[indice] = nodos[k];
        indice--;
}
```

e. Problema del árbol de expansión mínimo (algoritmo de Prim).

En general, se define como un *árbol* (o más general aún, un *árbol libre*) a un grafo que no tiene ciclos pero tiene conectados a todos sus vértices, y un rápido examen de esta definición muestra que entonces un árbol es un grafo que tiene la mínima cantidad de arcos posibles para manterlo conexo.

Muchas veces un grafo ponderado conexo y no dirigido es denso o incluso completo y en algunas situaciones podría ser necesario reducir el grafo a su "minima expresión" en cuanto a cantidad de arcos. Podría ser necesario, por ejemplo, crear otro grafo que sea un árbol con los mismos nodos que el grafo original, pero sólo los arcos del original que mínimamente sean necesarios para que el grafo siga siendo conexo. Ese segundo grafo comúnmente se designa como *árbol de expansión* del grafo original, o también *árbol recubridor* o *árbol abarcador*. Note que un grafo podría tener varios árboles de expansión distintos y válidos (ver gráficos en la página siguiente).

Si el árbol de expansión es tal que la suma de los pesos de sus arcos es la *mínima posible*, entonces el árbol de expansión se dice *árbol de expansión mínimo*. Note que un grafo podría tener varios árboles de expansión mínimos y válidos. La *Figura 11* muestra un grafo ponderado, conexo y no dirigido, junto a un árbol de expansión posible y un árbol de expansión mínimo.

Un algoritmo que calcula un árbol de expansión mínimo para un grafo puede basarse en una idea o propiedad intuitivamente simple: si *g1* y *g2* son dos subconjuntos de nodos del grafo original, entonces *el arco con menor peso* del grafo original que permite unir algún nodo de *g1* con algún nodo de *g2*, pertenece a un árbol de expansión mínimo. Esta propiedad es demostrable (aunque no formulamos aquí esa demostración), y se puede aplicar iterativamente para ir armando arco por arco el árbol de expansión mínimo.

El algoritmo resultante fue ideado por *Robert Prim* y se conoce por ello como *algoritmo de Prim*. Se comienza haciendo que el subconjunto *g1* contenga sólo a un nodo *x* del grafo, y se toma a *g2* como conteniendo al resto del grafo. Luego se toma el arco con menor peso que permite unir a *x* con algún nodo *y* de *g2*. Ese arco pertenece al árbol de expansión mínimo (que se va construyendo dentro de *g1*). Por lo tanto, a partir de ahora se asume que *g1* contiene a los nodos *x* e *y*, unidos por ese arco. Se aplica nuevamente el mecanismo, pero ahora se busca el arco con menor peso que permita unir a *x* o a *y* con algún otro nodo en *g2*. Y así se continúa hasta que todos los nodos de *g2* pasaron a *g1*. El conjunto *g1* finalmente armado, es un árbol de expansión mínimo.

En el modelo *Grafos [matricial]* la clase *GrafoNoDirigido* incluye el método *buscarAEM()* que aplica estas ideas (en forma algo compleja...) y calcula un árbol de expansión mínimo. El método simplemente retorna una cadena con ese árbol convertido a *String*, listo para ser visualizado. Si en la implementación del algoritmo se aplica alguna técnica veloz (de orden logarítmico) para encontrar el siguiente arco de menor peso entre todos los que restan (por ejemplo, usando un heap), entonces el algoritmo tiene tiempo de ejecución *O(m * log(n))*, siendo *m* el número de arcos del grafo, y *n* el número de vértices. Dejamos el análisis del código fuente para los alumnos, aunque hacemos notar que volveremos sobre este algoritmo, pero para implementarlo con detalle en grafos soportados en listas multiencadenadas.

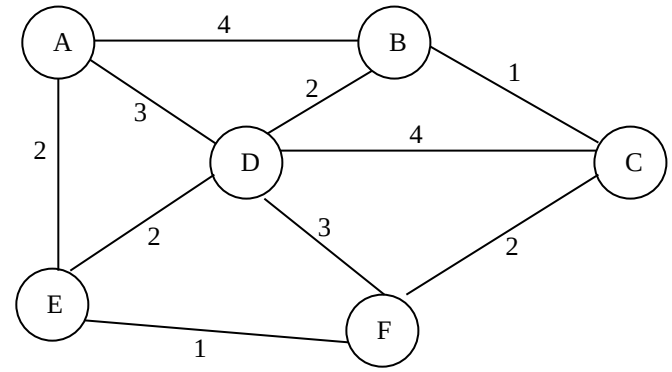
f. Problema del "camino más corto" de un nodo a cada uno de los otros (algoritmo de Dijkstra).

Un problema común (también propio de los grafos ponderados, conexos y no dirigidos) es el de calcular todos los "caminos más cortos" (esto es, los caminos cuya suma de pesos sea mínima) desde un vértice *x* dado, hasta los otros vértices del grafo. Si se hace un análisis detallado del requerimiento, se puede llegar a la conclusión que este problema puede entenderse como un caso particular del algoritmo de *Prim*: al fin y al cabo, también aquí debemos buscar caminos cuya suma de pesos sea mínima, con la única diferencia que ahora debemos centrar toda la búsqueda comenzando en el mismo vértice.

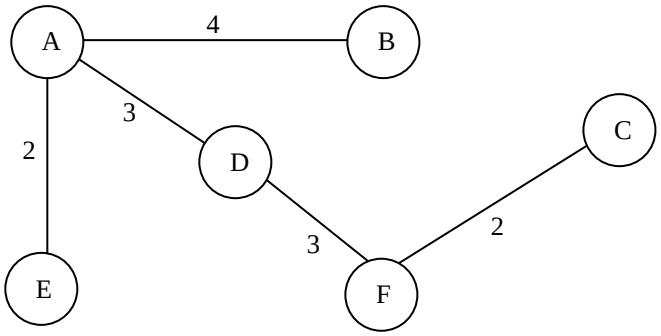
Un planteo del algoritmo con estas ideas y variaciones sobre el algoritmo de *Prim*, fue planteada por *Edsger Dijkstra* y se conoce como *algoritmo de Dijkstra*. La clase *GrafoNoDirigido* del modelo *Grafos [matricial]* incluye el método *buscarCMC()* que implementa lo dicho. El método toma como parámetro el vértice desde el cual comenzará la búsqueda de caminos de peso mínimo y retorna una cadena con la

información sobre los caminos de peso mínimo encontrados. Dejamos aquí también el análisis del código fuente para los alumnos, e indicamos que si el algoritmo se implementa de forma de poder encontrar el siguiente arco de peso mínimo en tiempo logarítmico, entonces el tiempo de ejecución total para el peor caso será *O(m + n * log(n))*, siendo *m* el número de arcos y *n* el número de vértices.

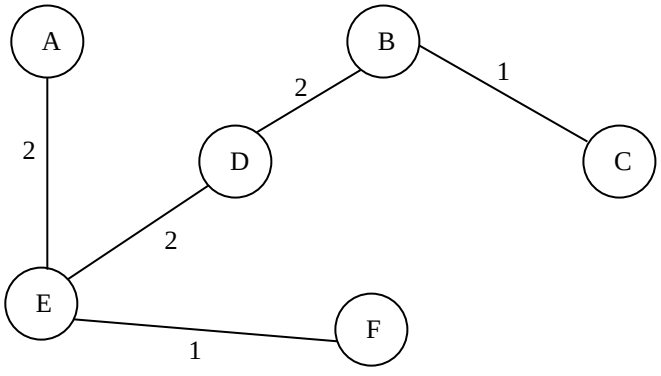
Figura 11: un grafo con algunos árboles de expansión



Grafo original: ponderado, conexo y no dirigido



Un árbol de expansión válido para el grafo original (suma de pesos: 14)



Ùn árbol de expansión mínimo para el grafo original (suma de pesos: 8) (mínima)