TEMA 6

GRAFOS

6.1. Fundamentos y terminología básica

Un grafo, **G**, es un par, compuesto por dos conjuntos **V** y **A**. Al conjunto **V** se le llama conjunto de vértices o nodos del grafo. **A** es un conjunto de pares de vértices, estos pares se conocen habitualmente con el nombre de arcos o ejes del grafo. Se suele utilizar la notación **G = (V, A)** para identificar un grafo.

Existen dos clase de grafos: dirigidos y no dirigidos. En un grafo no dirigido el par de vértices que representa un arco no está ordenado. Por lo tanto, los pares **(v1, v2)** y **(v2, v1)** representan el mismo arco. En un grafo dirigido cada arco está representado por un par ordenado de vértices **<v1,v2>**, de forma que **<v1,v2>** y **<v2,v1>** representan dos arcos diferentes.

Ejemplos de grafos (dirigidos y no dirigidos):

G1 = (V1,A1)
V1={1,2,3,4} A1={ (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4) }

G2 = (V2,A2)
V2={1,2,3,4,5,6} A2={ (1, 2), (1, 3), (2, 4), (2, 5), (3, 6) }

G3 = (V3,A3)
V3={1,2,3} A3={ <1, 2>, <2, 1>, <2, 3> }

Gráficamente estas tres estructuras de vértices y arcos se pueden representar de la siguiente manera:

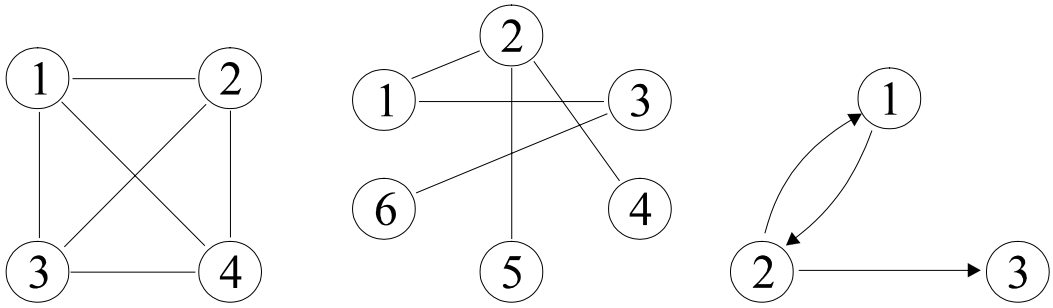


Figura 1. Ejemplos de grafos

Los grafos permiten representar conjuntos de objetos arbitrariamente relacionados. Se puede asociar el conjunto el vértices con el conjunto de objetos y el conjunto de arcos con las relaciones que se establecen entre ellos.

Los grafos son modelos matemáticos de numerosas situaciones reales: un mapa de carreteras, la red de ferrocarriles, el plano de un circuito eléctrico, el esquema de la red telefónica de una compañía, etc.

El número de distintos pares de vértices **(vi,vj)**, con **vi ≠ vj**, en un grafo con **n** vértices es **n(n-1)/2**. Este el número máximo de arcos en un grafo no dirirgido de **n** vértices. Un grafo no dirigido que tenga exactamente **n(n-1)/2** arcos se dice que un grafo completo. En el caso de un grafo dirigido de **n** vértices el número máximo de arcos es **n(n-1)**.

Algunas definiciones básicas en grafos:

- **Orden de un grafo:** Es el número de nodos (vértices) del grafo.
- **Grado de un nodo:** Es el número de ejes (arcos) que inciden sobre el nodo.
- **Grafo simétrico:** Es un grafo dirigido tal que si existe la relación **<u, v>** entonces existe **<v, u>**, con **u,v ∈ V**.
- **Grafo no simétrico:** Es un grafo que no cumple la propiedad anterior.
- **Grafo reflexivo:** Es el grafo que cumple que para todo nodo **u ∈ V** existe la relación **(u, u) ∈ A**.
- **Grafo transitivo:** Es aquél que cumple que si existen las relaciones **(u, v)** y **(v, z) ∈ A** entonces **(u,z) ∈ A**.
- **Grafo completo:** Es el grafo que contiene todos los posibles pares de relaciones, es decir, para cualquier par de nodos **u,v ∈ V**, **u ≠ v**,existe **(u,v) ∈ A**.
- **Camino:** Un camino en el grafo G es una sucesión de vértices y arcos: v_0, a_1, v_1, a_2, v_2, ..., a_k, v_k; tal que los extremos del arco a_i son los vértices v_{i-1} y v_i.
- **Longitud de un camino:** Es el número de arcos que componen el camino.
- **Camino cerrado (circuito):** Camino en el que coinciden los vértices extremos ($v_0 = v_k$).
- **Camino simple:** Camino donde sus vértices son distintos dos a dos, salvo a lo sumo los extremos v_0 y v_k.
- **Camino elemental:** Camino donde sus arcos son distintos dos a dos.
- **Camino euleriano:** Camino simple que contiene todos los arcos del grafo.
- **Grafo euleriano:** Es un grafo que tiene un camino euleriano cerrado.
- **Grafo conexo:** Es un grafo no dirigido tal que para cualquier par de nodos existe al menos un camino que los une.
- **Grafo fuertemente conexo:** Es un grafo dirigido tal que para cualquier par de nodos existe un camino que los une.

- **Punto de articulación:** Es un nodo que si desaparece provoca que se cree un grafo no conexo.

6.2. Representación de grafos

Existen tres maneras básicas de representar los grafos: Mediante matrices; mediante listas y mediante matrices dispersas. Cada representación tiene unas ciertas ventajas e inconvenientes respecto de las demás, que comentaremos más adelante.

Representación mediante matrices: Matrices de adyacencia

Un grafo es un par compuesto por dos conjuntos: Un conjunto de nodos; y un conjunto de relaciones entre los nodos.

La representación que realicemos en C++ tendrá que ser capaz de guardar esta información en memoria.

La forma más fácil de guardar la información de los nodos es mediante la utilización de un vector que indexe los nodos, de manera que los arcos entre los nodos se pueden ver como relaciones entre los índices. Esta relación entre índices se pueden guardar en una matriz, que llamaremos de adyacencia.

En base a esto, un grafo se puede representar en C++ como:

```
const int MAX_NODOS = .??.;

typedef .??. Valor_Nodo;
typedef .??. Valor_Arco;

struct Arco
{
   Valor_Arco Info;    /* Información asociada a cada arco */
   bool Existe;
};

struct Nodo
{
   Valor_Nodo Info;    /* Información asociada a cada nodo */
   bool Existe;
};

class Grafo
{
   public:
      ...

   private:
      Nodo Nodos [MAX_NODOS];
      Arco Arcos [MAX_NODOS][MAX_NODOS];
};
```

Con esta representación tendremos que reservar al menos del orden de n^2 'espacios' de memoria para la información de los arcos, y las operaciones relacionadas con el grafo implicarán, habitualmente, recorrer toda la matriz, con lo que el orden de las operaciones será, en general, cuadrático, aunque tengamos un número de relaciones entre los nodos mucho menor que n^2.

En cambio, con esta representación es muy fácil determinar, a partir de dos nodos, si están o no relacionados: Sólo hay que acceder al elemento adecuado de la matriz y comprobar el valor que guarda.

Ejemplo de representación:

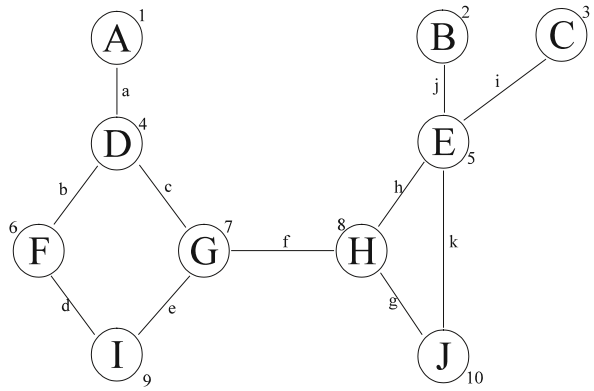


Figura 2. Ejemplo de grafo no dirigido

Supongamos el grafo representado en la figura 2. A partir de ese grafo la información que guardariamos, con esta representación, sería:

GRAFO:

Nodos

	1	2	3	4	5	6	7	8	9	10	11	12
Existe	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*F*	*F*
Info	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*?*	*?*

Arcos

Existe /Info	1	2	3	4	5	6	7	8	9	10	11	12
1	*F/?*	*F/?*	*F/?*	*V/a*	*F/?*	*F/?*	*F/?*	*F/?*	*F/?*	*F/?*		
2	*F/?*	*F/?*	*F/?*	*F/?*	*V/j*	*F/?*	*F/?*	*F/?*	*F/?*	*F/?*		
3	*F/?*	*F/?*	*F/?*	*F/?*	*V/i*	*F/?*	*F/?*	*F/?*	*F/?*	*F/?*		
4	*V/a*	*F/?*	*F/?*	*F/?*	*F/?*	*V/b*	*V/c*	*F/?*	*F/?*	*F/?*		
5	*F/?*	*V/j*	*V/i*	*F/?*	*F/?*	*F/?*	*F/?*	*V/h*	*F/?*	*V/k*		
6	*F/?*	*F/?*	*F/?*	*V/b*	*F/?*	*F/?*	*F/?*	*F/?*	*V/d*	*F/?*		
7	*F/?*	*F/?*	*F/?*	*V/c*	*F/?*	*F/?*	*F/?*	*V/f*	*V/e*	*F/?*		
8	*F/?*	*F/?*	*F/?*	*F/?*	*V/h*	*F/?*	*V/f*	*F/?*	*F/?*	*V/g*		
9	*F/?*	*F/?*	*F/?*	*F/?*	*F/?*	*V/d*	*Ve*	*F/?*	*F/?*	*F/?*		
10	*F/?*	*F/?*	*F/?*	*F/?*	*V/k*	*F/?*	*F/?*	*V/g*	*F/?*	*F/?*		
11												
12												

Representación mediante punteros: Listas de adyacencia

En las listas de adyacencia se intenta evitar justamente el reservar espacio para aquellos arcos que no continene ningún tipo de información. El sustituto obvio a los vectores con 'huecos' son las listas.

En las listas de adyacencia lo que haremos será guardar por cada nodo, además de la información que pueda contener el propio nodo, una lista dinámica con los nodos a los que se puede acceder desde él. La información de los nodos se puede guardar en un vector, al igual que antes, o en otra lista dinámica. Si elegimos la representación en un vector para los nodos, tendríamos la siguiente definición de grafo en C++:

```
const int MAX_NODOS = .??.;

typedef .??. Valor_Nodo;
typedef .??. Valor_Arco;

typedef struct Arco * Punt_Arco;

struct Arco
{
   Valor_Arco Info;    {* Información asociada a cada arco *}
   int Destino;
   Punt_Arco Sig_Arco;
}

struct Nodo
{
   Valor_Nodo Info; {* Información asociada a cada nodo *}
   bool Existe;
   Punt_Arco Lista_Arcos;
}

class Grafo
{
   public:
      ...

   private:
      Nodo Nodos [MAX_NODOS];
};
```

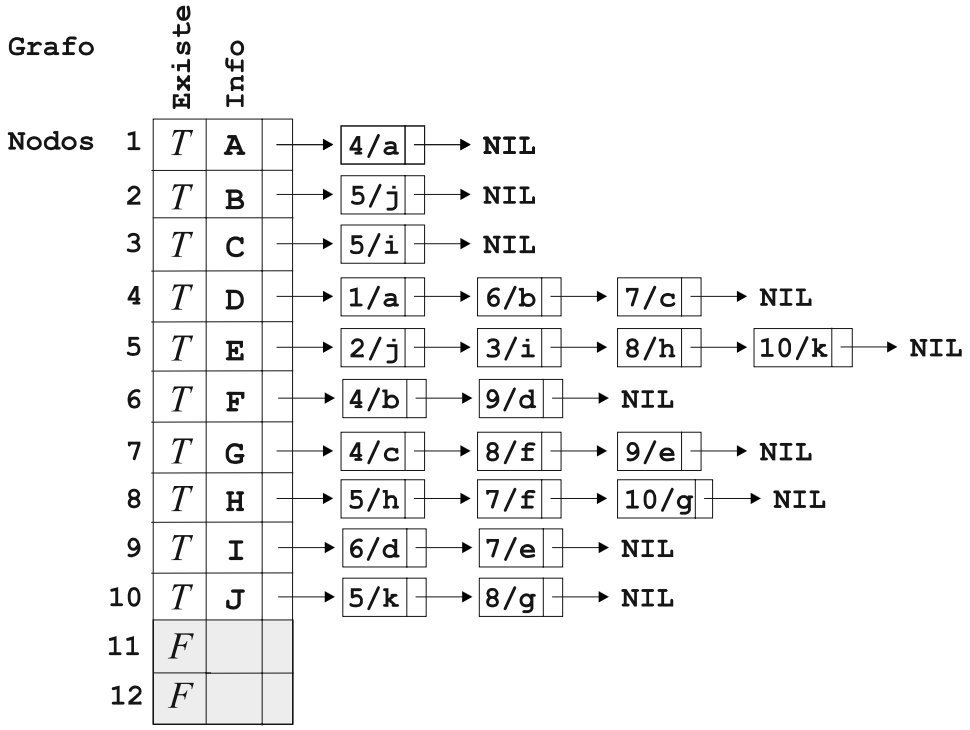
En general se está guardando menor catidad de elementos, sólo se reservará memoria para aquellos arcos que efectivamente existan, pero como contrapartida estamos guardando más espacio para cada uno de los arcos (estamos añadiendo el índice destino del arco y el puntero al siguiente elemento de la lista de arcos.)

Las tareas relacionadas con el recorrido del grafo supondrán sólo trabajar con los vértices existentes en el grafo, que puede ser mucho menor que $\mathbf{n^2}$. Pero comprobar las relaciones entre nodos no es tan directo como lo era en la matriz, sino que supone recorrer la lista de elementos adyacentes perteneciente al nodo analizado.

Además, sólo estamos guardando realmente la mitad de la información que guardábamos en el caso anterior, ya que las relaciones inversas (¿qué relaciones llegan a un cierto nodo?) en este caso no se guardan, y averiguarlas supone recorrer todas las listas de todos los nodos.

Ejemplo de representación:

La representación en esta estructura del grafo de la *figura 2* sería:



Representación mediante punteros: Matrices dispersas

Para evitar uno de los problemas que teníamos con las listas de adyacencia, que era la dificultad de obtener las relaciones inversas, podemos utilizar las matrices dispersas, que contienen tanta información como las matrices de adyacencia, pero, en principio, no ocupan tanta memoria como las matrices, ya que al igual que en las listas de adyacencia, sólo representaremos aquellos enlaces que existen en el grafo.

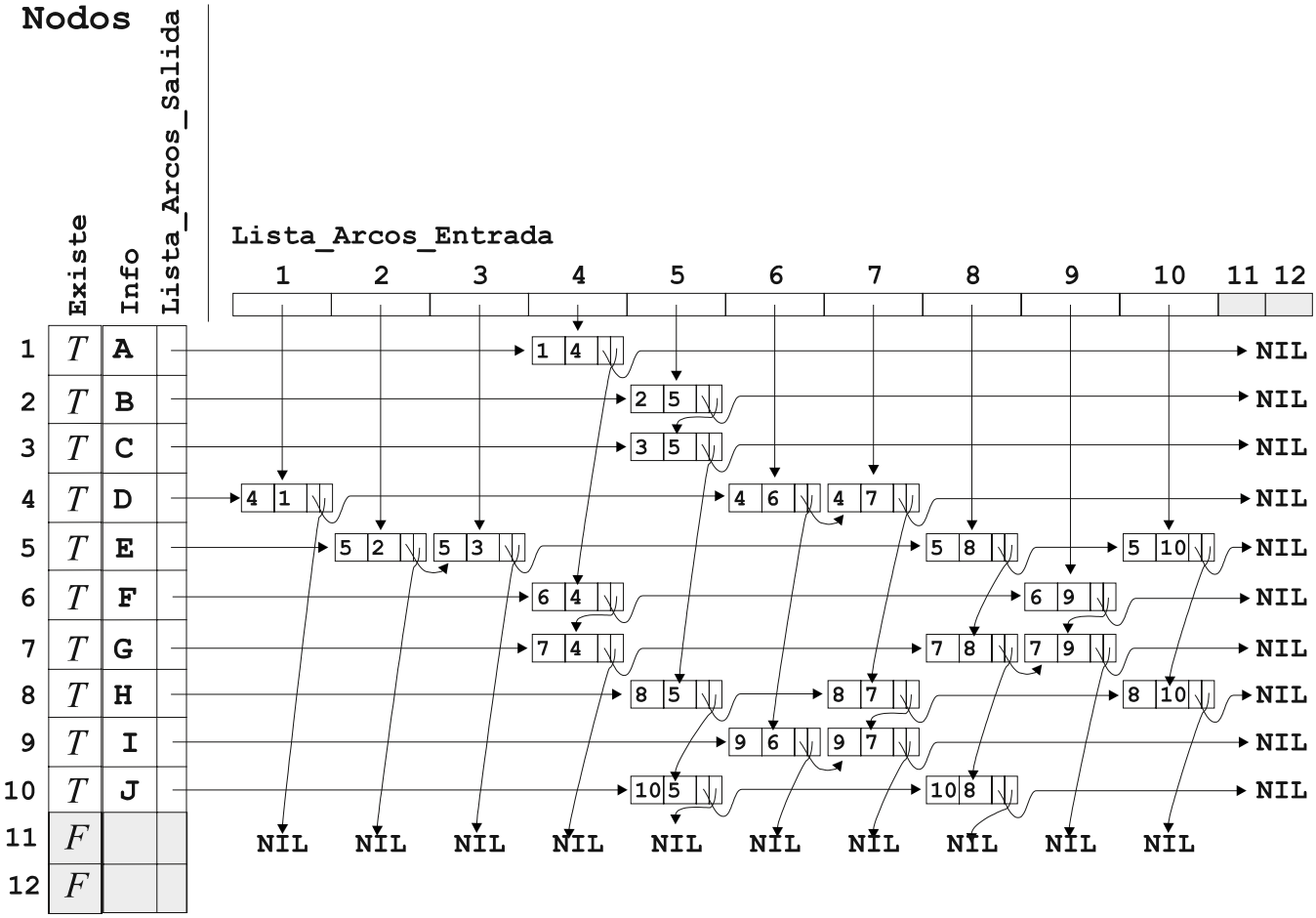
La estructura de datos en C++ para esta estructura será:

```
const int MAX_NODOS = .??.;

typedef Valor_Nodo = .??.;
typedef Valor_Arco = .??.;

typedef struct Arco * Punt_Arco;

struct Arco
{
   Valor_Arco Info;    /* Información asociada a cada arco */
   int Origen;
   int Destino;
   Punt_Arco Sig_Arco_Salida;
   Punt_Arco Sig_Arco_Entrada;
}

struct Nodo
{
   Valor_Nodo Info;    /* Información asociada a cada nodo */
   bool Existe;
   Punt_Arco Lista_Arcos_Salida;
   Punt_Arco Lista_Arcos_Entrada;
}

class Grafo
{
   public:
      ...

   private:
```

```
        Nodo Nodos[MAX_NODOS];
}
```

Ejemplo de representación:

La representación mediante matrices dispersas del grafo mostrado en la *figura 2* será:



6.3. Operaciones básicas con grafos

Las operaciones sobre grafos serán similares a las definidas para otros tipos de estructuras. Básicamente tendremos operaciones de creación de la estructura y de almacenamiento, consulta y eliminación de información en la estructura.

Las operaciones serán pues:

Iniciar la estructura:

Crear_Grafo -> Grafo

Consultas sobre el grafo:

Grafo_Vacio (Grafo) -> Booleano

Grafo_Lleno (Grafo) -> Booleano

Consultas sobre la información guardada en el grafo:

Numero_de_Nodos (Grafo) -> Entero

Numero_de_Arcos (Grafo) -> Entero

Obtener_Valor_Arco (Grafo, Arco) -> Valor_Arco

...

Métodos de modificación del grafo:

Añadir_Nodo (Grafo, Nodo) -> Grafo

Añadir_Arco (Grafo, Arco) -> Grafo
Eliminar_Nodo (Grafo, Nodo) -> Grafo
Eliminar_Arco (Grafo, Arco) -> Grafo

Con estas operaciones la parte pública de la clase nos quedará:

```
...

class Grafo
{
   public:
      Grafo (void);
      Grafo (const Grafo &)

      bool Grafo_Vacio (void);
      bool Grafo_Lleno (void);

      int Numero_de_Nodos (void);
      int Numero_de_Arcos (void);
      bool Obtener_Valor_Arco (Arco, Valor_Arco &);

      bool Anadir_Nodo (Valor_Nodo);
      bool Anadir_Arco (Valor_Nodo, Valor_Nodo, Valor_Arco);

      bool Eliminar_Nodo (Valor_Nodo);
      bool Eliminar_Arco (Valor_Nodo, Valor_Nodo);

   private:
      ...
};
```

Creación del grafo: Crear_Grafo

La creación del grafo supondrá tres pasos, al igual que en el resto de estructuras.

Un primer paso de definición de tipos capaces de soportar nuestra estructura. Un segundo paso de declaración de una variable de ese nuevo tipo. Y finalmente la aplicación de un subprograma que inicie la estructura a vacía.

Las definiciones en C++ ya han sido hechas. Faltaría implementar las funciones de iniciación para cada una de las implementaciones del grafo (con *arrays*, con listas y con matrices dispersas).

Iniciación del grafo: Matrices de adyacencia

La iniciación la haremos con el constructor por defecto de la clase:

```
Grafo::Grafo (void)
{
   int i, j;

   for (i = 0; i < MAX_NODOS; i++)
   {
      Nodos [i].Existe = false;

      /*
       * Realmente no haría falta iniciar a FALSE la matriz de arcos,
       * pero habría que tenerlo en cuenta a la hora de dar de alta
       * nuevos nodos
       */
       for (j = 0; j < MAX_NODOS; j++)
         Arcos [i][j].Existe = false;
```

```
   }
}
```

Iniciación del grafo: Listas de adyacencia

```
Grafo::Grafo (void)
{
   int i, j;

   for (i = 0; i < MAX_NODOS; i++)
   {
      Nodos [i].Existe = false;

      /*
       * Realmente no haría falta iniciar a NULL las listas de arcos,
       * pero habría que tenerlo en cuenta a la hora de dar de alta
       * nuevos nodos
       */
      Nodos [i].Lista_Arcos = NULL;
   }
}
```

Iniciación del grafo: Matrices dispersas

```
Grafo::Grafo (void)
{
   int i, j;

   for (i = 0; i < MAX_NODOS; i++)
   {
      Nodos [i].Existe = false;

      /*
       * Realmente no haría falta iniciar a NULL las listas de arcos
       * de salida y las listas de arcos de entrada pero habría que
       * tenerlo en cuenta a la hora de dar de alta nuevos nodos
       */
      Nodos [i].Lista_Arcos_Salida = NULL;
      Nodos [i].Lista_Arcos_Entrada = NULL;
   }
}
```

Añadir nodos al grafo: Anadir_Nodo

El proceso de añadir un nodo al grafo se limitará en todos los casos a marcar el nodo referido como existente, y a asignar un valor determinado a la información que contiene el nodo.

Añadir Nodo: Matrices de adyacencia, Listas de adyacencia, Matrices dispersas

```
bool Anadir_Nodo (Valor_Nodo x)
{
   int i_aux;
   bool b_aux;

   i_aux = 0;
   while ( (Nodos [i_aux].Existe == true) && (i_aux < MAX_NODOS) )
      i_aux++;

   if (i == MAX_NODOS)
      b_aux = false;
```

```
    else
    {
       b_aux = true;
       Nodos [i_aux].Existe := TRUE;
       gr.Nodos [ x.Nodo ].Info := x.Info;
    End;
End;
```

Añadir arcos al grafo: Anadir_Arco

Añadir un arco al grafo será una operación que dependerá fuertemente de la representación del grafo, aunque, en cualquier caso habrá que realizar las siguientes tareas:

1.- Mirar si los nodos origen y destino del arco existen, y el arco que queremos añadir no existe todavía en el grafo.

2.- Marcar como existente el arco (poniendo a TRUE el valor adecuado en la matriz de adyacencia, o creando el elemento adecuado de la lista en los casos de listas de adyacencia y matrices dispersas)

3.- Asignar el valor o peso adecuado al arco.

Matrices de adyacencia

```
bool Anadir_Arco (Valor_Nodo orig, Valor_Nodo dest, Valor_Arco x)
{
   int i_aux, j_aux;
   bool b_aux;

   i_aux = 0;
   while ( (i_aux < MAX_NODOS) && (Nodos [i_aux].Existe) &&
           (Nodos [i_aux].Info != orig) )
      i_aux++;

   if (i_aux == MAX_NODOS)
      b_aux = false;
   else
   {
      j_aux = 0;
      while ( (j_aux < MAX_NODOS) && (Nodos [j_aux].Existe) &&
              (Nodos [j_aux].Info != dest) )
         j_aux++;

      if (j_aux == MAX_NODOS)
         b_aux = false;
      else
      {
         if (Arcos [i_aux][j_aux].Existe)
            b_aux = false;
         else
         {
            b_aux = true;
            Arcos [i_aux][j_aux].Existe = true;
            Arcos [i_aux][j_aux].Info = x;
         }
      }
   }

   return b_aux;
}
```

Listas de adyacencia

```
bool Anadir_Arco (Valor_Nodo orig, Valor_Nodo dest, Valor_Arco x)
{
   int i_aux, j_aux;
   bool b_aux;

   i_aux = 0;
   while ( (i_aux < MAX_NODOS) && (Nodos [i_aux].Existe) &&
           (Nodos [i_aux].Info != orig) )
      i_aux++;

   if (i_aux == MAX_NODOS)
      b_aux = false;
   else
   {
      j_aux = 0;
      while ( (j_aux < MAX_NODOS) && (Nodos [j_aux].Existe) &&
              (Nodos [j_aux].Info != dest) )
         j_aux++;

      if (j_aux == MAX_NODOS)
         b_aux = false;
      else
      {
         p_aux = Nodos [i_aux].Lista_Arcos;
         while ( (p_aux != NULL) && (p_aux->Destino != j_aux) )
            p_aux = p_aux->Sig;

         if (p_aux == NULL)
            b_aux = false;
         else
         {
            b_aux = true;

            p_aux = new Arco;
            p_aux->Info = x;
            p_aux->Destino = j_aux;
            p_aux->Sig = Nodos [i_aux].Lista_Arcos;
            Nodos [i_aux].Lista_Arcos = p_aux->Sig;
         }
      }
   }

   return b_aux;
}
```

Matrices dispersas

```
Function Anadir_Arco ( Var gr: Grafo, x: Tipo_Arco ): Boolean;
Var
   aux_p : Punt_Arco;
   anadir: Boolean;
Begin
   If ( ( Not gr.Notos [ x.Origen ].Existe ) OR
        ( Not gr.Nodos [ x.Destino ].Existe ) ) Then
      anadir := FALSE;
   Else
   Begin
      aux_p := gr.Nodos [ x.Origen ].Lista_Arcos_Salida;
      While ( ( aux_p <> NIL ) AND ( aux_p.Destino <> x.Destino ) ) Do
         aux_p := aux_p^.Sig_Arco_Salida;
      If ( aux_p = NIL ) Then
```

```
            Anadir := TRUE
         Else
            Anadir := FALSE
      End;

      If ( Anadir = TRUE ) Then
      Begin
         New ( aux_p );
         aux_p.Info := x.Info;
         aux_p.Origen := x.Origen;
         aux_p.Destino := x.Destino;
         aux_p.Sig_Arco_Salida := gr.Nodos [ x.Origen ].Lista_Arcos_Salida;
         gr.Nodos [ x.Origen ].Lista_Arcos_Salida := aux_p;
         aux_p.Sig_Arco_Entrada := gr.Nodos [ x.Destino ].Lista_Arcos_Entrada;
         gr.Nodos [ x.Destino ].Lista_Arcos_Entrada := aux_p;
      End;
      Anadir_Arco := Anadir;
   End;
```

Eliminar nodos del grafo: Eliminar_Nodo

El problema de eliminar información del grafo es un poco más complejo que el almacenamiento de información. Eliminar un nodo supone, por un lado, decir que ese nodo ya no pertenece al grafo, y por otro, eliminar cualquier arco que tuviese como origen o como destino ese nodo.

Matrices de adyacencia

```
bool Eliminar_Nodo (Valor_Nodo x)
{
   int i_aux;
   bool b_aux;

   i_aux = 0;
   while ( (i_aux < MAX_NODOS) && (Nodos [i_aux].Existe) &&
           (Nodos [i_aux].Info != x) )
      i_aux++;

   if (i_aux == MAX_NODOS)
      b_aux = false;
   else
   {
      b_aux = true;

      Nodos [i_aux].Existe = false;
      for (j_aux = 0; j_aux < MAX_NODOS; j_aux++)
      {
         Arcos [i_aux, j_aux].Existe = false;
         Arcos [j_aux, i_aux].Existe = false;
      }
   }

   return b_aux;
}
```

Eliminar Nodo: Listas de adyacencia

Para eliminar los elementos de la lista de adyacencia nos apoyaremos en algunas operaciones vistas en el tema 4, Listas. Utilizaremos la función que elimina una cierta posición de la lista, **`Eliminar_Posicion`**, y la que nos dice si una lista está o no vacia, **`Lista_Vacia`**.

```
Function Eliminar_Nodo ( Var gr: Grafo; x: Tipo_Nodo );
```

```
Var
   aux_p1, aux_p2: Punt_Arco;
   aux_nod       : Indice;
   aux_b         : Boolean;
Begin
   If ( Not gr.Nodos [ x.Nodos ].Existe ) Then
      Eliminar_Nodo := FALSE
   Else
   Begin
      Eliminar_Nodo := TRUE;
      gr.Nodos [ x.Nodo ].Existe := FALSE;
      aux_p1 := gr.Nodos [ x.Nodo ].Lista_Arcos;
      While ( aux_p1 <> NIL ) Do
      Begin
         aux_p2 := aux_p1;
         aux_p1 := aux_p1^.Sig_Arco;
         Dispose ( aux_p2 );
      End;
      gr.Nodos [ x.Nodo ].Lista_Arcos := NIL;

      For aux_nod := 1 To MAX_NODOS Do
      Begin
         If ( ( gr.Nodos [ aux_nod ].Existe ) AND
              ( Not Lista_Vacia ( gr.Nodos [ aux_nod ].Lista_Arcos ) ) Then
         Begin
            aux_p1 := gr.Nodos [ aux_nod ].Lista_Arcos;
            While ( ( aux_p1 <> NIL ) AND ( aux_p1.Destino <> x.Destino ) ) Do
               aux_p1 := aux_p1^.Sig_Arco;
            If ( aux_p1.Destino = x.Destino ) Then
               aux_b := Elimina_Posicion ( gr.Nodos [ aux_nod ].Lista_Arcos,
                                           aux_p1 );
         End;
      End;
   End;
End;
```

Eliminar Nodo: Matrices dispersas

6.4. Otras Operaciones con grafos

Recorrido de grafos

Recorrer un grafo supone intentar alcanzar todos los nodos que estén relacionados con uno dado que tomaremos como nodo de salida.

Existen básicamente dos tácticas para recorrer un grafo: El recoorrido en anchura; y el recorrido en profundidad.

Recorrido en anchura o BFS (*Breadth First Search*)

El recorrido en anchura supone recorrer el grafo, a partir de un nodo dado, en niveles, es decir, primero los que están a una distancia de un arco del nodo de salida, después los que estan a dos arcos de distancia, y así sucesivamente hasta alcanzar todos los nodos a los que se pudiese llegar desde el nodo salida.

El algoritmo general de recorrido en anchura es el siguiente:

Algoritmo Recorrido_En_Anchura (BFS)

```
Entradas
        gr : Grafo                  {* Grafo a recorrer        *}
        nodo_salida: Indice         {* Origen del recorrido    *}
Variables
        queue: Cola de Indice
        aux_nod1, aux_nod2: Indice

Inicio
        Iniciar_Cola ( queue )
        Procesar ( nodo_salida )
        Visitado [ nodo_salida ] ← CIERTO
        Encolar ( queue, nodo_salida )
        Mientras no Cola_Vacia ( queue ) hacer
                aux_nod1 ← Desencolar ( queue )
                Para todos los nodos, aux_nod2, adyacentes a aux_nod1 hacer
                        Si no Visitado [ aux_nod2 ] entonces
                                Procesar ( aux_nod2 )
                                Visitado [ aux_nod2 ] ← CIERTO
                                Encolar ( queue, aux_nod2 )
                        fin_si
                fin_para
        fin_mientras
fin
```

La diferencia a la hora de implementar el algorimo general en Pascal para cada una de las implementaciones de la estructura de datos grafo, residirá en la manera de averiguar los diferentes nodos adyacentes a uno dado. En el caso de las matrices de adyacencia se tendrán que comprobar si los enlaces entre los nodos existen en la matriz. En los casos de las listas de adyacencia y de las matrices dispersas sólo habrá que recorrer las listas de enlaces que parten del nodo en cuestión para averiguar qué nodos son adyacentes al estudiado.

Recorrido en profundidad o DFS (*Depth First Search*)

A diferencia del algoritmo anterior, el recorrido en profundidad trata de buscar los caminos que parten desde el nodo de salida hasta que ya no es posible avanzar más. Cuando ya no puede avanzarse más sobre el camino elegido, se vuelve atrás en busca de caminos alternativos, que no se estudiaron previamente.

El algoritmo es similar al anterior, pero utilizando, para guardar los nodos accesibles desde uno dado una pila.

```
Algoritmo Recorrido_En_Profundidad (DFS)

Entradas
        gr : Grafo                  {* Grafo a recorrer        *}
        nodo_salida: Indice         {* Origen del recorrido    *}
Variables
        stack: Pila de Indice
        aux_nod1, aux_nod2: Indice

Inicio
        Iniciar_Pila ( stack )
        Procesar ( nodo_salida )
        Visitado [ nodo_salida ] ← CIERTO
        Apilar ( stack, nodo_salida )
        Mientras no Pila_Vacia ( stack ) hacer
                aux_nod1 ← Desapilar ( stack )
                Para todos los nodos, aux_nod2, adyacentes a aux_nod1 hacer
                        Si no Visitado [ aux_nod2 ] entonces
```

```
                        Procesar ( aux_nod2 )
                        Visitado [ aux_nod2 ] ← CIERTO
                        Encolar ( queue, aux_nod2 )
                    fin_si
                fin_para
            fin_mientras
fin
```

La utilización de la pila se puede sustituir por la utilización de la recurrencia, de manera que el algoritmo quedaría como sigue:

```
Algoritmo Recorrido_En_Profundidad (DFS)

Entradas
        gr : Grafo                  {* Grafo a recorrer          *}
        nodo_salida: Indice         {* Origen del recorrido      *}
Variables
        aux_nod2: Indice

Inicio
        Procesar ( nodo_salida )
        Visitado [ nodo_salida ] ← CIERTO
        Para todos los nodos, aux_nod2, adyacentes a nodo_salida hacer
            Si no Visitado [ aux_nod2 ] entonces
                    Recorrido_En_Profundidad ( gr, aux_nod2 )
            fin_si
        fin_para
fin
```

DETERMINACIÓN DE COMPONENTES CONEXAS

Componente conexa:

Subgrafo conexo máximo (mayor número posibles de vértices) de un grafo no dirigido.

```
Algoritmo Componentes

Entrada     G : Grafo
Variables   i : 1..n

Inicio
            desde i□1 hasta n hacer G[i].visitado□falso
            desde i□1 hasta n hacer :
                        si no G[i].visitado entonces:
                                    DFS(i)
                                    Escribir los vértices visitados en este
recorrido y los arcos
                        fin_si
            fin_desde
Fin
```

Árbol de expansión (arborescencia o spanning tree) de un grafo G:

Árbol formado con arcos de G y que contiene todos los vértices de G.

Un árbol de expansión de G es un subgrafo mínimo (menor número posible de arcos) conexo, G', de G tal que V(G')=V(G).

Árbol de expansión mínimo (minimal spanning tree) de un grafo G:

Árbol de expansión de un grafo con menor coste asociado.

Aplicaciones: Redes de comunicaciones.