TEMA 7
GRAFOS

- Modelo matemático y especificación
  - Técnicas de implementación
  - Matriz de adyacencia
  - Vector de listas/secuencias de adyacencia
  - Vector de multilistas de adyacencia
  - Implementación de las matrices de adyacencia
  - Implementación de las listas de adyacencia
- Recorridos de grafos
  - Recorrido en profundidad
  - Recorrido en anchura
  - Recorrido de ordenación topológica
- Caminos de coste mínimo
  - Caminos mínimos con origen fijo
  - Caminos mínimos entre todo par de vértices

7.1 Modelo matemático y especificación

- Un grafo está compuesto por un conjunto de *vértices* (o *nodos*) y un conjunto de *aristas* (*arcos* en los grafos dirigidos) que definen conexiones entre los vértices. A partir de esta idea básica, se definen distintos tipos de grafos:
  - Dirigidos o no dirigidos.
  - Valorados y no valorados.
  - Simples o multigrafos.

- Un grafo se puede modelar como una pareja formada por un conjunto de vértices y un conjunto de aristas:

  $G = (V, A) \quad A \subseteq V \times V$

  Otro modelo para los grafos consiste en representarlos como un aplicación:

  $g : V \times V \rightarrow \mathbf{Bool}$ para los grafos no valorados
  $g : V \times V \rightarrow W$ para los grafos valorados

- Algunos conceptos ya conocidos sobre grafos
  - Adyacencia, incidencia.
  - Grado –de entrada y de salida– de un vértice.
  - Relaciones entre el número de vértices NV y el de aristas NA.
    - Grafo completo: existe una arista entre todo par de vértices
      - dirigido: $na = nv \cdot (nv - 1) = nv^2 - nv$
      - no dirigido: $na = (nv^2 - nv) / 2$
  - Un *camino* es una sucesión de vértices $v_0, v_1, \ldots, v_n$ tal que para $1 \leq i \leq n$ (v_{i-1}, v_i) es una arista. La longitud del camino es n.
    - Camino abierto: $v_n \neq v_0$
    - Circuito, camino cerrado o circular: $v_n = v_0$
    - Camino simple: no repite vértices (excepto quizá $v_0 = v_n$)
    - Camino elemental: no repite arcos
    - Ciclo: camino circular simple
  - Grafo conexo: existe un camino entre cada 2 vértices

Especificación algebraica

- Los detalles de la especificación cambian según la clase de grafos a especificar. En el caso más general de los grafos valorados, necesitamos especificar los grafos con dos parámetros: el tipo de los vértices y el tipo de los valores de los arcos.

- A los vértices les exigimos que pertenezcan a la clase de los tipos discretos, que sirve como generalización de los tipos que se pueden utilizar como índices de los vectores.

clase DIS
hereda
ORD
operaciones
card: → Nat
ord: Elem → Nat
elem: Nat – → Elem
prim, ult: → Elem
suc, pred: Elem – → Elem
axiomas
∀ x, y : Elem : ∀ i : Nat :
card ≥ 1
1 ≤ ord(x) ≤ card
def elem(i) **si** 1 ≤ i ≤ card
ord(elem(i)) $=^d$ i
elem(ord(x)) = x
prim = elem(1)
ult = elem(card)
def suc(x) **si** x /= ult
suc(x) $=^d$ elem(ord(x) – 1)
x == y = ord(x) == ord(y)
x ≤ y = ord(x) ≤ ord(y)
fclase

La razón de exigir esta condición a los vértices de los grafos está en que algunos algoritmos sobre grafos utilizan vectores indexados por vértices.

- En cuanto a las etiquetas de los arcos, les exigimos que pertenezcan a un tipo ordenado, que exporte una operación de suma, y que incluya el valor *NoDef*, que renombramos a ∞, y el valor 0. La razón de estas restricciones está, otra vez, en los algoritmos que queremos implementar sobre los grafos

clase VAL-ORD
hereda
EQ–ND, ORD
renombra
Elem **a** Valor
Nodef **a** ∞
operaciones
0: → Valor
(+): (Valor, Valor) → Valor
axiomas
∀ x, y, z : Valor :
0 ≤ x
x ≤ ∞
x + y = y + x
(x + y) + z = x + (y + z)
x + 0 = 0
x + ∞ = ∞
fclase

- En la especificación de los grafos incluimos las siguientes operaciones:
  - *Vacío* construye un grafo vacío
  - *PonArista* añade una arista, indicando los vértices que une, y su coste
  - *quitaArista* quita la arista que une a dos vértices dados
  - *costeArista* obtiene el coste de la arista que une a dos vértices dados
  - *hayArista?* consulta si existe una arista que une a dos vértices dados
  - *sucesores* obtiene los vértices sucesores de uno dado
  - *predecesores* obtiene los vértices predecesores de uno dado

  No prohibimos que haya *bucles*, con origen y destino en el mismo vértice. Especificamos *PonArista* de forma que no pueda haber dos aristas entre los mismos vértices; en la especificación de los multigrafos habría que suprimir esta restricción.

tad WDGRAFO[V :: DIS, W :: VAL-ORD]

renombra
V.Elem **a** Vértice

usa
BOOL, SEC[PAREJA[V, W]]

usa privadamente
CJTO[PAREJA[V, W]]

tipo
DGrafo[Vértice, Valor]

operaciones

```
Vacío: → DGrafo[Vértice, Valor]                                   /* gen */
PonArista: (DGrafo[Vértice, Valor], Vértice, Vértice, Valor)
            → DGrafo[Vértice, Valor]                              /* gen */
quitaArista: (DGrafo[Vértice, Valor], Vértice, Vértice)
            → DGrafo[Vértice, Valor]                              /* mod */
costeArista: (DGrafo[Vértice, Valor], Vértice, Vértice) → Valor   /* obs */
hayArista?: (DGrafo[Vértice, Valor], Vértice, Vértice) → Bool     /* obs */
sucesores: (DGrafo[Vértice, Valor], Vértice)
            → Sec[Pareja[Vértice, Valor]]                         /* obs */
predecesores: (DGrafo[Vértice, Valor], Vértice)
            → Sec[Pareja[Vértice, Valor]]                         /* obs */
```

operaciones privadas

cjtoSuc: (DGrafo[Vértice, Valor], Vértice)
→ Cjto[Pareja[Vértice, Valor]] /* obs */

cjtoPred: (DGrafo[Vértice, Valor], Vértice)
→ Cjto[Pareja[Vértice, Valor]] /* obs */

enumera: Cjto[Pareja[Vértice, Valor]]
→ Sec[Pareja[Vértice, Valor]] /* obs */

insertaOrd: (Pareja[Vértice, Valor], Sec[Pareja[Vértice, Valor]]
→ Sec[Pareja[Vértice, Valor]] /* obs */

ecuaciones

∀ g : DGrafo[Vértice, Valor] : ∀ u, u_1, u_2, v, v_1, v_2 : Vértice :
∀ c, c_1, c_2 : Valor : ∀ ps : Sec[Pareja[Vértice, Valor]] :
∀ xs : Cjto[Pareja[Vértice, Valor]] :

PonArista(PonArista(g, u_1, v_1, c_1), u_2, v_2, c_2) = PonArista(g, u_2, v_2, c_2)
si u_1 == u_2 AND v_1 == v_2

PonArista(PonArista(g, u_1, v_1, c_1), u_2, v_2, c_2)
= PonArista(PonArista(g, u_2, v_2, c_2), u_1, v_1, c_1)
si NOT (u_1 == u_2 AND v_1 == v_2)

quitaArista(Vacío, u, v) = Vacío

quitaArista(PonArista(g, u_1, v_1, c), u_2, v_2) = quitaArista(g, u_2, v_2)
si u_1 == u_2 AND v_1 == v_2

quitaArista(PonArista(g, u_1, v_1, c), u_2, v_2)
= PonArista(quitaArista(g, u_2, v_2), u_1, v_1, c)
si NOT (u_1 == u_2 AND v_1 == v_2)

costeArista(Vacío, u, v) = ∞

costeArista(PonArista(g, u_1, v_1, c), u_2, v_2) = c
si u_1 == u_2 AND v_1 == v_2

costeArista(PonArista(g, u_1, v_1, c), u_2, v_2) = costeArista(g, u_2, v_2)
si NOT (u_1 == u_2 AND v_1 == v_2)

hayArista?(g, u, v) = costeArista(g,u,v) /= ∞

sucesores(g, u) = enumera(cjtoSuc(g, u))

predecesores(g, v) = enumera(cjtoPred(g, v))

% *enumera* devuelve una secuencia con parte izquierda vacía

enumera(CJTO.Vacío) = SEC.Crea

enumera(Pon(Par(v, c)), xs))
= insertaOrd(Par(v, c), enumera(quita(Par(v, c), xs)))

% *insertaOrd* devuelve una secuencia con parte izquierda vacía

insertaOrd(Par(v, c), ps)
= reinicia(inserta(Par(v, c), ps))
si fin?(ps)

insertaOrd(Par(v, c), ps)
= reinicia(inserta(Par(v, c), ps))
si NOT fin?(ps) AND actual(ps) = Par(v_1, c_1) AND v < v_1

insertaOrd(Par(v, c), ps)
= insertaOrd(Par(v, c), avanza(ps))
si NOT fin?(ps) AND actual(ps) = Par(v_1, c_1) AND v ≥ v_1

cjtoSuc(Vacío, u) = CJTO.Vacío

cjtoSuc(PonArista(g, u_1, v, c), u)
= Pon(Par(v, c), cjtoSuc(quitaArista(g, u, v), u))
si u == u_1

cjtoSuc(PonArista(g, u_1, v, c), u) = cjtoSuc(g, u)
si u /= u_1

cjtoPred(Vacío, v) = CJTO.Vacío

cjtoPred(PonArista(g, u, v_1, c), v)
= Pon(Par(u, c), cjtoPred(quitaArista(g, u, v), v))
si v == v_1

cjtoPred(PonArista(g, u, v_1, c), v) = cjtoPred(g, v)
si v /= v_1

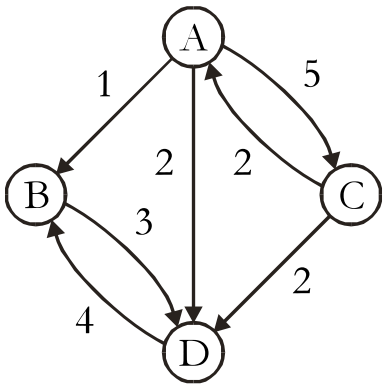
ftad

7.2 Técnicas de implementación

Matriz de adyacencia

- El grafo se representa como una matriz bidimensional indexada por vértices. En los grafos valorados, en la posición (i, j) de la matriz se almacena el peso de la arista que va del vértice i al vértice j, ∞ si no existe tal arista.

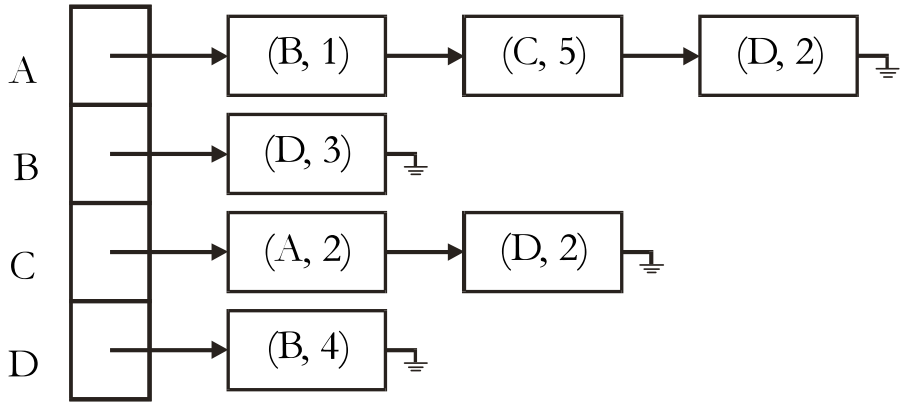
  Por ejemplo:



	A	B	C	D
A	∞	1	5	2
B	∞	∞	∞	3
C	2	∞	∞	2
D	∞	4	∞	∞

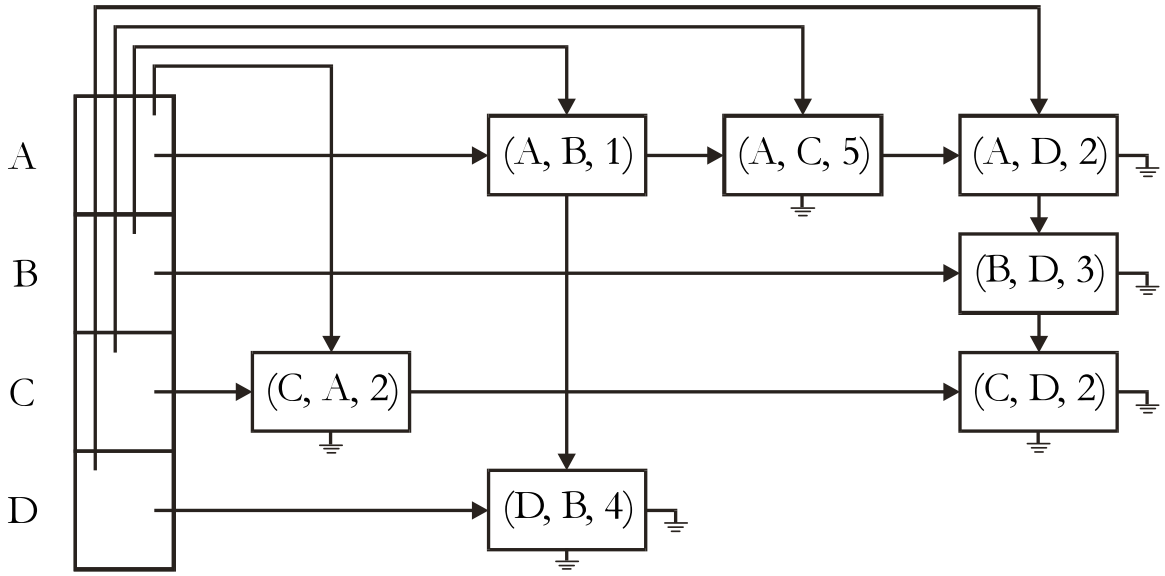
Vector de listas/secuencias de adyacencia

- Se representa como un vector indexado por vértices, donde cada posición del vector contiene la lista de aristas que parten de ese vértice –la lista de sucesores–, representadas como el vértice de destino y la etiqueta de la arista. En el grafo del ejemplo anterior:



Vector de multilistas de adyacencia

- Se representa el grafo como un vector indexado por vértices, donde cada posición del vector contiene dos listas: una con las aristas que inciden en ese vértice –lista de predecesores–, y otra con las aristas que parten de él –lista de sucesores–. La representación de cada arista se compone de el vértice de partida, el de destino y el peso. En el grafo del ejemplo anterior:



En este caso no se puede realizar una implementación modular que importe las listas/secuencias de otro módulo; hay que construir directamente un tipo representante usando registros y punteros.

Variantes

- Grafos no dirigidos.
  - La matriz de adyacencia es triangular y admite una representación optimizada.
  - En las listas de adyacencia puede ahorrarse espacio suponiendo un orden entre vértices y poniendo v en la lista de u sólo si $u < v$ –con lo que nos ahorramos representar dos veces la misma arista–.

- Grafos no valorados.
  - La matriz de adyacencia puede ser de booleanos
  - Las listas de adyacencia se reducen a listas de vértices.

- Multigrafos.
  - Las matrices de adyacencia no son una representación válida.
  - Las listas de adyacencia deben ser listas de ternas *(identificador, vértice, valor*), donde el *identificador* identifica unívocamente al arco.

Eficiencia de las operaciones

- Dados los parámetros de tamaño:
  - NV: número de vértices
  - NA: número de aristas
  - GE: máximo grado de entrada
  - GS: máximo grado de salida

Operación	Matriz	Lista	Multilista
Vacío	$O(NV^2)$ (1)	$O(NV)$	$O(NV)$
PonArista	$O(1)$	$O(GS)$ (2)	$O(GS+GE)$ (4)
quitaArista	$O(1)$	$O(GS)$ (2)	$O(GS+GE)$ (4)
costeArista	$O(1)$	$O(GS)$ (2)	$O(GS)$ (2)
hayArista	$O(1)$	$O(GS)$ (2)	$O(GS)$ (2)
sucesores	$O(NV)$ (1)	$O(GS)$ (2) (C)	$O(GS)$ (2) (C)
predecesores	$O(NV)$ (1)	$O(NV+NA)$ (3)	$O(GE)$ (5) (C)

(1) Recorrido de todos los vértices

(2) Recorrido de la lista de sucesores de un vértice. Nótese que $GS \leq NV-1$ y que por lo tanto $O(GS) \subseteq O(NV)$.

(3) Recorrido de todos los vértices, y para cada uno, recorrido de su lista de sucesores. El tiempo es $O(NV+NA)$ porque cada arista $u \rightarrow v$ se atraviesa una sola vez –en el recorrido de los sucesores de u–.

(4) Para poner o quitar la arista $u \rightarrow v$ hay que recorrer los sucesores de u y los predecesores de v –para así determinar la modificación oportuna de los enlaces en la multilista de adyacencia–. Nótese que $GE, GS \leq NV-1$ y que por lo tanto $O(GE+GS) \subseteq O(NV)$.

(5) Recorrido de los predecesores de un vértice.

Los tiempos marcados como *(C)* se reducen a $O(1)$ si no se hace una copia de la lista de sucesores/predecesores devuelta como resultado.

Implementación con multilistas de adyacencia

Tipo representante

- Clase de los vértices

```
template <class TVerticeElem, class TAristaElem>
class TVertice {
 private:
  TVerticeElem _elem;
  int _ord;
  TVertice( const TVerticeElem&, int );
 public:
  const TVerticeElem& elem() const;
  int ord() const;
  friend TGrafo<TVerticeElem, TAristaElem>;
};
```

- Clase de las aristas

```
template <class TVerticeElem, class TAristaElem>
class TArista {
 private:
  TAristaElem _elem;
  TVertice<TVerticeElem, TAristaElem> *_origen, *_destino;
  TVertice<TVerticeElem, TAristaElem>* origen() const;
  TVertice<TVerticeElem, TAristaElem>* destino() const;
  TArista( const TAristaElem&,
        TVertice<TVerticeElem, TAristaElem>*,
        TVertice<TVerticeElem, TAristaElem>* );
 public:
  const TAristaElem& elem() const;
  friend TGrafo<TVerticeElem, TAristaElem>;
};
```

- Nodos de las multilistas

```
template <class TVerticeElem, class TAristaElem>
class TNodoGrafo {
 private:
   TNodoGrafo *_suc, *_pred;
   TArista<TVerticeElem, TAristaElem>* _arista;
   TNodoGrafo* suc() const;
   TNodoGrafo* pred() const;
   TArista<TVerticeElem, TAristaElem>* arista() const;
   TNodoGrafo( TNodoGrafo*, TNodoGrafo*, TArista<TVerticeElem,TAristaElem>* );
 public:
   friend TGrafo<TVerticeElem, TAristaElem>;
};
```

- Representación de los grafos

```
   int _numVertices, _longitud;
   TVertice<TVerticeElem,TAristaElem>* *_vertices;
   TNodoGrafo<TVerticeElem,TAristaElem> *_aristas;
```

Interfaz de la clase TGrafo

```
// Excepciones que generan las operaciones de este TAD
// EVerticeInexistente

// El tipo TVerticeElem debe implementar
//    operator==
//    int TVerticeElem::num() const
// El tipo TAristaElem debe implementar
//    operator+
//    operator<=

template <class TVerticeElem, class TAristaElem>
class TGrafo {
 public:

  // Constructoras, destructora y operador de asignación
    TGrafo( int );
    // El parámetro de la constructora permite especificar el número
    // máximo de vértices que se preve almacenar.
    // Por defecto es MaxVertices

    static const int MaxVertices = 10;

    TGrafo( const TGrafo<TVerticeElem,TAristaElem>& );
    ~TGrafo( );
    TGrafo<TVerticeElem,TAristaElem>& operator=(
      const TGrafo<TVerticeElem,TAristaElem>& );

  // Operaciones de los grafos
    int insertaVertice( const TVerticeElem& );
    // Pre : true
    // Post : inserta un nuevo vértice en el grafo, asignándole un número de
    //       orden que devuelve como resultado

    void insertaArista( int, int, const TAristaElem& )
    throw ( EVerticeInexistente );
    // Pre : son ordinales válidos 0 <= i < numVertices y son distintos
    // Post : inserta una nueva arista entre los vértices identificados
    //       por los parámetros
```

```
void borraArista( int, int ) throw ( EVerticeInexistente );
// Pre : son ordinales válidos 0 <= i < numVertices
// Post : elimina la arista que conecta los vértices identificados
//       por los parámetros

// observadoras
bool hayArista( int, int ) const;
// Pre : true
// Post : devuelve true | false según si el grafo contiene o no
//       una arista conectando los vértices indicados

bool esVacio( ) const;
// Pre: true
// Post: Devuelve true | false según si el grafo está o no vacía

int numVertices( ) const;
// Pre: true
// Post: Devuelve el número de vértices del grafo

int ord( const TVerticeElem& ) const  throw ( EVerticeInexistente );
// Pre: el elemento es un vértice del grafo
// Post: Devuelve el ordinal del vértice

TSecuenciaDinamica<TVerticeElem> enumera( ) const;
// Pre: true
// Post: Devuelve una secuencia con los vértices del grafo

TSecuenciaDinamica<TVerticeElem> sucesores( int ) const;
// Pre: true
// Post: Devuelve una secuencia con los vértices sucesores de uno dado

TSecuenciaDinamica<TVerticeElem> predecesores( int ) const;
// Pre: true
// Post: Devuelve una secuencia con los vértices predecesores de uno dado

// Recorridos
TSecuenciaDinamica<TVerticeElem> enumeraProfundidad( ) const;
// Pre: true
// Post: Devuelve una secuencia con los vértices del grafo, obtenidos
//      con un recorrido en profundidad

TSecuenciaDinamica<TVerticeElem> enumeraAnchura( ) const;
// Pre: true
// Post: Devuelve una secuencia con los vértices del grafo, obtenidos
//      con un recorrido en anchura
```

```
    TSecuenciaDinamica<TVerticeElem> enumeraTopologico( ) const;
    // Pre: true
    // Post: Devuelve una secuencia con los vértices del grafo, obtenidos
    //       con un recorrido en orden topológico

  // Búsqueda de caminos mínimos
    TSecuenciaDinamica< TPareja< TAristaElem,
                        TSecuenciaDinamica<TVerticeElem> > >
    Dijkstra( int ) const  throw ( EVerticeInexistente );
    // Pre: es un ordinal válido 0 <= i < numVertices
    // Post: devuelve una secuencia con pares de vértice y la distancia
    //       mínima desde el origen a todos los vértices que le son
    //       accesibles, junto con el camino más corto que los conecta

  // Escritura
    void escribe( ostream& salida ) const;

  private:
  // Variables privadas
    int _numVertices, _longitud;
    TVertice<TVerticeElem,TAristaElem>* *_vertices;
    TNodoGrafo<TVerticeElem,TAristaElem> *_aristas;

  // Operaciones privadas
    void libera();
    void copia( const TGrafo<TVerticeElem,TAristaElem>& );
    void buscaSucesor( int,
                TNodoGrafo<TVerticeElem,TAristaElem>* &,
                TNodoGrafo<TVerticeElem,TAristaElem>* & ) const;
    void buscaPredecesor( int,
                  TNodoGrafo<TVerticeElem,TAristaElem>* &,
                  TNodoGrafo<TVerticeElem,TAristaElem>* & ) const;
    void profundidad( int,
                TCjto<int>& ,
                TSecuenciaDinamica<TVerticeElem>& ) const;
    void anchura( int,
              TCjto<int>& ,
              TSecuenciaDinamica<TVerticeElem>& ) const;
};
```

- Implementación de los vértices

```
template <class TVerticeElem, class TAristaElem>
TVertice<TVerticeElem, TAristaElem>::TVertice(
  const TVerticeElem& elem, int ord ) :
  _elem(elem), _ord(ord) { };

template <class TVerticeElem, class TAristaElem>
const TVerticeElem& TVertice<TVerticeElem, TAristaElem>::elem() const {
  return _elem;
};

template <class TVerticeElem, class TAristaElem>
int TVertice<TVerticeElem, TAristaElem>::ord() const {
  return _ord;
};
```

- Clase de las aristas

```
template <class TVerticeElem, class TAristaElem>
TArista<TVerticeElem,TAristaElem>::TArista(
  const TAristaElem& elem,
  TVertice<TVerticeElem, TAristaElem>* origen,
  TVertice<TVerticeElem, TAristaElem>* destino      ) :
  _elem(elem), _origen(origen), _destino(destino) { };

template <class TVerticeElem, class TAristaElem>
const TAristaElem& TArista<TVerticeElem,TAristaElem>::elem() const {
  return _elem;
};

template <class TVerticeElem, class TAristaElem>
TVertice<TVerticeElem,TAristaElem>*
TArista<TVerticeElem,TAristaElem>::origen() const {
  return _origen;
};

template <class TVerticeElem, class TAristaElem>
TVertice<TVerticeElem,TAristaElem>*
TArista<TVerticeElem,TAristaElem>::destino() const {
  return _destino;
};
```

- Clase de los nodos de las multilistas de adyacencia

```
template <class TVerticeElem, class TAristaElem>
TNodoGrafo<TVerticeElem,TAristaElem>::TNodoGrafo(
  TNodoGrafo* suc = 0, TNodoGrafo* pred = 0,
  TArista<TVerticeElem,TAristaElem>* arista = 0  ) :
  _suc(suc), _pred(pred), _arista(arista) { };

template <class TVerticeElem, class TAristaElem>
TNodoGrafo<TVerticeElem,TAristaElem>*
TNodoGrafo<TVerticeElem,TAristaElem>::suc() const {
  return _suc;
};

template <class TVerticeElem, class TAristaElem>
TNodoGrafo<TVerticeElem,TAristaElem>*
TNodoGrafo<TVerticeElem,TAristaElem>::pred() const {
  return _pred;
};

template <class TVerticeElem, class TAristaElem>
TArista<TVerticeElem,TAristaElem>*
TNodoGrafo<TVerticeElem,TAristaElem>::arista() const {
  return _arista;
};
```

- Constructora de los grafos

```
template <class TVerticeElem, class TAristaElem>
TGrafo<TVerticeElem,TAristaElem>::TGrafo(
  int maxVertices = MaxVertices ) :
  _numVertices(0),
  _longitud(maxVertices),
  _vertices(new TVertice<TVerticeElem,TAristaElem>*[_longitud]),
  _aristas(new TNodoGrafo<TVerticeElem,TAristaElem>[_longitud]) { };
```

- Copia, asignación y destrucción

```
template <class TVerticeElem, class TAristaElem>
TGrafo<TVerticeElem,TAristaElem>::TGrafo(
  const TGrafo<TVerticeElem,TAristaElem>& grafo ) {
   copia(grafo);
};

template <class TVerticeElem, class TAristaElem>
TGrafo<TVerticeElem,TAristaElem>::~TGrafo( ) {
  libera();
};

template <class TVerticeElem, class TAristaElem>
TGrafo<TVerticeElem,TAristaElem>&
TGrafo<TVerticeElem,TAristaElem>::operator=(
  const TGrafo<TVerticeElem,TAristaElem>& grafo ) {
  if( this != &grafo ) {
    libera();
    copia(grafo);
  }
  return *this;
};
```

- Inserción de un vértice

```
template <class TVerticeElem, class TAristaElem>
int TGrafo<TVerticeElem,TAristaElem>::insertaVertice(
 const TVerticeElem& vertice ) {
 _vertices[_numVertices] =
  new TVertice<TVerticeElem,TAristaElem>( vertice, _numVertices );
 return _numVertices++;
};
```

- Inserción de una arista

```
template <class TVerticeElem, class TAristaElem>
void TGrafo<TVerticeElem,TAristaElem>::insertaArista(
 int origen,
 int destino,
 const TAristaElem& arista )
throw ( EVerticeInexistente ) {
 if ( ( origen == destino ) ||
     ( origen < 0 ) || ( origen >= _numVertices ) ||
     ( destino < 0 ) || ( destino >= _numVertices )  )
  throw EVerticeInexistente();

 TArista<TVerticeElem,TAristaElem>* nuevaArista =
  new TArista<TVerticeElem,TAristaElem>( arista,
                                _vertices[origen],
                                _vertices[destino] );
 TNodoGrafo<TVerticeElem,TAristaElem>* nuevoNodo =
  new TNodoGrafo<TVerticeElem,TAristaElem>( _aristas[origen].suc(),
                                  _aristas[destino].pred(),
                                  nuevaArista           );
 _aristas[origen]._suc = nuevoNodo;
 _aristas[destino]._pred = nuevoNodo;
};
```

- Borrado de una arista

```
template <class TVerticeElem, class TAristaElem>
void TGrafo<TVerticeElem,TAristaElem>::borraArista(
 int origen, int destino )
throw ( EVerticeInexistente ) {
 if ( ( origen == destino ) ||
     ( origen < 0 ) || ( origen >= _numVertices ) ||
     ( destino < 0 ) || ( destino >= _numVertices )  )
  throw EVerticeInexistente();

 TNodoGrafo<TVerticeElem,TAristaElem> *act, *ant;
 act = _aristas[origen].suc();
 buscaSucesor( destino, act, ant );
 if ( act != 0 ) {
  if ( ant == 0 )
   _aristas[origen]._suc = act->suc();
  else
   ant->_suc = act->suc();
  act = _aristas[destino].pred();
  buscaPredecesor( origen, act, ant );
  if ( ant == 0 )
   _aristas[destino]._pred = act->pred();
  else
   ant->_pred = act->pred();
  delete act->arista();
  delete act;
 }
};
```

- Consulta si existe una arista conectando dos vértices dados

```
template <class TVerticeElem, class TAristaElem>
bool TGrafo<TVerticeElem,TAristaElem>::hayArista(
 int origen, int destino ) const {
 TNodoGrafo<TVerticeElem,TAristaElem> *act, *ant;
 act = _aristas[origen].suc();
 buscaSucesor( destino, act, ant );
 return act != 0;
};
```

- Consulta si el grafo está vacío

```
template <class TVerticeElem, class TAristaElem>
bool TGrafo<TVerticeElem,TAristaElem>::esVacio( ) const {
  return _numVertices == 0;
};
```

- Consulta el número de vértices

```
template <class TVerticeElem, class TAristaElem>
int TGrafo<TVerticeElem,TAristaElem>::numVertices( ) const {
  return _numVertices;
};
```

- Dado una etiqueta de un vértice obtiene el ordinal del vértice

```
template <class TVerticeElem, class TAristaElem>
int TGrafo<TVerticeElem,TAristaElem>::ord( const TVerticeElem& vertice )
const throw ( EVerticeInexistente ) {
  bool encontrado = false;
  int i = 0;

  while ( (! encontrado) && (i < _numVertices) )
    encontrado = _vertices[i++]->elem() == vertice;
  if ( ! encontrado )
    throw EVerticeInexistente();
  return --i;
};
```

- Obtiene unas secuencia con los vértices del grafo

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica<TVerticeElem>
TGrafo<TVerticeElem,TAristaElem>::enumera( ) const {
  TSecuenciaDinamica<TVerticeElem> resultado;
  for ( int i = 0; i < _numVertices; i++ )
    resultado.inserta( _vertices[i]->elem() );
  return resultado;
};
```

- Obtiene la secuencia de sucesores de un vértice dado

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica<TVerticeElem>
TGrafo<TVerticeElem,TAristaElem>::sucesores( int vertice ) const {
 TSecuenciaDinamica<TVerticeElem> resultado;
 TNodoGrafo<TVerticeElem, TAristaElem>* p = _aristas[vertice].suc();

 while( p != 0 ) {
  resultado.inserta( p->arista()->destino()->elem() );
  p = p->suc();
 }

 return resultado;
};
```

- Obtiene la secuencia de predecesores de un vértice dado

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica<TVerticeElem>
TGrafo<TVerticeElem,TAristaElem>::predecesores( int vertice ) const {
 TSecuenciaDinamica<TVerticeElem> resultado;
 TNodoGrafo<TVerticeElem, TAristaElem>* p = _aristas[vertice].pred();

 while( p != 0 ) {
  resultado.inserta( p->arista()->origen()->elem() );
  p = p->pred();
 }

 return resultado;
};
```

- Operaciones auxiliares de búsqueda

```
template <class TVerticeElem, class TAristaElem>
 void TGrafo<TVerticeElem,TAristaElem>::buscaSucesor(
  int destino,
  TNodoGrafo<TVerticeElem,TAristaElem>* & act,
  TNodoGrafo<TVerticeElem,TAristaElem>* & ant ) const {
  ant = 0;
  while ( ( act != 0 ) &&
        ( act->arista()->destino()->ord() != destino ) ) {
   ant = act;
   act = act->suc();
  }
 }

template <class TVerticeElem, class TAristaElem>
 void TGrafo<TVerticeElem,TAristaElem>::buscaPredecesor(
  int origen,
  TNodoGrafo<TVerticeElem,TAristaElem>* & act,
  TNodoGrafo<TVerticeElem,TAristaElem>* & ant ) const {
  ant = 0;
  while ( ( act != 0 ) &&
        ( act->arista()->origen()->ord() != origen ) ) {
   ant = act;
   act = act->pred();
  }
 }
```

7.3 Recorridos de grafos

- En analogía con los árboles, los grafos admiten recorridos en profundidad y en anchura. En general, los recorridos no dependen de los valores de los arcos, por lo que en este apartado nos limitaremos a grafos no valorados.

- El TAD grafo no impone un orden determinado a los sucesores (o predecesores) de un vértice. Por lo tanto, no es posible especificar de manera unívoca una lista de vértices como resultado de un recorrido.

- Los grafos dirigidos acíclicos admiten un tercer tipo de recorrido, denominado *recorrido de ordenación topológica*.

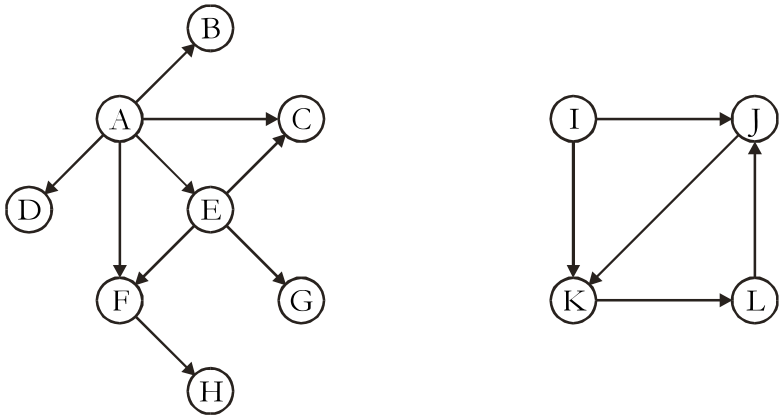
- Realizaremos implementaciones modulares de las operaciones, sin acceder a la representación interna de los grafos.

7.3.1 Recorrido en profundidad

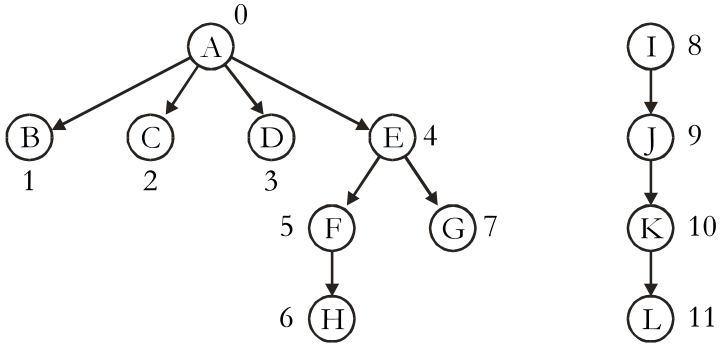
- Se puede considerar como una generalización del recorrido en preorden de un árbol. La idea es:
  - Visitar el vértice inicial
  - Si es posible, avanzar a un sucesor aún no visitado
  - En otro caso, retroceder al vértice visitado anteriormente, e intentar continuar el recorrido desde éste.

  Una vez visitados todos los descendientes del vértice inicial, si quedan vértices no visitados se inicia un recorrido de otra componente del grafo.

  Por ejemplo, el recorrido del grafo:



  Representado como un bosque:



  El algoritmo recursivo de recorrido en profundidad

  - Devuelve una secuencia en la cual aparece cada vértice del grafo una sola vez.
  - Usa un conjunto de vértices para llevar cuenta de los visitados.
  - Usa un procedimiento auxiliar privado encargado del recorrido de una sola componente.

- La implementación

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica<TVerticeElem>
TGrafo<TVerticeElem,TAristaElem>::enumeraProfundidad( ) const {
 TSecuenciaDinamica<TVerticeElem> resultado;
 TCjto<int> visitados;

 for ( int i = 0; i < _numVertices; i++ )
  if ( ! visitados.esta( i ) )
   profundidad( i, visitados, resultado );

 return resultado;
};
```

El procedimiento auxiliar que recorre una componente:

```
template <class TVerticeElem, class TAristaElem>
void
TGrafo<TVerticeElem,TAristaElem>::profundidad(
 int vertice,
 TCjto<int>& visitados,
 TSecuenciaDinamica<TVerticeElem>& resultado  ) const {

 TNodoGrafo<TVerticeElem, TAristaElem>* act = _aristas[vertice].suc();

 visitados.inserta( vertice );
 resultado.inserta( _vertices[vertice]->elem() );
 while ( act != 0 ) {
  if ( ! visitados.esta( act->arista()->destino()->ord() ) )
   profundidad( act->arista()->destino()->ord(),
            visitados, resultado );
  act = act->suc();
 }
};
```

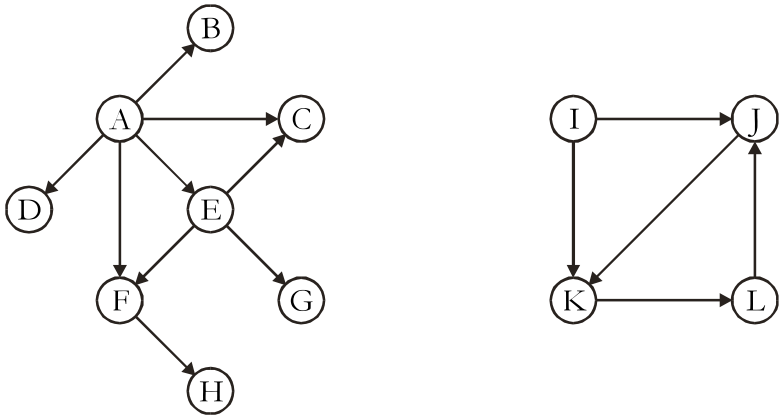
- La complejidad de la operación depende de la representación elegida para los grafos:
  - Si se usan listas o multilistas de adyacencia, el tiempo es $O(NV+NA)$, ya que cada vértice se visita una sola vez, pero se exploran todas las aristas que salen de él.
  - Si se usan matrices de adyacencia el tiempo es $O(NV^2)$, ya que cada vértice se visita una sola vez, pero el cálculo de sus sucesores requiere tiempo $O(NV)$.

  Para conseguir estos tiempos es necesario que las operaciones de CJTO sean $O(1)$, lo que se puede conseguir con una implementación basada en tablas dispersas (un vector de booleanos).

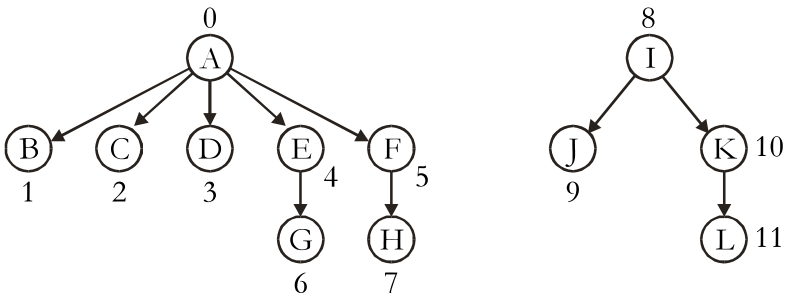
7.3.2 Recorrido en anchura

- El recorrido en anchura, o por niveles, generaliza el recorrido de árboles con igual denominación. La idea es:
  - Visitar el vértice inicial.
  - Si el último vértice visitado tiene sucesores aún no visitados, realizar sucesivamente un recorrido desde cada uno de estos.
  - En otro caso, continuar con un recorrido iniciado en cualquier vértice no visitado aún.

  Por ejemplo, para el mismo grafo del ejemplo anterior:



representado como un bosque, resultado del recorrido por niveles



- La implementación es similar a la del recorrido en profundidad, con ayuda de un procedimiento auxiliar que recorre una componente del grafo. La diferencia está en que en lugar de utilizar un procedimiento recursivo, lo hacemos iterativo con ayuda de una cola.

- La implementación

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica<TVerticeElem>
TGrafo<TVerticeElem,TAristaElem>::enumeraAnchura( ) const {
 TSecuenciaDinamica<TVerticeElem> resultado;
 TCjto<int> visitados;

 for ( int i = 0; i < _numVertices; i++ )
  if ( ! visitados.esta( i ) )
   anchura( i, visitados, resultado );

 return resultado;
};
```

El procedimiento auxiliar que recorre por niveles una componente:

```
template <class TVerticeElem, class TAristaElem>
void
TGrafo<TVerticeElem,TAristaElem>::anchura(
 int vertice,
 TCjto<int>& visitados,
 TSecuenciaDinamica<TVerticeElem>& resultado  ) const {

 TNodoGrafo<TVerticeElem, TAristaElem>* act;
 TColaDinamica<int> pendientes;
 int actual;
 visitados.inserta( vertice );
 pendientes.ponDetras( vertice );
 while ( ! pendientes.esVacio() ) {
  actual = pendientes.primero();
  pendientes.quitaPrim();
  resultado.inserta( _vertices[actual]->elem() );
  act = _aristas[actual].suc();
  while ( act != 0 ) {
   if ( ! visitados.esta( act->arista()->destino()->ord() ) ) {
    visitados.inserta( act->arista()->destino()->ord() );
    pendientes.ponDetras( act->arista()->destino()->ord() );
   }
   act = act->suc();
  }
 }
};
```

El análisis de la complejidad es el mismo que hemos hecho para el recorrido en profundidad.

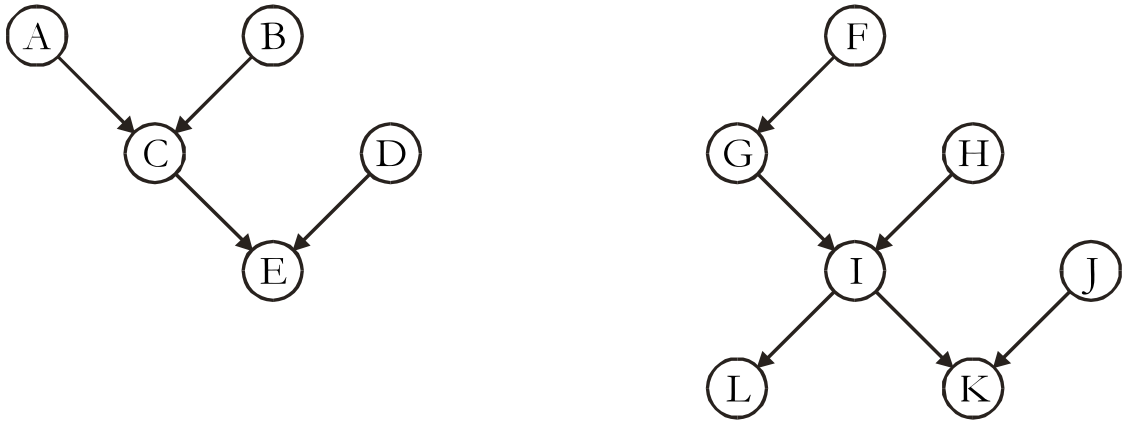
7.3.3 Recorrido de ordenación topológica

- Este tipo de recorridos sólo es aplicable a grafos dirigidos acíclicos.
  Dado un grafo dirigido acíclico *G*, la relación entre vértices definida como

  $u \prec_G v \Leftrightarrow_{\text{def}}$ existe un camino de *u* a *v* en *G*

  es un orden parcial. Se llama *recorrido de ordenación topológica* de *G* a cualquier recorrido de *G* que visite cada vértice *v* solamente después de haber visitado todos los vértices de *u* tales que $u \prec_G v$. En general, son posibles varios recorridos de ordenación topológica para un mismo grafo *G*.

  Por ejemplo, algunos recorridos en ordenación topológica del grafo:



  xs : [A, B, C, D, E, F, G, H, I, J, K, L]
  xs : [A, B, D, F, H, J, C, E, G, I, L, K] …

- La idea básica del algoritmo es reiterar la elección de un vértice aún no visitado y tal que todos sus predecesores hayan sido ya visitados. El problema es que el algoritmo resultante de seguir directamente esta idea es poco eficiente.
  Una forma de lograr un algoritmo más eficiente es:
  - Mantener un vector *P* indexado por vértices, tal que *P*[*v*] es el número de predecesores de *v* aún no visitados.
  - Mantener los vértices *v* tal que *P*[*v*] = 0 en un conjunto *M*.
  - Organizar un bucle que en cada vuelta:
    - añade un vértice de *M* al recorrido
    - actualiza *P* y *M*

- La implementación del algoritmo

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica<TVerticeElem>
TGrafo<TVerticeElem,TAristaElem>::enumeraTopologico( ) const {
 TSecuenciaDinamica<TVerticeElem> resultado;
 int* numPred = new int[_numVertices];
 TSecuenciaDinamica<int> visitables;
 TNodoGrafo<TVerticeElem, TAristaElem> *act;
 int actual, destino;
 for ( int i = 0; i < _numVertices; i++ ) {
  numPred[i] = 0;
  act = _aristas[i].pred();
  while ( act != 0 ) {
   numPred[i]++;
   act = act->pred();
  }
  if ( numPred[i] == 0 )
   visitables.inserta(i);
 }
 while ( ! visitables.esVacio() ) {
  visitables.reinicia();
  actual = visitables.actual();
  visitables.borra();
  resultado.inserta( _vertices[actual]->elem() );
  act = _aristas[actual].suc();
  while ( act != 0 ) {
   destino = act->arista()->destino()->ord();
   numPred[ destino ]--;
   if ( numPred[ destino ] == 0 )
    visitables.inserta( destino );
   act = act->suc();
  }
 }
 return resultado;
};
```

- En cuanto a la complejidad, el algoritmo opera en tiempo:
  - $O(NV^2)$ si el grafo está representado como matriz de adyacencia.
  - $O(NV+NA)$ si el grafo está representado conlistas de adyacencia.

- Este algoritmo se puede modificar para detectar si el grafo es acíclico o no, en lugar de exigir aciclicidad en la precondición. Si el conjunto *M* es queda vacío antes de haber visitado NV vértices, el grafo no es acíclico.

7.4 Caminos de coste mínimo

- En este apartado vamos a estudiar algoritmos que calculan caminos de coste mínimo en grafos dirigidos valorados. El coste de un camino se calcula como la suma de los valores de sus arcos. Se presupone por tanto que existe una operación de suma entre valores. En las aplicaciones prácticas, los valores suelen ser números no negativos. Sin embargo, la corrección de los algoritmos que vamos a estudiar sólo exige que el tipo de los valores satisfaga los requisitos expresados en la especificación de la clase de tipos VAL-ORD que presentamos al principio del tema.

7.4.1 Caminos mínimos con origen fijo

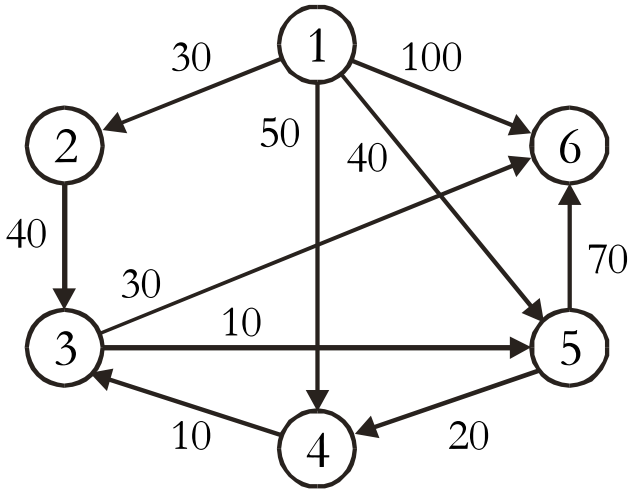
- Dado un vértice *u* de un grafo dirigido valorado *g*, nos interesa calcular todos los caminos mínimos con origen *u* y sus correspondientes costes.

- Para resolver este problema vamos a utilizar el *algoritmo de Dijkstra* (1959). El algoritmo mantiene un conjunto *M* de vértices *v* para los cuales ya se ha encontrado el camino mínimo desde *u*. La idea del algoritmo:
  - Se inicializa $M := \{u\}$. Para cada vértice *v* diferente de *u*, se inicializa un coste estimado $C(v) := costeArista(G, u, v)$.
  - Se entra en un bucle. En cada vuelta se elige un vértice *w* que no esté en *M* y cuyo coste estimado sea mínimo. Se añade *w* a *M*, y se actualizan los costes estimados de los restantes vértices *v* que no estén en *M*, haciendo $C(v) := min(C(v), C(w) + costeArista(G, w, v))$.

  Al terminar, se han encontrado caminos de coste mínimo *C(v)* desde *u* hasta los restantes vértices *v*. Esta idea se puede refinar:

  - Si en algún momento llega a cumplirse que para todos los vértices *v* que no están en *M* $C(v) = \infty$, el algoritmo puede terminar.
  - Además de calcular *C*, calculamos otro vector *p*, indexado por los ordinales de los vértices, que representa los caminos mínimos desde *u* a los restantes vértices, según el siguiente criterio:
    - $p(ord(v)) = 0$ si *v* no es accesible desde *u*.
    - $p(ord(v)) = ord(w)$ si *v* es accesible desde *u* y *w* es el predecesor inmediato de *v* en el camino mínimo de *u* a *v*.

- Como ejemplo del funcionamiento del algoritmo, vamos a ejecutarlo para el grafo dirigido de la figura siguiente, tomando como vértice inicial *u = 1*.



It.	M	w	c/p(1)	c/p(2)	c/p(3)	c/p(4)	c/p(5)	c/p(6)
0	{1}	–	0/1	30/1	∞/0	50/1	40/1	100/1
1	{1,2}	2	0/1	30/1	70/2	50/1	40/1	100/1
2	{1,2,5}	5	0/1	30/1	70/2	50/1	40/1	100/1
3	{1,2,5,4}	4	0/1	30/1	60/4	50/1	40/1	100/1
4	{1,2,5,4,3}	3	0/1	30/1	60/4	50/1	40/1	90/3
5	{1,2,5,4,3,6}	6	0/1	30/1	60/4	50/1	40/1	90/3

De aquí se deduce, por ejemplo, que un camino mínimo de 1 a 6, con coste *c(v) = 90*, es [1, 4, 3, 6].

Implementación del algoritmo

- Necesitamos una clase auxiliar para representar las distancias entre vértices

```
template <class TAristaElem>
class TDistancia {
 public:

  enum TTipo { Cero, Valor, Infinito };

  TDistancia ( TTipo tipo = Infinito ) :
   _tipo(tipo), _valor(0) {
  };

  TDistancia ( const TAristaElem& valor ) :
   _tipo( Valor ) {
   _valor = new TAristaElem( valor );
  };

  TDistancia ( const TDistancia& distancia ) { copia( distancia ); }

  ~TDistancia ( ) { libera( ); }

  TDistancia& operator=( const TDistancia& distancia ) {
   if ( &distancia != this ) {
     libera();
     copia( distancia );
   }
   return *this;
  }

  bool operator== ( const TDistancia& distancia ) const {
   return ( esCero() && distancia.esCero() ) ||
          ( esInfinito() && distancia.esInfinito() ) ||
          ( ( esValor() ) && ( distancia.esValor() ) &&
            ( *_valor == distancia.valor() )          );
  }

  bool operator<= ( const TDistancia& distancia ) const {
   return esCero() || distancia.esInfinito() ||
          ( ( esValor() ) && ( distancia.esValor() ) &&
            ( *_valor <= distancia.valor() )          );
  }
```

```
  const TDistancia& operator+= ( const TDistancia& distancia ) const {
   if ( esCero() || ( esValor() && distancia.esInfinito() ) ) {
    libera();
    copia( distancia );
   }
   else if ( esValor() && distancia.esValor() )
    *_valor += distancia.valor();
   return *this;
  }

  TDistancia& operator+ ( const TDistancia& distancia ) const {
   TDistancia<TAristaElem> *resultado =
    new TDistancia<TAristaElem>( *this );
   *resultado += distancia;
   return *resultado;
  }

  const TAristaElem& valor( ) const { return *_valor; }

  bool esCero () const { return _tipo == Cero; }
  bool esInfinito () const { return _tipo == Infinito; }
  bool esValor () const { return _tipo == Valor; }

 private:
  TAristaElem * _valor;
  TTipo _tipo;

  void libera() { if ( esValor() ) delete _valor; }

  void copia( const TDistancia& distancia ) {
   _tipo = distancia._tipo;
   if ( distancia.esValor() )
    _valor = new TAristaElem( distancia.valor() );
  }
};
```

- Finalmente el algoritmo de Dijkstra

```
template <class TVerticeElem, class TAristaElem>
TSecuenciaDinamica< TPareja< TAristaElem,
                    TSecuenciaDinamica<TVerticeElem> > >
TGrafo<TVerticeElem,TAristaElem>::Dijkstra( int vertice ) const
throw ( EVerticeInexistente )
{
 if ( ( vertice < 0 ) || ( vertice >= _numVertices ) )
  throw EVerticeInexistente();
 TSecuenciaDinamica< TPareja< TAristaElem,
                     TSecuenciaDinamica<TVerticeElem> > >
  resultado;
 TCjto<int> calculados;
 int *caminos = new int[_numVertices];
 // el array de distancias se inicializa con todas las posiciones a
 // Infinito
 TDistancia<TAristaElem> *distancias =
  new TDistancia<TAristaElem>[_numVertices];
 TNodoGrafo<TVerticeElem,TAristaElem> *act, *ant;

 calculados.inserta(vertice);
 for ( int i = 0; i < _numVertices; i++ ) {
  if ( i == vertice ) {
   caminos[i] = i;
   distancias[i] =
    TDistancia<TAristaElem>(TDistancia<TAristaElem>::Cero);
  }
  else {
   act = _aristas[vertice].suc();
   buscaSucesor( i, act, ant );
   if ( act != 0 ) {
    caminos[i] = vertice;
    distancias[i] = TDistancia<TAristaElem>( act->arista()->elem() );
   }
   else
    caminos[i] = -1;
  }
 }
```

```
bool fin = false;
int n = 1, masCercano;
TDistancia<TAristaElem> minDist, nuevaDist;
while ( ( n < _numVertices ) && ! fin ) {
 minDist = TDistancia<TAristaElem>(TDistancia<TAristaElem>::Infinito);
 for ( int i = 0; i < _numVertices; i++ ) {
  if ( ! calculados.esta(i) )
   if ( distancias[i] <= minDist ) {
    minDist = distancias[i];
    masCercano = i;
   }
 }
 if ( minDist.esInfinito() )
  fin = true;
 else {
  calculados.inserta( masCercano );
  n++;
  for ( int i = 0; i < _numVertices; i++ )
   if ( ! calculados.esta(i) ) {
    act = _aristas[masCercano].suc();
    buscaSucesor( i, act, ant );
    if ( act != 0 ) {
     nuevaDist = TDistancia<TAristaElem> ( act->arista()->elem() )
            + minDist;
     if ( ! ( distancias[i] <= nuevaDist ) ) {
      caminos[i] = masCercano;
      distancias[i] = nuevaDist;
     }
    }
   }
 }
}
```

```
  TSecuenciaDinamica<TVerticeElem> camino;
  int actual;
  for ( int i = 0; i < _numVertices; i ++ ) {
   if ( ( caminos[i] != -1 ) && ( i != vertice ) ) {
    camino = TSecuenciaDinamica<TVerticeElem>();
    camino.inserta( _vertices[i]->elem() );
    actual = i;
    do {
     actual = caminos[actual];
     camino.reinicia();
     camino.inserta( _vertices[actual]->elem() );
    } while ( actual != vertice );
    resultado.inserta(
     TPareja< TAristaElem, TSecuenciaDinamica< TVerticeElem > > (
      distancias[i].valor(), camino                    ) );
   }
  }
  return resultado;
};
```

Complejidad del algoritmo de Dijkstra

- La realización anterior está pensada para una representación de los grafos con matrices de adyacencia. El coste de las diferentes etapas:
  - Incialización de *c* y *p*: $O(NV)$. NV iteraciones donde cada iteración sólo involucra operaciones con coste constante. Esto es cierto si los grafos se implementan con matrices de adyacencia, donde efectivamente *costeArista* es $O(1)$.
  - El bucle principal se compone de $O(NV)$ iteraciones, donde:
    - La selección de *w* se realiza con un bucle de coste $O(NV)$, suponiendo que utilizamos una implementación de los conjuntos donde *pertenece* es $O(1)$.
    - La actualización de *c* y *p* se realiza con un bucle que ejecuta $O(NV)$ iteraciones. Cada iteración tiene coste constante siempre y cuando *costeArista* tenga coste $O(1)$.

  Por lo tanto, el coste total es $O(NV^2)$.

- Si se utilizan grafos representados con listas de adyacencia, es necesario realizar algunos cambios en el algoritmo para obtener esa complejidad, debido a que *costeArista* pasa a ser $O(NV)$:
  - Inicializaciones, para obtener tiempo $O(NV)$
    - *c* se inicializa con ∞ y *p* se inicializa con 0, excepto *c(u) := 0* y *p(ord(u)) := ord(u)*. $O(NV)$
    - Para cada pareja *(c, v)* perteneciente a *sucesores(g, u)* se hace: *c(v) := c; p(v) := u*. $O(GS) \subseteq O(NV)$; $O(GS) \subseteq O(NA)$.
  - Bucle principal, para obtener tiempo $O(NV^2)$
    - Se cambia el bucle interno que actualiza *c* y *p*, escribiéndolo como bucle que recorre la secuencia de sucesores de *w*. $O(GS) \subseteq O(NV)$

- En cuanto al espacio, se requiere espacio adicional $O(NV)$ para el conjunto de vértices *m*, además del espacio ocupado por el propio grafo y los resultados.

7.4.2 Caminos mínimos entre todo par de vértices

- El problema es, dado un grafo dirigido valorado *g*, se quieren calcular todos los caminos mínimos entre todas las parejas de vértices de *g*, junto con sus costes.

  Aplicando reiteradamente el algoritmo de Dijkstra, se obtiene una solución de complejidad $O(NV^3)$.

- La otra posibilidad es utilizar el *algoritmo de Floyd* (1962). Este algoritmo, aunque con el mismo coste, tiene la ventaja de ser más elegante y compacto. Además sólo necesita espacio adicional $O(1)$, mientras que el algoritmo de Dijkstra necesita espacio auxiliar $O(NV)$.

- La idea básica del algoritmo consiste en mejorar la estimación *c(u, v)* del coste de un camino mínimo de *u* a *v* mediante la asignación:

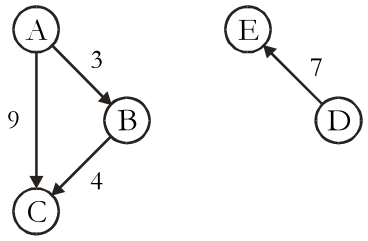
  ```
  c(v, w) := min( c(v, w), c(v, u) + c(u, w) )
  ```

  Diremos que el vértice *u* actúa como *pivote* en esta actualización de *c(v, w)*. El algoritmo comienza con una inicialización natural de *c*, y a continuación reitera la actualización de *c(v, w)* para todas las parejas de vértices *(v, w)* y con todos los pivotes *u*.

  Al igual que en el algoritmo de Dijkstra, construimos un resultado adicional que representa los caminos mínimos entre cada par de vértices. Este resultado viene representado por un vector bidimensional *s* indexado por parejas de ordinales de los vértices, según el siguiente criterio:

  – *s(ord(v), ord(w)) = 0* si *v* no hay caminos de *v* a *w*.
  – *s(ord(v), ord(w)) = ord(u)* si *w* es accesible desde *v* y *u* es el sucesor inmediato de *v* en el camino mínimo de *v* a *w* calculado por el algoritmo.

- Apliquemos el algoritmo al siguiente grafo:



0: Estado inicial

Matriz de costes *c*

	A	**B**	**C**	**D**	**E**
A	0	3	9	∞	∞
B	∞	0	4	∞	∞
C	∞	∞	0	∞	∞
D	∞	∞	∞	0	7
E	∞	∞	∞	∞	0

Matriz de sucesores *s*

	1	**2**	**3**	**4**	**5**
1	1	2	3	0	0
2	0	2	3	0	0
3	0	0	3	0	0
4	0	0	0	4	5
5	0	0	0	0	5

1: Después de actualizar *c* y *s* usando *A* como vértice pivote.

Ningún arco entra en *A* por lo tanto usar *A* como pivote no mejora nada. *c* y *s* quedan inalterados.

2: Después de actualizar *c* y *s* usando *B* como vértice pivote.

Esto permite mejorar el coste del camino entre *A* y *C*. *c(A, C)* y *s(1, 3)* se modifican, las demás posiciones no se modifican.

Matriz de costes *c*

	A	**B**	**C**	**D**	**E**
A	0	3	7	∞	∞
B	∞	0	4	∞	∞
C	∞	∞	0	∞	∞
D	∞	∞	∞	0	7
E	∞	∞	∞	∞	0

Matriz de sucesores *s*

	1	**2**	**3**	**4**	**5**
1	1	2	2	0	0
2	0	2	3	0	0
3	0	0	3	0	0
4	0	0	0	4	5
5	0	0	0	0	5

3, 4, 5: Actualizaciones de *c* y *s* usando como vértices pivote *C, D* y *E*.

No mejoran nada, *c* y *s* quedan como en la figura anterior.