Capítulo 6 Relaciones binarias y grafos

En este capítulo nos centraremos en el modelo de las relaciones binarias[1] entre elementos de dos dominios A y B, donde un elemento de A está relacionado con un elemento de B si se cumple el enunciado de la relación. Generalmente, un elemento de A estará relacionado con varios de B y viceversa; por ello, en algunos textos informáticos las relaciones binarias se denominan relaciones m:n como abreviatura de: m elementos de A se relacionan con n de B y viceversa.

Un ejemplo de aplicación de las relaciones binarias podría ser la gestión de la matriculación de alumnos en una universidad. La estructura necesaria se puede considerar como una relación entre dos conjuntos de elementos: los alumnos y las asignaturas, por la que cada alumno está relacionado con todas las asignaturas que cursa y cada asignatura con todos los alumnos que se han matriculado de la misma.

	IP	IC	Alg	Fis	PM	ILo	EDA
Abad	x	x	x	x			
Abadía		x		x	x	x	
Aguilar					x	x	x
...							

Fig. 6.1: representación de la relación alumnos-asignaturas (la cruz significa que el alumno cursa la asignatura).

Dado el contexto de uso esperado, las operaciones sobre el tipo que parecen adecuadas son: matricular un alumno de una asignatura, desmatricularlo (cuando la ha aprobado), averiguar si un alumno cursa una asignatura concreta, y listar las asignaturas que cursa un alumno y los alumnos que cursan una asignatura. Extrapolando estas operaciones a un marco general, podemos formular una especificación para las relaciones y deducir implementaciones eficientes. Concretamente, en la primera sección estudiaremos la

[1] Si bien nos podríamos plantear el estudio de relaciones sobre un número arbitrario de dominios, el caso habitual es el binario y por eso nos limitamos a éste.

especificación y la implementación de las relaciones binarias en general, mientras que en el resto del capítulo nos centraremos en el caso particular de relaciones binarias sobre un único dominio, denominadas grafos. Sobre los grafos se definen varios algoritmos de gran interés, cuya resolución será ampliamente comentada y en cuya implementación intervendrán diversos TAD ya conocidos, como las colas prioritarias y las relaciones de equivalencia.

6.1 Relaciones binarias

Se quiere especificar el TAD de las relaciones binarias R_{AxB} definidas sobre dos dominios de datos A y B, tal que todo valor R del TAD, $R \in R_{AxB}$, es un conjunto de pares ordenados <a, b>, $a \in A$ y $b \in B$. Dados $R \in R_{AxB}$, $a \in A$ y $b \in B$, la signatura del tipo es:

- Crear la relación vacía: crea, devuelve Ø.
- Añadir un par a la relación: inserta(R, a, b), devuelve $R \cup \{<a, b>\}$.
- Borrar un par de la relación: borra(R, a, b), devuelve $R - \{<a, b>\}$.
- Mirar si un par de elementos están relacionados: existe?(R, a, b), averigua si $<a, b> \in R$.
- Dos operaciones para determinar el conjunto recorrible[2] de elementos (v. fig. 4.27) que están relacionados con uno dado: fila(R, a), devuelve el conjunto $s \in P(B)$, definido como $b \in s \Leftrightarrow <a, b> \in R$, y columna(R, b), que se define simétricamente.

Alternativamente, se podría considerar la relación como una función de pares de elementos en el álgebra booleana B de valores cierto y falso, $f : A \times B \rightarrow B$, en la que f(a, b) vale cierto si a y b están relacionados. Este modelo se adoptará en el caso de relaciones binarias valoradas, que se introduce más adelante.

En la figura 6.2 se presenta una especificación para las relaciones. Notemos que aparecen diferentes parámetros formales. Por un lado, los géneros sobre los cuales se define la relación así como sus operaciones de igualdad, imprescindibles en la especificación. Por otro lado, y para establecer un criterio de ordenación de los elementos al insertarlos en el conjunto recorrible, se requiere que los géneros presenten una operación de comparación adicional, tal como establece el universo ELEM_<_= presentado en la fig. 4.28. Se ha optado por definir los conjuntos recorribles como ordenados porque éste será el caso habitual y así no surgen problemas de consistencia al implementar el tipo (concretamente, se evita que las implementaciones secuenciales inserten los elementos en los conjuntos en un orden diferente que en la especificación).

Notemos que, al haber dos operaciones que devuelven dos conjuntos de elementos de tipos diferentes, es necesario efectuar dos instancias de los conjuntos genéricos para

[2] El conjunto ha de ser recorrible para que disponga de operaciones cómodas y eficientes de recorrido al implementar algoritmos posteriores. No se usan listas con punto de interés para evitar repetidos.

determinar su signatura. Los tipos resultantes son implícitamente exportados y podrán ser empleados por los módulos usuarios de las relaciones para declarar variables o parámetros.

Por lo que respecta a las ecuaciones, la relaciones establecidas sobre inserta confirman que la relación representa un conjunto de pares. Notemos también que, al borrar, hay que sacar todas las posibles apariciones del par y que si el par no existe la relación queda igual. Por último, cabe destacar que la inserción reiterada de elementos en obtener una fila o columna no afecta el resultado por las ecuaciones propias de los conjuntos recorribles.

universo RELACIÓN (A, B son ELEM_<_=) es
  usa BOOL
  instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_=) donde
      C.elem es A.elem, C.< es A.<, C.= es A.=
    renombra cjt por cjt_a
  instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_=) donde
      C.elem es B.elem, C.< es B.<, C.= es B.=
    renombra cjt por cjt_b
  tipo relación
  ops crea: $\rightarrow$ relación
    inserta, borra: relación A.elem B.elem $\rightarrow$ relación
    existe?: relación A.elem B.elem $\rightarrow$ bool
    fila: relación A.elem $\rightarrow$ cjt_b
    columna: relación B.elem $\rightarrow$ cjt_a
  ecns $\forall$R$\in$ relación; $\forall$a,a_1,$a_2\in$ A.elem; $\forall$b,b_1,$b_2\in$ B.elem
    inserta(inserta(R, a, b), a, b) = inserta(R, a, b)
    inserta(inserta(R, a_1, b_1), a_2, b_2) = inserta(inserta(R, a_2, b_2), a_1, b_1)
    borra(crea, a, b) = crea
    borra(inserta(R, a, b), a, b) = borra(R, a, b)
    $[(a_1 \neq a_2) \vee (b_1 \neq b_2))] \Rightarrow$ borra(inserta(R, a_1, b_1), a_2, b_2) = inserta(borra(R, a_2, b_2), a_1, b_1)
    existe?(crea, a, b) = falso
    existe?(inserta(R, a_1, b_1), a_2, b_2) = existe?(R, a_2, b_2) $\vee$ $((a_1 = a_2) \wedge (b_1 = b_2))$
    fila(crea, a) = CJT_ORDENADO_RECORRIBLE.crea
    fila(inserta(R, a, b), a) = CJT_ORDENADO_RECORRIBLE.añade(fila(R, a), b)
    $[a_1 \neq a_2] \Rightarrow$ fila(inserta(R, a_1, b), a_2) = fila(R, a_2)
    columna(crea, b) = CJT_ORDENADO_RECORRIBLE.crea
    columna(inserta(R, a, b), b) = CJT_ORDENADO_RECORRIBLE.añade(columna(R, b), a)
    $[b_1 \neq b_2] \Rightarrow$ columna(inserta(R, a, b_1), b_2) = columna(R, b_2)
funiverso

Fig. 6.2: especificación del tipo abstracto de datos de las relaciones.

El uso de las relaciones exige a veces, no sólo relacionar pares de elementos, sino también explicitar un valor que caracteriza este nexo, en cuyo caso diremos que la relación es valorada; en el ejemplo de la fig. 6.1, esto sucede si se quiere guardar la nota que un alumno obtiene de cada asignatura que cursa. El modelo resultante exige algunos cambios en la especificación y, posteriormente, en la implementación. Por lo que respecta al modelo, la relación se puede considerar como una función de dos variables (los dominios de la relación) sobre el codominio V de los valores, f : A x B $\rightarrow$ V, de manera que f(a, b) represente el valor de la relación. Supondremos que V presenta un valor especial $\perp$ tal que, si a y b no están relacionados, consulta(f, a, b) = $\perp$ y, así, la función f siempre es total. De lo contrario, debería añadirse una operación definida?: relación A.elem B.elem $\rightarrow$ bool para ver si un par está en el dominio de la función y considerar un error que se consulte el valor de un par indefinido.

En la figura 6.3 se presenta la signatura del TAD (la especificación queda como ejercicio para el lector); notemos que el género de los valores y el valor $\perp$ se definen en el universo ELEM_ESP (v. fig. 4.1). Las operaciones fila y columna no sólo devuelven los elementos con que se relacionan sino también el valor, por lo que los conjuntos resultantes son de pares de elementos que se forman como instancia del universo genérico PAR, en las que el concepto de igualdad tan sólo tiene en cuenta el componente elemento y no el valor.

universo RELACIÓN_VALORADA (A, B son ELEM_<_=, V es ELEM_ESP) es
  instancia PAR (P1, P2 son ELEM) donde
    P1.elem es A.elem, P2.elem es V.elem
    renombra par por a_y_valor, _.c_1 por _.a, _.c_2 por _.val
  instancia PAR (P1, P2 son ELEM) donde
    P1.elem es B.elem, P2.elem es V.elem
    renombra par por b_y_valor, _.c_1 por _.b, _.c_2 por _.val
  instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_=) donde
    C.elem es a_y_valor, C.= es (_.a) A.= (_.a), C.< es (_.a) A.< (_.a)
    renombra cjt por cjt_a_y_valor
  instancia CJT_ORDENADO_RECORRIBLE(C es ELEM_<_=) donde
    C.elem es b_y_valor, C.= es (_.b) B.= (_.b), C.< es (_.b) B.< (_.b)
    renombra cjt por cjt_b_y_valor
  tipo relación
  ops crea: $\rightarrow$ relación
    inserta: relación A.elem B.elem V.elem $\rightarrow$ relación
    borra: relación A.elem B.elem $\rightarrow$ relación
    consulta: relación A.elem B.elem $\rightarrow$ V.elem
    fila: relación A.elem $\rightarrow$ cjt_b_y_valor
    columna: relación B.elem $\rightarrow$ cjt_a_y_valor
funiverso

Fig. 6.3: signatura del tipo abstracto de datos de las relaciones valoradas.

En el resto de la sección estudiaremos la implementación del TAD de las relaciones binarias no valoradas; la extensión al caso valorado queda como ejercicio para el lector.

La representación más intuitiva del tipo de las relaciones binarias consiste en usar un vector bidimensional de booleanos, con lo que se registra si el par correspondiente a cada posición pertenece o no a la relación. Si hay muchos elementos interrelacionados esta solución puede ser satisfactoria, pero no lo será en el caso general, principalmente por razones de espacio y también por la ineficiencia de fila y columna, que quedan de orden lineal independientemente del número de elementos que en hay en ellas. Por ejemplo, si consideramos una universidad que imparta 400 asignaturas con una media de 6 asignaturas cursadas por alumno, la matriz de la fig. 6.1 estará ocupada sólo en un 1.5% de sus posiciones. Estas matrices se denominan matrices dispersas (ing., sparse matrix) y es evidente que su representación exige buscar una estrategia alternativa.

Una implementación más razonable guarda sólo los elementos no nulos de la matriz dispersa; cada elemento formará parte exactamente de dos listas diferentes: la de elementos de su fila y la de elementos de su columna. En vez de duplicar los elementos en cada lista, se les asocian dos campos de encadenamiento y así pueden insertarse en las dos listas simultáneamente. Esta estrategia será especialmente útil cuando la relación sea valorada, porque no hará falta duplicar el valor de la relación en varias listas diferentes. La estructura se denomina multilista de grado dos; multilista, porque los elementos forman parte de más de una lista, y de grado dos, porque forman parte exactamente de dos listas. Para abreviar, a las multilistas de grado dos las llamaremos simplemente multilistas (ing., multilist).

Dada esta política de representación, las operaciones sobre las relaciones quedan así:

- Insertar un par: se busca la lista de la fila y la columna del elemento y se encadena la celda. Las celdas se han de mantener odenadas porque la especificación establece que fila y columna deben devolver un conjunto ordenado y así no es necesario un proceso explícito de ordenación; además, la obtención ordenada de los elementos puede ser útil en algunos contextos. Notemos que la inserción ordenada no enlentece en exceso la operación porque, igualmente, para evitar las repeticiones hay que recorrer una de las listas para ver si el par existe o no.
- Borrar un par: se busca el par en la fila o la columna involucradas. Para actualizar las listas hay que conocer la posición que ocupan los elementos que lo preceden, lo que exigirá una exploración adicional en la lista en la que no se haya buscado el par. Esta segunda búsqueda puede evitarse encadenando doblemente las listas correspondientes.
- Comprobar si un par existe: hay que recorrer una de las dos listas a la que pertenece hasta encontrarlo o llegar al final sin éxito. En el caso de que la longitud esperada de uno de los dos tipos de listas (de filas o de columnas) sea significativamente menor que la del otro, tal vez resulte mejor buscar en las primeras.
- Obtener una fila o columna: hay que recorrer la lista y formar el conjunto resultante.

Existen todavía un par de detalles por concretar:

- Todas las celdas de una fila y de una columna están encadenadas, pero no hay un camino de acceso a la primera celda. Es necesario, pues, añadir una tabla que asocie a cada elemento su lista; si el número de filas y columnas es acotado y enumerable, se puede implementar la tabla con un simple vector de apuntadores a la primera celda de cada fila y de cada columna, si no, hay que organizar estos apuntadores en una lista o bien utilizar tablas de dispersión o árboles de búsqueda, si se pide un acceso eficiente.
- Los nodos se almacenan en un único vector o en memoria dinámica según se conozca o no el número de pares que habrá en la relación.
- Dentro de las celdas no hay ninguna información que indique qué columna o fila representan. Hay varias soluciones a este problema:
  - ◊ Añadir a cada celda los identificadores de la fila y la columna a los que pertenece. Es la solución óptima en tiempo, a costa de mantener dos campos adicionales.
  - ◊ Cerrar circularmente las listas, de forma que el último nodo de cada lista "apunte" a su cabecera; entonces la cabecera tendrá que identificar la fila o columna correspondiente. Hay que resaltar que, en este contexto, la circularidad significa que el último elemento de una lista se encadena a su apuntador de inicio citado en el punto anterior. Esta opción es más costosa en tiempo, porque para saber la fila o la columna a la que pertenece un elemento hay que recorrer la lista hasta el final. En cuanto al espacio, puede acarrear más o menos problemas según el uso de vectores o punteros en las diferentes estructuras encadenadas, y dependiendo de que la relación sea valorada o no, lo que puede permitir aprovechar campos ya existentes o bien obligar a declarar alguno adicional, posiblemente usando el mecanismo de tuplas variantes, si el lenguaje de programación lo presenta.
  - ◊ Se puede optar por una solución intermedia: un tipo de listas con identificadores y el otro circular. Esta solución es útil cuando la longitud esperada de uno de los dos tipos es pequeña; así, en el ejemplo de alumnos y asignaturas, como la lista de asignaturas por alumno siempre será muy corta, se podrá implementar circularmente, mientras que para las otras parece mas adecuado el uso de identificadores.

En la figura 6.4 se muestra una representación de las multilistas en las que tanto las filas como las columnas forman listas circulares, usando vectores para acceder al inicio e implementando con punteros las estructuras lineales necesarias; se supone pues que los géneros A y B son válidos como índice de vector lo que se traduce en el uso de ELEM_ORDENADO como universo de caracterización. Destacamos el uso de tuplas variantes para la implementación del tipo de las celdas, y de diversas operaciones privadas cuyo comportamiento se especifica convenientemente. Notemos también que la ausencia de fantasmas complica algunos algoritmos, pero aun así no se incluyen para ahorrar espacio a la estructura. Por último, destaquemos que el invariante evita que dos listas de filas o dos listas de columnas compartan nodos, porque se violaría la condición de igualdad dada.

```
universo MULTILISTA_TODO_CIRCULAR (A, B son ELEM_ORDENADO)
    implementa RELACIÓN (A, B son ELEM_<_=)
    usa BOOL
    tipo relación es tupla
                      fil es vector [A.elem] de ^nodo
                      col es vector [B.elem] de ^nodo
                    ftupla
    ftipo
    tipo privado nodo es
        tupla
            caso ult_fil? de tipo bool igual a
                cierto entonces punt_fil es A.elem
                falso entonces enc_fil es ^nodo
            caso ult_col? de tipo bool igual a
                cierto entonces punt_col es B.elem
                falso entonces enc_col es ^nodo
        ftupla
    ftipo
    invariante (r es relación)
        ∀a: a∈A.elem: r.fil[a] ≠ NULO ⇒ (cabecera_fil(r.fil[a]) = a  ∧ ord_fil(r.fil[a]))   ∧
        ∀b: b∈B.elem: r.col[b] ≠ NULO ⇒ (cabecera_col(r.col[b]) ∧ ord_col(r.col[b])) = b
     donde se define cabecera_fil: ^nodo → A.elem como:
        p^.ult_fil? ⇒ cabecera_fil(p) = p^.punt_fil
        ¬p^.ult_fil? ⇒ cabecera_fil(p) = cabecera_fil(p^.enc_fil),
     ord_fil: ^nodo → bool como:
        p^.ult_fil? ⇒ ord_fil(p) = p^.cierto
        ¬p^.ult_fil? ⇒ ord_fil(p) = (cabecera_col(p) < cabecera_col(p^.enc_fil)) ∧
                                    ∧ ord_fil(p^.enc_fil)
      y de forma similar ord_col y cabecera_col
    función crea devuelve relación es
    var r es relación; a es A.elem; b es B.elem fvar
        para todo a dentro de A.elem hacer r.fil[a] := NULO fpara todo
        para todo b dentro de B.elem hacer r.col[b] := NULO fpara todo
    devuelve r
    función existe? (r es relación; a es A.elem; b es B.elem) devuelve bool es
    var antf, antc, p son ^nodo; está? es bool fvar
            {se apoya íntegramente en la función auxiliar busca}
        <está?, antf, antc, p> := busca(r, a, b)    {está? indica si <a, b> está en r}
    devuelve está?
```

Fig. 6.4: universo para la implementación del tipo abstracto de las relaciones.

```
función inserta (r es relación; a es A.elem; b es B.elem) devuelve relación es
var antf, antc, p son ^nodo; está? es bool fvar
    <está?, antf, antc, p> := busca(r, a, b)
    si ¬está? entonces {si existe, nada}
      p := obtener_espacio
      si p = NULO entonces error  {no hay espacio}
      si no {inserción ordenada en la lista de filas, controlando si la fila está vacía}
         si antf = NULO entonces p^.ult_fil := (r.fil[a] = NULO) si no p^.ult_fil := antf^.ult_fil fsi
         si p^.ult_fil entonces p^.punt_fil := a si no p^.enc_fil := antf^.enc_fil fsi
         si antf = NULO entonces r.fil[a] := p si no antf^.ult_fil := falso; antf^.enc_fil := p fsi
               {repetición del proceso en la lista de columnas}
         si antc = NULO entonces p^.ult_col := (r.col[b] = NULO) si no p^.ult_col := antc^.ult_col fsi
         si p^.ult_col entonces p^.punt_col := b si no p^.enc_col := antc^.enc_col fsi
         si antc = NULO entonces r.col[b] := p si no antc^.ult_col := falso; antc^.enc_col := p fsi
      fsi
    fsi
devuelve r

función borra (r es relación; a es A.elem; b es B.elem) devuelve relación es
var antf, antc, p son ^nodo; está? es bool fvar
        {si existe, se obtienen los apuntadores a la celda y a sus predecesores en las listas}
    <está?, antf, antc, p> := busca(r, a, b)
    si está? entonces {si no existe, nada}
            {supresión de la lista de filas, vigilando si es el primero y/o el último}
        opción
            caso antf = NULO ∧ p^.ult_fil? hacer r.fil[a] := NULO
            caso antf = NULO ∧ ¬ p^.ult_fil? hacer r.fil[a] := p^.enc_fil
            caso antf ≠ NULO ∧ p^.ult_fil? hacer antf^.ult_fil? := cierto; antf^.punt_fil := a
            caso antf ≠ NULO ∧ ¬ p^.ult_fil? hacer antf^.enc_fil := p^.enc_fil
        fopción
            {repetición del proceso en la lista de la columna}
        opción
            caso antc = NULO ∧ p^.ult_col? hacer r.col[b] := NULO
            caso antc = NULO ∧ ¬p^.ult_col? hacer r.col[b] := p^.enc_col
            caso antc ≠ NULO ∧ p^.ult_col? hacer antc^.ult_col? := cierto; antc^.punt_col := a
            caso antc ≠ NULO ∧ ¬p^.ult_col? hacer antc^.enc_col := p^.enc_col
        fopción
        liberar_espacio(p)
        fsi
    devuelve r
```

Fig. 6.4: universo para la implementación del tipo abstracto de las relaciones (cont.).

```
{Funciones fila y columna: siguen el encadenamiento correspondiente y, para cada
 celda, averiguan la columna o la fila siguiendo el otro campo. Esta indagación se lleva a
 término usando diversas funciones auxiliares}
función fila (r es relación; a es A.elem)
    devuelve cjt_b es
var s es cjt_b; p es ^nodo fvar
    s := crea; p := r.fil[a]
    si p ≠ NULO entonces
        mientras ¬p^.ult_fil? hacer
            s := añade(s, qué_col(r, p))
            p := p^.enc_fil
        fmientras
        s := añade(s, qué_col(r, p))
    fsi
devuelve s

función columna (r es relación; b es B.elem)
    devuelve cjt_a es
var s es cjt_a; p es ^nodo fvar
    s :=crea; p := r.col[b]
    si p ≠ NULO entonces
        mientras ¬p^.ult_col? hacer
            s := añade(s, qué_fil(r, p))
            p := p^.enc_col
        fmientras
        s := añade(s, qué_fil(r, p))
    fsi
devuelve s

{Función busca_col(r, p) → <qué_b, antc>: usada en busca, busca la columna que
 representa p dentro de la relación r y, además, devuelve el apuntador al anterior dentro
 de la lista de la columna (que valdrá NULO si el elemento es el primero de la columna);
 existe una función busca_fil, similar pero actuando sobre las listas de filas
        P ≅ p ≠ NULO ∧ (∃b: b∈B.elem: p∈cadena_col(r.col[b]))
        Q ≅ qué_b = cabecera_col(p) ∧
            (antc ≠ NULO ⇒ antc^.enc_col = p) ∧ (antc = NULO ⇒ r.col[qué_b] = p),
 siendo cadena_col similar a cadena siguiendo el encadenamiento de columna}
función privada busca_col (r es relación; p es ^nodo) devuelve <B.elem, ^nodo> es
var antc, p, temp son ^nodo; qué_b es bool fvar
        {primero, se busca la columna}
    temp := p
    mientras ¬temp^.ult_col? hacer temp := temp^.enc_col fmientras
    qué_b := temp^.punt_col   {ya está localizada}
        {a continuación, se busca el anterior a p en la columna}
    antc := NULO; temp := r.col[qué_b]
    mientras temp ≠ p hacer antc := temp; temp := temp^.enc_col fmientras
devuelve <qué_b, antc>

{Función auxiliar qué_col(r, p): busca la columna que representa p dentro de la relación r;
 existe una función qué_fil, similar pero actuando sobre las filas.
        P ≅ p ≠ NULO ∧ (∃b: b∈B.elem: p∈cadena_col(r.col[b]))
        Q ≅ qué_col(r, p) = cabecera_col(p) }
función privada qué_col (r es relación; p es ^nodo) devuelve B.elem es
    mientras ¬ p^.ult_col? hacer  p := p^.enc_col fmientras
devuelve p^.punt_col
```

Fig. 6.4: universo para la implementación del tipo abstracto de las relaciones (cont.).

{Función auxiliar busca(r, a, b) → <encontrado, antf, antc, p>: en caso que el par <a, b> exista, pone encontrado a cierto y devuelve: el apuntador p al elemento que lo representa dentro de las listas de filas y de columnas de la relación r, y también los apuntadores antf y antc a sus antecesores en estas listas; en caso que el par sea el primero de alguna lista devuelve NULO en el puntero correspondiente al anterior en esta llista. En caso que el par <a, b> no exista pone encontrado a falso y antf y antc apuntan a los anteriores (según el criterio de ordenación correspondiente) de b y de a en las listas correspondientes a la fila a y a la columna b; en caso que a o b no tengan menores en la lista correspondiente, el puntero devuelto vale NULO

$P \cong$ cierto

$Q \cong$ encontrado $\equiv \exists q$: q$\in$cadena_fil(r.fil[a]): cabecera_col(q) = b $\wedge$
encontrado $\Rightarrow$ (cabecera_fil(p) = a) $\wedge$ (antf $\neq$ NULO $\Rightarrow$ antf^.enc_fil = p) $\wedge$
(cabecera_col(p) = b) $\wedge$ (antc $\neq$ NUL $\Rightarrow$ antc^.enc_col = p)
$\neg$ encontrado $\Rightarrow$ (antf $\neq$ NULO $\Rightarrow$ cabecera_col(antf) < b) $\wedge$
(antc $\neq$ NULO $\Rightarrow$ cabecera_fil(antc) < a) $\wedge$
(antf = NULO $\wedge$ r.fil[a] $\neq$ NULO) $\Rightarrow$ cabecera_col(r.fil[a]) > b $\wedge$
(antc = NULO $\wedge$ r.col[b] $\neq$ NULO) $\Rightarrow$ cabecera_fil(r.col[b]) > a

siendo cadena_fil similar a cadena siguiendo el encadenamiento de fila}

```
función privada busca (r es relación; a es A.elem; b es B.elem)
                    devuelve <bool, ^nodo, ^nodo, ^nodo> es
var encontrado, éxito es bool; antf, antc, p son ^nodo; qué_b es B.elem fvar
    encontrado := falso; éxito := falso
    si r.fil[a] = NULO entonces antf := NULO
    si no p := r.fil[a]; antf := NULO        {se busca por la fila; antf : anterior en la fila}
        mientras ¬ p^.ult_fil? ∧ ¬ éxito hacer
            <qué_b, antc> := busca_col(r, p)    {antc: anterior en la columna}
            si qué_b ≥ b entonces éxito := cierto; encontrado := (qué_b = b)
                        si no antf := p; p := p^.enc_fil
            fsi
        fmientras
        si ¬ éxito entonces { tratamiento de la última clave }
            <qué_b, antc> := busca_col(r, p)
            si qué_b = b entonces encontrado := cierto
                        si no si qué_b < b entonces antf := p fsi
            fsi
        fsi
    fsi
    ... repetición del proceso en la lista de columnas
 devuelve <encontrado, antf, antc, p>
funiverso
```

Fig. 6.4: universo para la implementación del tipo abstracto de las relaciones (cont.).

6.2 Grafos

En la sección anterior hemos introducido el TAD de las relaciones binarias sobre dos dominios cualesquiera de elementos; si los dos dominios son el mismo, la relación se denomina grafo[3] (ing., graph). Por ello, se puede considerar un grafo como un conjunto de pares ordenados tales que los elementos de cada par cumplen el enunciado de la relación; los elementos del dominio de la relación se llaman vértices o nodos (ing., vertex o node) y el par que representa dos vértices relacionados se llama arista o arc (ing., edge o arc).

Un ejemplo típico de grafo es la configuración topológica de una red metropolitana de transportes, donde los vértices son las estaciones y las aristas los tramos que las conectan. A menudo los grafos representan una distribución geográfica de elementos dentro del espacio[4] (ciudades, componentes dentro de un circuito electrónico, computadores pertenecientes a una red, etc.). La representación de la red metropolitana en forma de grafo permite formular algunas cuestiones interesantes, como por ejemplo averiguar el camino más rápido para ir de una estación a otra. El estudio de éste y otros algoritmos configura el núcleo central del resto del capítulo. Hay que destacar, no obstante, que no se introducen algunas versiones especialmente hábiles de estos algoritmos, puesto que su dificultad excede los objetivos de este libro, sobre todo en lo que se refiere al uso de estructuras de datos avanzadas; el lector que esté interesado puede consultar, por ejemplo, [BrB87] para una visión general y [CLR90] para un estudio en profundidad.

Del mismo modo que hay varios tipos de listas o de árboles, se dispone de diferentes modelos de grafos. Concretamente, definimos cuatro TAD que surgen a partir de las cuatro combinaciones posibles según los dos criterios siguientes:

- Un grafo es dirigido (ing., directed; también se abrevia por digrafo) si la relación no es simétrica; si lo es, se denomina no dirigido.
- Un grafo es etiquetado (ing., labelled o weighted) si la relación es valorada; si no lo es, se denomina no etiquetado. Los valores de la relación son sus etiquetas (ing., label).

En la fig. 6.5 se presenta un ejemplo de cada tipo. Para mostrar los grafos gráficamente, se encierran en un círculo los identificadores de los vértices y las aristas se representan mediante líneas que unen estos círculos; las líneas llevan una flecha si el grafo es dirigido y, si es etiquetado, el valor de la etiqueta aparece a su lado. En cada caso se dice cuál es el conjunto V de vértices y el enunciado R de la relación. En el resto de la sección estudiaremos detalladamente la especificación y la implementación de los grafos dirigidos y etiquetados, y comentaremos brevemente los otros modelos.

[3] Hay una clase particular de grafos, llamados grafos bipartitos (ing., bipartite graph), que pueden definirse sobre dos dominios distintos, que no se estudian aquí; v., por ejemplo, [AHU83, pp.245-249].

[4] De hecho, los grafos fueron definidos en el siglo XVIII por el matemático L. Euler para decidir si era posible atravesar todos los puentes de la ciudad de Königsberg, Prusia (actualmente Kaliningrado, Rusia, ciudad bañada por un río que rodea una isla) sin pasar más de una vez por ninguno de ellos.

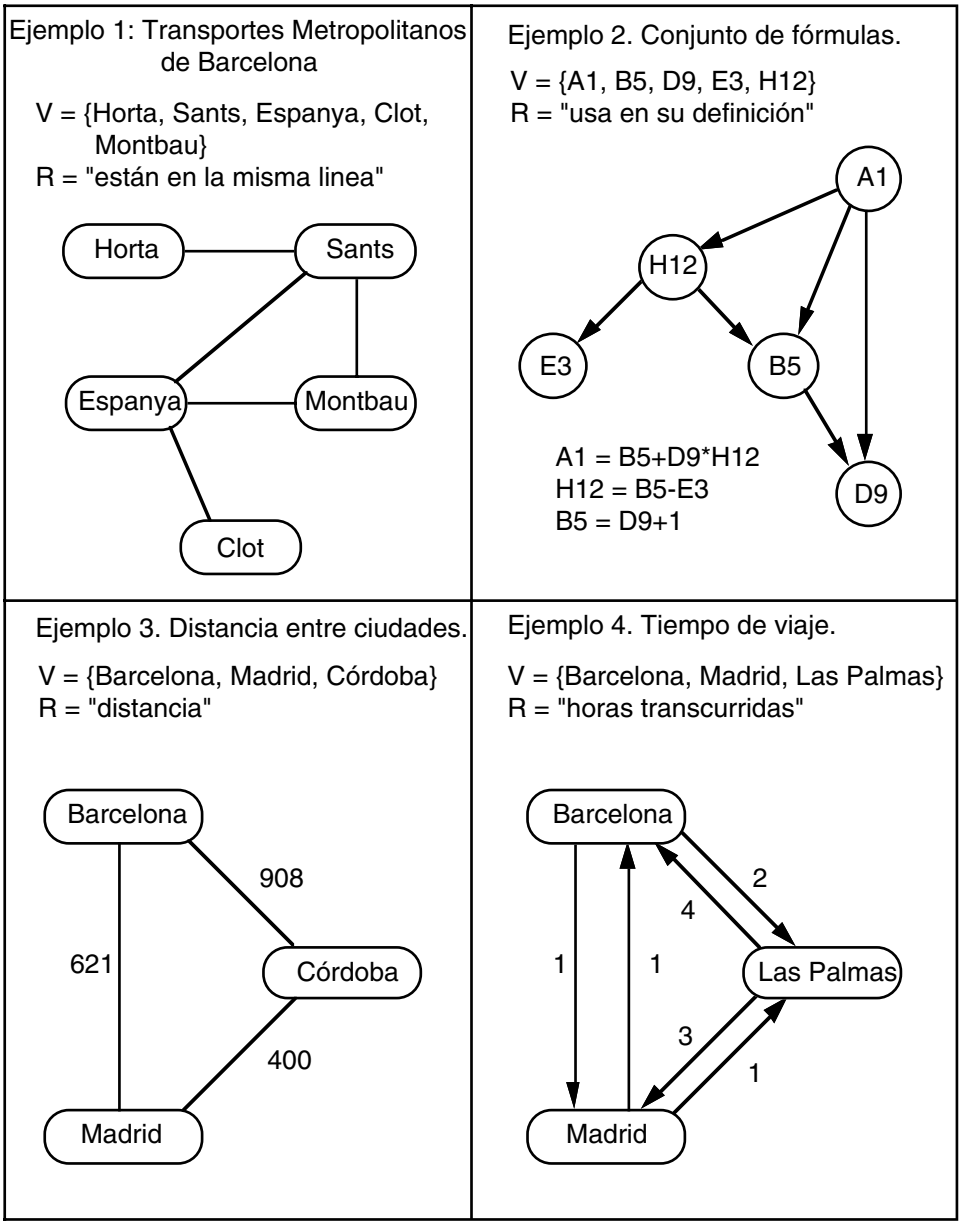


Fig. 6.5: diversos ejemplos de grafos.

6.2.1 Modelo y especificación

Dado un dominio V de vértices y un dominio E de etiquetas, definimos un grafo dirigido y etiquetado, g, como una función que asocia etiquetas a pares de vértices, $g \in \{f : V \times V \rightarrow E\}$ (igual que las relaciones binarias valoradas, de las cuales son un caso particular). A partir de ahora, denotaremos por n el número de vértices del grafo $g \in \{f : V \times V \rightarrow E\}$, $n = ||V||$, y por a el número de aristas, $a = ||dom(g)||$. Por lo que respecta al resto de modelos, en los grafos no dirigidos tenemos que, dados dos vértices cualesquiera u y v, se satisface la condición $<u, v> \in dom(g) \Leftrightarrow <v, u> \in dom(g) \wedge g(u, v) = g(v, u)$, mientras que en el caso de grafos g no etiquetados, el codominio de la función es el álgebra B de valores cierto y falso, $g \in \{f : V \times V \rightarrow B\}$, donde g(u, v) vale cierto si u y v cumplen el enunciado de la relación[5].

Cabe destacar que la mayoría de textos del ámbito de las estructuras de datos definen un grafo como un par ordenado de dos conjuntos, el conjunto de vértices y el conjunto de aristas. La especificación, la implementación y los diversos algoritmos sobre grafos que se presentan en este libro pueden adaptarse sin problemas a este modelo.

De ahora en adelante supondremos que V es finito (y, por consiguiente, dom(g) también); si no, la mayoría de los algoritmos y las implementaciones que estudiaremos en las próximas secciones no funcionarán. Supondremos también que no hay aristas de un vértice a sí mismo (es decir, que la relación es antirreflexiva), ni más de una arista entre el mismo par de vértices si el grafo es no dirigido, ni más de una arista en el mismo sentido entre el mismo par de vértices si el grafo es dirigido; estas propiedades serán garantizadas por la especificación algebraica del tipo. En estas condiciones, el número máximo de aristas[6] de un grafo dirigido de n nodos es n^2-n, si de cada nodo sale una arista al resto (en los grafos no dirigidos, hay que dividir esta magnitud por dos); este grafo se denomina completo (ang., complete o full). Si el grafo no es completo, pero el número de aristas se acerca al máximo, diremos que es denso; por el contrario, si el grafo tiene del orden de n aristas o menos, diremos que es disperso. Supondremos, por último, que E presenta un valor especial $\perp$, que denota la inexistencia de arista; si el valor $\perp$ no puede identificarse en E, se pueden hacer las mismas consideraciones que en el TAD de las relaciones valoradas.

Dado g un grafo dirigido y etiquetado, $g \in \{f : V \times V \rightarrow E\}$, dados $u,v \in V$ dos vértices y $e \in E$ una etiqueta, $e \neq \perp$, definimos las siguientes operaciones sobre el tipo:

- Crear el grafo vacío: crea, devuelve la función Ø tal que dom(Ø) = Ø. Es decir, Ø representa el grafo con todos los vértices, pero sin ninguna arista.
- Añadir una nueva arista al grafo: añade(g, u, v, e) devuelve el grafo g' definido como:
  - ◊ $dom(g') = dom(g) \cup \{<u, v>\} \wedge g'(u, v) = e$
  - ◊ $\forall u', v' : <u', v'> \in dom(g') - \{<u, v>\} : g'(u', v') = g(u', v')$.

[5] En este tipo de grafo se puede adoptar también el modelo $R_{V \times V}$ de las relaciones no valoradas.

[6] El número máximo de aristas interviene en el cálculo de la cota superior del coste de los algoritmos sobre grafos.

- Borrar una arista del grafo: borra(g, u, v) devuelve el grafo g' definido como:
  ◊ dom(g') = dom(g) - {<u, v>}.
  ◊ ∀u',v': <u',v'>∈ dom(g'): g'(u', v') = g(u', v').
- Consultar la etiqueta asociada a una arista del grafo: etiqueta(g, u, v), devuelve g(u, v) si <u, v>∈ dom(g), y ⊥ en caso contrario.
- Obtener el conjunto de vértices a los cuales llega una arista desde uno dado, llamados sucesores (ing., successor), juntamente con su etiqueta: suc(g, u) devuelve el conjunto recorrible s∈ P(V x E), tal que ∀v : v ∈ V : <u, v>∈ dom(g) ⇔ <v, g(u, v)>∈ s.
- Obtener el conjunto de vértices de los cuales sale una arista a uno dado, llamados predecesores (ing., predecessor), juntamente con su etiqueta: pred(g, v) devuelve el conjunto recorrible s∈ P(V x E), tal que ∀v : v ∈ V : <u, v>∈ dom(g) ⇔ <u, g(u, v)>∈ s.

Notemos, pues, que el modelo de los grafos dirigidos etiquetados es casi idéntico al de las relaciones binarias valoradas y esto se refleja en la especificación algebraica correspondiente, que consiste en una instancia de las relaciones, asociando el tipo de los vértices a los dos géneros que definen la relación y el tipo de las etiquetas como valor de la relación; los vértices serán un dominio con una relación de orden <. Previamente se hacen diversos renombramientos para mejorar la legibilidad del universo y, después de la instancia, se renombran algunos símbolos para obtener la nueva signatura. Como la relación se define sobre un único dominio de datos, los tipos de los pares y de los conjuntos definidos en el universo de las relaciones són idénticos y, por ello, se les da el mismo nombre[7]. Por último, se prohíben las aristas de un nodo a sí mismo y la inserción de aristas indefinidas (esta última restricción posiblemente ya existiría en RELACIÓN_VALORADA).

```
universo DIGRAFO_ETIQ(V es ELEM_<_=, E es ELEM_ESP) es
  renombra V.elem por vértice, E.elem por etiq, E.esp por indef
  instancia RELACIÓN_VALORADA(A, B son ELEM_<_=, X es ELEM_ESP) donde
      A.elem, B.elem son vértice, A.=, B.= son V.=, A.<, B.< son V.<
      X.elem es etiq, X.esp es indef
  renombra a_y_valor por vértice_y_etiq, cjt_a_y_valor por cjt_vértices_y_etiqs
           b_y_valor por vértice_y_etiq, cjt_b_y_valor por cjt_vértices_y_etiqs
           _.a por _.v, _.b por _.v, _.val por _.et
           relación por grafo, inserta por añade, consulta por etiqueta
           fila por suc, columna por pred
  error ∀g∈ grafo; ∀v∈ vértice; ∀e∈ etiq: añade(g, v, v, e); añade(g, v, w, indef)
funiverso
```

Fig. 6.6: especificación del TAD de los grafos dirigidos y etiquetados.

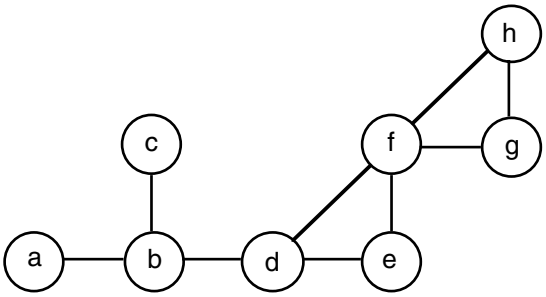
[7] Formalmente este paso no es del todo correcto, porque los dos tipos son diferentes aunque se llamen igual, pero se podría corregir con poco esfuerzo añadiendo funciones de conversión del tipo; como el resultado complicaría el universo innecesariamente, se deja tal como está.

En cuanto al resto de modelos, hay pocos cambios en la especificación. En los grafos no etiquetados desaparece el universo de caracterización de las etiquetas y, en consecuencia, la operación etiqueta, sustituida por la operación existe?: grafo vértice vértice $\rightarrow$ bool, que responde si dos vértices están directamente unidos o no por una arista. Por lo que respecta a los grafos no dirigidos, hay que establecer una relación de conmutatividad de los dos vértices parámetros de añade. Por otro lado, la distinción entre sucesores y predecesores de un vértice no tiene sentido y se sustituye por el concepto más general de adyacencia (ing., adjacency): en un grafo no dirigido, dos vértices son adyacentes si hay una arista que los une.

Una característica común a todos estos modelos de grafo es que el conjunto de sus vértices es fijo; no obstante, ocasionalmente puede interesar que sea un subconjunto W, $W \subseteq V$, que pueda crecer y menguar durante la existencia del grafo. Esta variación introduce un cambio en el modelo porque a las operaciones del tipo habrá que añadir dos más para poner y sacar vértices del grafo, y otra para comprobar su presencia. La construcción precisa del modelo, así como su especificación e implementación, quedan como ejercicio para el lector.

Por último, antes de comenzar a estudiar implementaciones y algoritmos, definimos diversos conceptos a partir del modelo útiles en secciones posteriores (la especificación algebraica de los mismos se propone en el ejercicio 6.1). Un camino (ing., path) de longitud $s \geq 0$ dentro de un grafo $g \in \{f : V \times V \rightarrow E\}$ es una sucesión de vértices $v_0...v_s$, $v_i \in V$, tales que hay una arista entre todo par consecutivo de nodos $<v_i, v_{i+1}>$ de la secuencia. Dado el camino $v_0...v_s$, diremos que sus extremidades son v_0 y v_s, y al resto de vértices los denominaremos intermedios. Igualmente, diremos que es propio (ing., proper) si $s > 0$; que es abierto si $v_0 \neq v_s$, o cerrado en caso contrario; que es simple si no pasa dos veces por el mismo vértice (exceptuando la posible igualdad entre extremidades) y que es elemental si no pasa dos veces por el mismo arco (un camino simple es forzosamente elemental). Por otro lado, el camino $w_0...w_r$ es subcamino (ing., subpath) del camino $v_0...v_s$ si está incluido en él, es decir, si se cumple la propiedad: $\exists i: 0 \leq i \leq s - r - 1: v_0...v_s = v_0...v_i w_0...w_r v_{i+r+2}...v_s$. Notemos que todas estas definiciones son independientes de si el grafo es o no dirigido y de si es o no etiquetado.

Dado un grafo $g \in \{f : V \times V \rightarrow E\}$, diremos que dos vértices $v,w \in V$ están conectados si existe algún camino dentro de g que tenga como extremidades v y w; el sentido de la conexión es significativo si el grafo es dirigido. Si todo par de vértices de V está conectado diremos que g es conexo (ing., connected); en caso de grafos dirigidos, requeriremos la conexión de los vértices en los dos sentidos y hablaremos de un grafo fuertemente conexo (ing., strongly connected). Definimos un componente conexo (ing., connected component; fuertemente conexo en el caso dirigido) del grafo g o, simplemente, componente, como un subgrafo de g conexo maximal (es decir, que no forma parte de otro subgrafo conexo de g); un grafo $g' \in \{f : V \times V \rightarrow E\}$ es subgrafo de g si $dom(g') \subseteq dom(g)$ y además, en caso de ser etiquetado, se cumple que $\forall <u, v>: <u, v> \in dom(g'): g'(u, v) = g(u, v)$

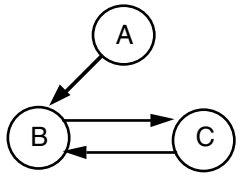


a b c b d e es camino propio abierto no elemental
a b d e f g es camino propio abierto simple
a b d e f d es camino propio abierto elemental no simple
a b d g no es camino

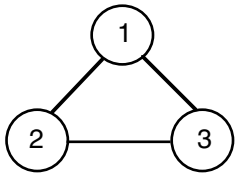
Fig. 6.7: estudio de la existencia de caminos en un grafo.

En un grafo dirigido, un ciclo (ing., cycle) es un camino cerrado propio. Si, además, el camino no contiene ningún subcamino cerrado propio, entonces se denomina ciclo elemental. De forma simétrica, diremos que el ciclo A es un subciclo del ciclo B si A es un subcamino de B. En el caso de grafos no dirigidos, es necesario evitar ciclos de la forma vw...wv (es decir, caminos que vuelven al principio pasando por la misma arista de salida); por tanto, haremos las siguientes precisiones:

- El camino $A = v_0v_1...v_{n-1}v_n$ es simplificable si $v_0 = v_n$ y $v_1 = v_{n-1}$, y su simplificación es el camino v_0.
- La simplificación extendida de un camino es la sustitución de todo sus subcaminos simplificables por la simplificación correspondiente.
- Un ciclo es un camino cerrado propio A tal que su simplificación extendida sigue siendo un camino cerrado propio. Si, además, la simplificación extendida de A no contiene ningún subcamino cerrado propio, entonces el ciclo se denomina ciclo elemental. Diremos que el ciclo $A = v_0...v_0$ es un subciclo del ciclo B si A es un subcamino de B.



A no es un ciclo
A B no es un ciclo
B C B es un ciclo elemental
B C B C B es un ciclo no elemental



1 no es un ciclo
1 2 1 no es un ciclo
1 2 3 1 es un ciclo elemental
1 2 3 2 1 no es un ciclo

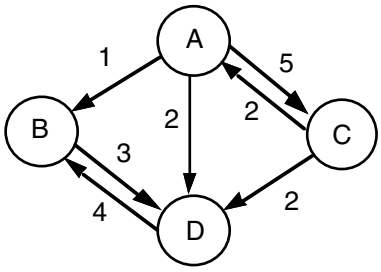
Fig. 6.8: estudio de la existencia de ciclos en un grafo.

6.2.2 Implementación

Estudiaremos tres tipos de implementaciones para grafos dirigidos y etiquetados; la extensión al resto de casos es inmediata y se comenta brevemente al final de la sección. Consideramos que los vértices se pueden usar directamente para indexar vectores[8] y obtenerlos uno detrás de otro en un bucle "para todo" mediante las operaciones definidas en ELEM_ORDENADO (v. fig. 5.30)..

a) Implementación por matriz de adyacencia

Dado g un grafo dirigido y etiquetado definido sobre un dominio V de vértices y con etiquetas de tipo E, $g \in \{f : V \times V \rightarrow E\}$, definimos una matriz bidimensional M, indexada por pares de vértices, que almacena en cada posición $M[u, v]$ la etiqueta de la arista que une u y v. Si entre los vértices u y v no hay ninguna arista, el valor de $M[u, v]$ es el valor indefinido de las aristas (de acuerdo con la especificación del tipo); si la especificación no considera la existencia de este valor especial, se requiere añadir a cada posición un campo booleano que marque la ausencia de arista.



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

Fig. 6.9: un grafo y su implementación por matriz de adyacencia.

En la fig. 6.10 se muestra una posible implementación de esta estrategia. Notemos el uso implícito de las instancias de los pares y de los conjuntos recorribles, formuladas en la especificación de las relaciones y renombradas en la especificación de los grafos. Por su parte, el invariante simplemente confirma la inexistencia de aristas de un vértice a sí mismo, lo cual obliga a que las posiciones de la diagonal principal valgan ⊥. Como esta representación exige comparar etiquetas, el universo de caracterización correspondiente es ELEM_ESP_=, similar a ELEM_2_ESP_= (v. fig. 4.18) pero con una única constante esp. La ordenación de los elementos en suc y pred coincide con el recorrido ordenado de las filas y columnas de la matriz.

[8] Si no, sería necesario usar el TAD tabla como alternativa a los vectores; según la eficiencia exigida, la tabla se podría implementar posteriormente por dispersión, árboles de búsqueda o listas.

```
universo DIGRAFO_ETIQ_MATRIZ(V es ELEM_ORDENADO, E es ELEM_ESP_=) es
  implementa DIGRAFO_ETIQ(V es ELEM_<_=, E es ELEM_ESP)
  renombra V.elem por vértice, E.elem por etiq, E.esp por indef
  usa BOOL
  tipo grafo es vector [vértice, vértice] de etiq ftipo
  invariante (g es grafo): ∀v: v∈ vértice: g[v, v] = indef
  función crea devuelve grafo es
  var g es grafo; v, w son vértice fvar
    para todo v dentro_de vértice hacer
      para todo w dentro_de vértice hacer g[v, w] := indef fpara todo
    fpara todo
  devuelve g
  función añade (g es grafo; v, w son vértice; et es etiq) devuelve grafo es
    si (v = w) ∨ (et = indef) entonces error si no g[v, w] := et fsi
  devuelve g
  función borra (g es grafo; v, w son vértice) devuelve grafo es
    g[v, w] := indef
  devuelve g
  función etiqueta (g es grafo; v, w son vértice) devuelve etiq es
  devuelve g[v, w]
  función suc (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
  var w es vértice; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro_de vértice hacer
      si g[v, w] ≠ indef entonces s := añade(s, <w, g[v, w]>) fsi
    fpara todo
  devuelve s
  función pred (g es grafo; v es vértice) ret cjt_vértices_y_etiqs es
  var w es vértice; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro_de vértice hacer
      si g[w, v] ≠ indef entonces s := añade(s, <w, g[w, v]>) fsi
    fpara todo
  devuelve s
funiverso
```

Fig. 6.10: implementación de los grafos dirigidos etiquetados por matriz de adyacencia.

El coste temporal de la representación, suponiendo una buena implementación de los conjuntos recorribles (v. sección 4.5), queda: crea es $\Theta(n^2)$, las operaciones individuales sobre aristas son $\Theta(1)$ y tanto suc como pred quedan $\Theta(n)$, independientemente de cuántos sucesores o predecesores tenga el nodo en cuestión. Así, el coste de consultar todas las aristas del grafo es $\Theta(n^2)$. El coste espacial de la estructura es considerable, $\Theta(n^2)$.

b) Implementación por listas de adyacencia

Para un grafo g dirigido y etiquetado, definido sobre un dominio V de vértices y con etiquetas de tipo E, $g \in \{f : V \times V \rightarrow E\}$, se asocia a cada vértice una lista con sus sucesores; para acceder al inicio de estas listas, se usa un vector indexado por vértice.

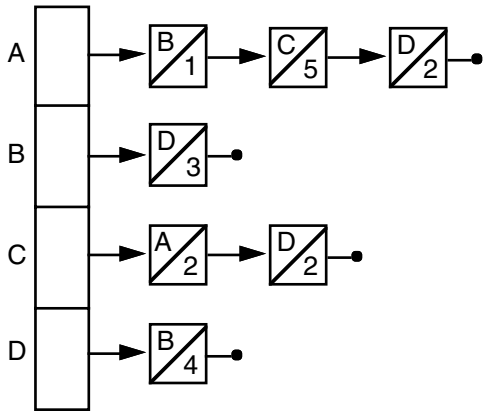


Fig. 6.11: implementación del grafo de la fig. 6.9 por listas de adyacencia.

Para que la implementación sea correcta respecto la especificación, deben ordenarse las listas de sucesores usando la operación <. Ésta es una diferencia fundamental con la implementación de grafos por listas de adyacencia propuestas en otros textos, donde el orden de los nodos es irrelevante. También se podría haber elegido una noción de corrección de las implementaciones menos restrictiva que la usada aquí, pero hemos preferido no salirnos del enfoque seguido en el resto del libro. En todo caso, la eficiencia asintótica no se ve afectada por el hecho de ordenar las listas o no ordenarlas.

En la fig. 6.12 se muestra una posible implementación de esta estrategia, que representa las citadas listas usando una instancia de las listas con punto de interés que incluya operaciones de insertar y borrar según el orden de los elementos; el resultado es una codificación simple, clara, modular y resistente a las modificaciones. Se exige una representación encadenada por punteros, por motivos de eficiencia. Notemos que la situación concreta del punto de interés no es significativa y, por ello, no necesita mantenerse durante la ejecución de las diversas operaciones (si bien podría acarrear alguna inconsistencia al definir el modelo

inicial). El invariante del grafo confirma de nuevo la ausencia de aristas de un vértice a sí mismo con una variante de la típica función cadena, que se define usando operaciones del TAD lista; las listas cumplirán su propio invariante (en concreto, el criterio de ordenación de los elementos). El universo introduce una función auxiliar, busca, que devuelve un booleano indicando si un elemento dado está dentro de una lista o no y, en caso afirmativo, coloca el punto de interés sobre él; gracias a esta función, evitamos algún recorrido redundante de la lista ordenada.

```
universo DIGRAFO_ETIQ_LISTAS(V es ELEM_ORDENADO, E es ELEM_ESP) es
    implementa DIGRAFO_ETIQ(V es ELEM_<_=, E es ELEM_ESP)
    renombra V.elem por vértice, E.elem por etiq, E.esp por indef
    usa BOOL
    instancia LISTA_INTERÉS_ORDENADA(ELEM_<) donde
            elem es vértice_y_etiq, < es V.<
        implementada con LISTA_INTERÉS_ORDENADA_ENC_PUNT(ELEM_<)
    renombra lista por lista_vértices_y_etiq
    tipo grafo es vector [vértice] de lista_vértices_y_etiqs ftipo
    invariante (g es grafo): ∀v: v∈ vértice: v∉ cadena(principio(g[v]))
        donde cadena: lista_vértices_y_etiqs → P(vértice) se define como:
            final?(l) ⇒ cadena(l) = Ø
            ¬final?(l) ⇒ cadena(l) = {actual(l).v} ∪ cadena(avanza(l))
    función crea devuelve grafo es
    var g es grafo; v es vértice fvar
        para todo v dentro de vértice hacer
            g[v] := LISTA_INTERÉS_ORDENADA.crea
        fpara todo
    devuelve g
    función añade (g es grafo; v, w son vértice; et es etiq) devuelve grafo es
    var está? es bool fvar
        si (v = w) ∨ (et = indef) entonces error    {arista reflexiva}
        si no <g[v], está?> := busca(g[v], w)
            si está? entonces
                g[v] := LISTA_INTERÉS_ORDENADA.borra_actual(g[v])
            fsi
            g[v] := LISTA_INTERÉS_ORDENADA.inserta_actual(g[v], <w, et>)
        fsi
    devuelve g
```

Fig. 6.12: implementación de los grafos dirigidos etiquetados por listas de adyacencia.

```
función borra (g es grafo; v, w son vértice) devuelve grafo es
var está? es bool fvar
    <g[v], está?> := busca(g[v], w)
    si está? entonces g[v] := LISTA_INTERÉS_ORDENADA.borra_actual(g[v]) fsi
devuelve g

función etiqueta (g es grafo; v, w son vértice) devuelve etiq es
var está? es bool; et es etiq fvar
    <g[v], está?> := busca(g[v], w)
    si está? entonces et := LISTA_INTERÉS_ORDENADA.actual(g[v]).et si no et := indef fsi
devuelve et

función suc (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
var w es vértice; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro de vértice hacer s := añade(s, actual(g[w])) fpara todo
devuelve s

función pred (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
var w es vértice; está? es bool; s es cjt_vértices_y_etiqs fvar
    s := crea
    para todo w dentro de vértice hacer
       <g[w], está?> := busca(g[w], v)
       si está? entonces s := añade(s, <w, actual(g[w]).et>) fsi
    fpara todo
devuelve s

{Función auxiliar busca(l, v) → <l', está?>: dada la lista de vértices l, busca el vértice v.
 Devuelve un booleano con el resultado de la búsqueda; si vale cierto, el punto de
 interés queda sobre v; si vale falso, queda sobre el elemento mayor de los mayores, o a
 la derecha de todo, si v es mayor que el máximo de l

función privada busca (l es lista_vértices_y_etiqs; v es vértice)
                                          devuelve <lista_vértices_y_etiqs, bool> es
var está, éxito es bool fvar
    l := principio(l); éxito := falso
    mientras ¬final?(l) ∧ ¬ éxito hacer
          si actual(l).v ≥ v entonces éxito := cierto si no l := avanza(l) fsi
    fmientras
    si éxito entonces está := (actual(l).v = v) si no está := falso fsi
devuelve <l, está?>
funiverso
```

Fig. 6.12: implementación de los grafos dirigidos etiquetados por listas de adyacencia (cont.).

Dado el coste constante de las operaciones sobre listas con el punto de interés como medio de referencia y el coste lineal de suprimir un elemento de una lista ordenada, y suponiendo de nuevo una buena representación para los conjuntos recorribles, la eficiencia temporal de la representación presenta las siguientes características:

- crea queda $\Theta(n)$.
- Las operaciones individuales sobre aristas dejan de ser constantes porque ahora es necesario buscar un elemento en la lista de sucesores de un nodo. El coste es, pues, $\Theta(k)$, siendo k el número estimado de aristas que saldrán de un nodo; este coste se convierte en $\Theta(n)$ en el caso peor.
- La operación suc queda $\Theta(k)$, que mejora en el caso medio la eficiencia de la implementación por matrices, porque los nodos sucesores están siempre calculados en la misma representación del grafo. No obstante, el caso peor sigue siendo $\Theta(n)$.
- Finalmente, pred queda $\Theta(a+n)$ en el caso peor; el factor a surge del examen de todas las aristas (si el vértice no tiene ningún predecesor), mientras que n se debe al hecho de que, si el grafo presenta menos aristas que nodos, el tiempo de examen del vector índice pasa a ser significativo. El caso mejor no bajará, pues, de $\Theta(n)$.

Es decir, excepto la creación (que normalmente no influye en el estudio de la eficiencia) y suc, el coste temporal parece favorecer la implementación por matriz. Ahora bien, precisamente el buen comportamiento de suc hace que muchos de los algoritmos que se presentan en el capítulo sean más eficientes con una implementación por listas, porque se basan en un recorrido de todas las aristas del grafo que se pueden obtener en $\Theta(a+n)$, frente al coste $\Theta(n^2)$ de las matrices, siendo $0 \leq a \leq n^2 - n$.

El coste espacial es claramente $\Theta(a+n)$ que, como mucho, es asintóticamente equivalente al espacio $\Theta(n^2)$ requerido por la representación matricial, pero que en grafos dispersos queda $\Theta(n)$. Ahora bien, es necesario tener en cuenta que el espacio para los encadenamientos puede no ser despreciable porque, generalmente, las etiquetas serán enteros y cada celda de las listas ocupará relativamente más que una posición de la matriz de adyacencia; en grafos densos, el coste real puede incluso contradecir los resultados asintóticos (v. ejercicio 6.5). Debe notarse también que, si los vértices no son un tipo escalar, no sólo debe sustituirse el vector índice por una tabla, sino que también es necesario sustituir el campo vértice por un apuntador a una entrada de la tabla en las listas, si se quiere evitar la redundancia de la información. No obstante, esta opción obliga a conocer la representación elegida para la tabla, motivo por el cual no es posible usar directamente instancias de los tipos genéricos, pues se violarían las reglas de transparencia de la información propuesta en el texto. Estos comentarios también son válidos para la representación que estudiamos a continuación.

c) Implementación por multilistas de adyacencia

Ya se ha comentado la extrema ineficiencia de la función pred en el esquema de listas de adyacencia. Si se necesita que esta operación sea lo más rápida posible, se puede repetir la estrategia anterior, pero sustituyendo las listas de sucesores por listas de predecesores, lo que hace que suc sea ineficiente. Ahora bien, si se exige que suc sea también eficiente, deben asociarse dos listas a cada vértice, la de sus sucesores y la de sus predecesores. En realidad esta solución corresponde al concepto de multilista de grado dos usada para implementar las relaciones binarias entre dos conjuntos cualesquiera de elementos. En el caso de los grafos, la relación binaria se establece entre elementos de un mismo conjunto, pero este hecho no afecta al esquema general.

En la fig. 6.14 se ofrece una implementación instanciando el TAD de las relaciones valoradas; notemos que, al contrario de la estrategia anterior, no se obliga a representar las multilistas de una manera concreta, porque las diversas implementaciones que de ellas existen ya se ocupan de optimizar espacio, y será el usuario del grafo quien escogerá la estrategia adecuada en su contexto (listas circulares, con identificadores en las celdas, etc.). El invariante establece, una vez más, la antirreflexividad de la relación. La eficiencia de las operaciones queda como en el caso anterior, excepto pred, que mejora por los mismos razonamientos que suc. Espacialmente, el coste asintótico tampoco cambia, aunque debe considerarse que ahora el vector índice ocupa el doble y que también hay el doble de encadenamientos en las celdas (o más, según la representación de las multilistas).

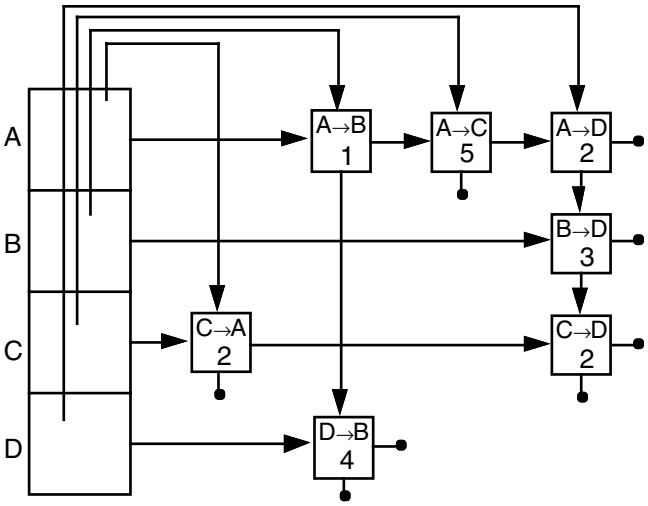


Fig. 6.13: implementación del grafo de la fig. 6.9 por multilistas de adyacencia.

```
universo DIGRAFO_ETIQ_MULTILISTAS(V es ELEM_ORDENADO, E es ELEM_ESP) es
    implementa DIGRAFO_ETIQ(V es ELEM_<_=, E es ELEM_ESP)
    renombra V.elem por vértice, E.elem por etiq, E.esp por indef
    usa BOOL
    instancia RELACIÓN_VALORADA(A, B son ELEM_<_=, X es ELEM_ESP) donde
        A.elem, B.elem son vértice, X.esp es indef, ...
    tipo grafo es relación ftipo
    invariante (g es grafo): ∀v: v∈vértice: v∉fila(g, v) ∧ v∉columna(g, v)
    función crea devuelve grafo es devuelve RELACIÓN_VALORADA.crea
    función añade (g es grafo; v, w son vértice; et es etiq) devuelve grafo es
      si (v = w) ∨ (et = indef) entonces error
                                si no g := RELACIÓN_VALORADA.inserta(g, v, w, et)
      fsi
    devuelve g
    función borra (g es grafo; v, w son vértice) devuelve grafo es
    devuelve RELACIÓN_VALORADA.borra(g, v, w)
    función etiqueta (g es grafo; v, w son vértice) devuelve etiq es
    devuelve RELACIÓN_VALORADA.consulta(g, v, w)
    función suc (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
    devuelve RELACIÓN_VALORADA.fila(g, v)
    función pred (g es grafo; v es vértice) devuelve cjt_vértices_y_etiqs es
    devuelve RELACIÓN_VALORADA.columna(g, v)
funiverso
```

Fig. 6.14: implementación de los digrafos etiquetados por multilistas de adyacencia.

d) Extensión al resto de modelos

Si el grafo es no dirigido, en la matriz de adyacencia no cambia nada en particular, tan sólo cabe destacar su simetría. En las listas y multilistas de adyacencia, puede optarse por repetir las aristas dos veces, o bien guardar la arista una vez solamente, en la lista de sucesores del nodo más pequeño de los dos según el orden establecido entre los vértices. Estas convenciones deben consignarse en el invariante de la representación.

Si el grafo es no etiquetado, el campo de etiqueta de la matriz se sustituye por un booleano que indica existencia o ausencia de arista, mientras que en las representaciones encadenadas simplemente desaparece y entonces el espacio relativo empleado en encadenamientos es todavía más considerable.

6.3 Recorridos de grafos

Con frecuencia, los programas que trabajan con grafos precisan aplicar sistemáticamente un tratamiento a todos los vértices que forman parte de los mismos (es decir, se visitan todos los vértices del grafo). A veces no importa el orden de obtención de los nodos y es suficiente un bucle "para todo" sobre el tipo de los vértices, a veces se necesita imponer una política de recorrido que depende de las aristas existentes. El segundo caso conduce al estudio de diversas estrategias de recorrido de grafos que se presentan en esta sección; como no dependen de si el grafo está etiquetado o no, consideramos el segundo caso.

Por analogía con los árboles, distinguimos diferentes recorridos en profundidad, un recorrido en anchura y el llamado recorrido por ordenación topológica. Todos ellos usan conjuntos para almacenar los vértices visitados en cada momento porque, a diferencia de los árboles, puede accederse a un mismo nodo siguiendo diferentes caminos y deben evitarse las visitas reiteradas. Las operaciones sobre estos conjuntos serán la constante conjunto vacío, añadir un elemento y comprobar la pertenencia y, por ello, nos basaremos en la signatura presentada en la fig. 1.29, definiendo una operación más, $\notin$, como $\neg \in$. Por lo que respecta a la implementación, supondremos que las operaciones son de orden constante (por ejemplo, usando un vector de booleanos indexado por el tipo de los vértices).

Para fijar la signatura definimos los recorridos como funciones recorrido: grafo $\rightarrow$lista_vértice, siendo lista_vértice el resultado de instanciar las listas con punto de interés con el tipo de los vértices. Se podría considerar la alternativa de introducir operaciones de obtención de los vértices uno a uno según las diferentes estrategias; el resultado sería el tipo de los grafos con punto de interés, definibles de manera similar a los árboles. Igualmente, una variante habitual consiste en incluir la visita de los vértices dentro del recorrido de manera que no sea necesario construir primero una lista y después recorrerla; la eficiencia de esta opción choca frontalmente con la modularidad de los enfoques anteriores. Una política diferente que podemos encontrar en diversas fuentes bibliográficas consiste en sustituir la lista por un vector indexado por vértices de modo que en cada posición residirá el ordinal de visita. Notemos que esta versión es un caso particular de devolver una lista y como tal se podría obtener efectuando las instancias oportunas. Ahora bien, debemos destacar que el algoritmo resultante permite eliminar el conjunto de vértices ya visitados, previa inicialización del vector a un valor especial; por otra parte, no es posible un recorrido ordenado rápido posterior del vector, porque es necesario ordenarlo antes.

Tanto en esta sección como en las siguientes nos centraremos en el estudio de los algoritmos y no en su encapsulamiento dentro de universos. Por ejemplo, escribiremos las pre- y postcondiciones y los invariantes de los bucles especialmente interesantes, y supondremos que todas las instancias necesarias de conjuntos, colas, listas, etc., han sido formuladas previamente en alguna parte, explicitando las implementaciones que responden a los requerimientos de eficiencia pedidos. Por lo que respecta al encapsulamiento, tan sólo

decir que hay tres opciones principales:

- Encerrar cada algoritmo individualmente dentro de un universo: el resultado sería un conjunto de módulos muy grande, seguramente difícil de gestionar y por ello poco útil.
- Encerrar todos los algoritmos referidos a un mismo modelo de grafo en un único universo: si bien facilita la gestión de una hipotética biblioteca de tipos de datos, hay que tener presente que diferentes algoritmos pueden exigir diferentes propiedades de partida sobre los grafos (que sean conexos, que no haya etiquetas negativas, etc.).
- Encerrar todos los algoritmos del mismo tipo y referidos a un mismo modelo de grafo en un único universo: parece la opción más equilibrada, porque todos los algoritmos orientados a resolver una misma clase de problemas requieren propiedades muy similares o idénticas sobre el grafo de partida.

6.3.1 Recorrido en profundidad

El recorrido de búsqueda en profundidad (ing., depth-first search; algunos autores añaden el calificativo "prioritaria") o, simplemente, recorrido en profundidad, popularizado por J.E. Hopcroft y R.E. Tarjan el año 1973 en "Efficient algorithms for graph manipulation", Communications ACM, 16(6), pp. 372-378, es aplicable indistintamente a los grafos dirigidos y a los no dirigidos, considerando en el segundo caso cada arista no dirigida como un par de aristas dirigidas. El procedimiento es una generalización del recorrido preorden de un árbol: se comienza visitando un nodo cualquiera y, a continuación, se recorre en profundidad el componente conexo que "cuelga" de cada sucesor (es decir, se examinan los caminos hasta que se llega a nodos ya visitados o sin sucesores); si después de haber visitado todos los sucesores transitivos del primer nodo (es decir, él mismo, sus sucesores, los sucesores de sus sucesores, y así sucesivamente) todavía quedan nodos por visitar, se repite el proceso a partir de cualquiera de estos nodos no visitados. En el resto de la sección, a los sucesores transitivos de un nodo los llamaremos descendientes y, simétricamente, a los predecesores transitivos los llamaremos antecesores; su especificación se propone en el ejercicio 6.1.

El recorrido en profundidad tiene diferentes aplicaciones. Por ejemplo, se puede usar para analizar la robustez de una red de computadores representada por un grafo no dirigido (v. [AHU83, pp. 243-245]). También se puede utilizar para examinar si un grafo dirigido presenta algún ciclo, antes de aplicar sobre él cualquier algoritmo que exija que sea acíclico.

La especificación del recorrido se estudia en el ejercicio 6.31. A la vista de la descripción dada queda claro que, en realidad, no hay un único recorrido en profundidad prioritaria de un grafo, sino un conjunto de recorridos, todos ellos igualmente válidos, y por ello no se especifica el procedimiento de construcción de una solución, sino un predicado que comprueba si una lista de vértices es un recorrido válido para un grafo; generalmente, esta indeterminación no causa ningún problema en la aplicación que necesita el recorrido[9]. Al

[9] Podría obligarse a empezar cada recorrido de un nuevo componente por un nodo sin predecesores.

contrario que los árboles, el indeterminismo es una característica común a todas las estrategias de recorridos sobre grafos.

En la fig. 6.15 se presenta un algoritmo recursivo de recorrido en profundidad prioritaria, que usa un conjunto S para almacenar los vértices ya visitados, y un procedimiento auxiliar para recorrer recursivamente todos los descendientes de un nodo (incluído él mismo); su resultado es la lista de los vértices en orden de visita. Tal como establece el invariante, los elementos de la lista resultado y de S son siempre los mismos, pero el conjunto se mantiene para poder comprobar rápidamente si un vértice ya está en la solución. En el invariante se usa la operación subgrafo, cuya especificación se propone en el ejercicio 6.1. La transformación de este algoritmo en iterativo queda como ejercicio para el lector.

```
{P ≅ g es un grafo dirigido y no etiquetado}
función recorrido_en_profundidad (g es grafo) devuelve lista_vértices es
var S es cjt_vértices; v es vértice; l es lista_vértices fvar
   S := Ø; l := LISTA_INTERÉS.crea
   para todo v dentro_de vértice hacer
      {I ≅ es_recorrido_profundidad(l, subgrafo(g, S)) ∧
            ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ⇒ ∀w: w∈descendientes(g, u): w∈l)}
      si v ∉ S entonces visita_componente(g, v, S, l) fsi
   fpara todo
devuelve l
{Q ≅ es_recorrido_profundidad(l, g)}          -- v. ejercicio 6.31

{P ≅ v ∉S ∧ es_recorrido_profundidad(l, subgrafo(g, S)) ∧
     ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ⇒ ∀w: w∈descendientes(g, u): w∈l)}
acción privada visita_componente (ent g es grafo; ent v es vértice;
                                  ent/sal S es cjt_vértices; ent/sal l es lista_vértices) es
var w es vértice fvar
   S := S ∪ {v}; l := LISTA_INTERÉS.inserta(l, v)
      {visita de todos los descendientes de v}
   para todo w dentro_de suc(g, v) hacer
      {I ≅ es_recorrido_profundidad(l, subgrafo(g, S)) ∧
            ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ∧ u≠v ⇒ ∀w: w∈descendientes(g, u): w∈l)}
      si w ∉ S entonces visita_componente(g, w, S, l) fsi
   fpara todo
facción
{Q ≅ v ∈S ∧ es_recorrido_profundidad(l, subgrafo(g, S)) ∧
     ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ⇒ ∀w: w∈descendientes(g, u): w∈l)}
```

Fig. 6.15: algoritmo del recorrido en profundidad prioritaria.

El recorrido en profundidad estándar se puede modificar adoptando justamente el enfoque simétrico, sin visitar un nodo hasta haber visitado todos sus descendientes, en lo que se puede considerar como la generalización del recorrido postorden de un árbol. Diversos textos denominan a este recorrido recorrido en profundidad con numeración hacia atrás o, simplemente, recorrido en profundidad hacia atrás (ing., backward depth-first search); por el mismo motivo, el recorrido en profundidad estudiado hasta ahora también se denomina recorrido en profundidad hacia delante (ing., forward depth-first search). En la fig. 6.16 se presenta un ejemplo para ilustrar la diferencia. Por lo que respecta al algoritmo de la fig. 6.15, simplemente sería necesario mover la inserción de un nodo a la lista tras el bucle.

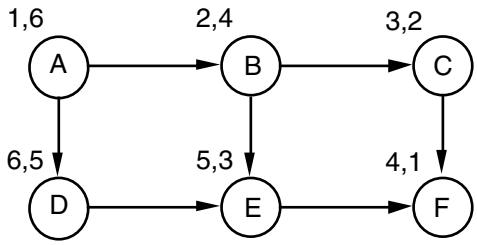


Fig. 6.16: numeración de los vértices de un grafo en un recorrido en profundidad iniciado en A (izquierda: hacia delante; derecha: hacia atrás) considerando el orden alfabético para resolver indeterminaciones.

El coste temporal del algoritmo depende exclusivamente de la implementación elegida para el grafo. Usando una representación por listas o multilistas de adyacencia obtenemos $\Theta(a+n)$, porque, dado un vértice, siempre se consultan las aristas que de él parten, aunque no se visite más que una vez; el factor n aparece por si el grafo tiene menos aristas que vértices. Si se hace la representación mediante matriz de adyacencia el coste es $\Theta(n^2)$, por culpa del tiempo lineal de la operación suc; es decir, si el grafo es disperso, la representación por listas es más eficiente por la rapidez en la obtención de todas las aristas del grafo. Por lo que respecta al espacio adicional empleado, es lineal sobre el número de nodos debido al conjunto.

6.3.2 Recorrido en anchura

El recorrido de búsqueda en anchura o expansión (ing., breadth-first search) o, simplemente, recorrido en anchura es otra estrategia aplicable indistintamente al caso de grafos dirigidos y no dirigidos, que generaliza el concepto de recorrido por niveles de un árbol: después de visitar un vértice se visitan los sucesores, después los sucesores de los sucesores, después los sucesores de los sucesores de los sucesores, y así reiteradamente.

Si después de visitar todos los descendientes del primer nodo todavía quedan más nodos por visitar, se repite el proceso. La especificación de este recorrido se propone también en el ejercicio 6.31.

Una aplicación típica del recorrido en anchura es la resolución de problemas de planificación, donde se dispone de un escenario que se puede modelizar mediante un estado. El estado representa la configuración de un conjunto de elementos de manera que, si cambia su disposición o número, cambia el estado; los cambios de estado atómicos (es decir, que no se pueden considerar como la composición de otros cambios más simples) se denominan transiciones. En estas condiciones, una cuestión habitual consiste en determinar la secuencia más corta de transiciones que lleva de un estado inicial a un estado final, y la pregunta se puede contestar en términos de un recorrido en expansión de un grafo. De hecho, en 1959 E.F. Moore introdujo el recorrido en anchura en este contexto, como una ayuda para atravesar un laberinto ("The shortest path through a maze", en Proceedings of the International Symposium on the Theory of Switching, Harvard Univesity Press). En el apartado 7.1.3 se resuelve este problema para una situación concreta.

El algoritmo sobre grafos dirigidos (v. fig. 6.17) utiliza una cola para poder aplicar la estrategia expuesta. Notemos que el procedimiento principal es idéntico al del recorrido en profundidad prioritaria y que visita_componente es similar al anterior usando una cola de vértices[10]. Asímismo observemos que hay que marcar un vértice (es decir, insertarlo en el conjunto S) antes de meterlo en la cola, para no encolarlo más de una vez. El invariante del procedimiento auxiliar obliga a que todos los elementos entre v y los que están en la cola (pendientes de visita) ya hayan sido visitados, y a que todos los elementos ya visitados tengan sus sucesores, o bien visitados, o bien en la cola para ser visitados de inmediato.

Los resultados sobre la eficiencia son idénticos al caso del recorrido en profundidad.

6.3.3 Recorrido en ordenación topológica

La ordenación topológica (ing., topological sort) es un recorrido solamente aplicable a grafos dirigidos acíclicos, que cumple la propiedad de que un vértice sólo se visita si han sido visitados todos sus predecesores dentro del grafo. De esta manera, las aristas definen una restricción más fuerte sobre el orden de visita de los vértices.

La aplicación más habitual del recorrido en ordenación topológica aparece cuando los vértices del grafo representan tareas y las aristas relaciones temporales entre ellas. Por ejemplo, en el plan de estudios de una carrera universitaria cualquiera, puede haber asignaturas con la restricción de que sólo se puedan cursar si previamente se han aprobado

[10] De hecho, la transformación en iterativo del procedimiento visita_componente del recorrido en profundidad da el mismo algoritmo sustituyendo la cola por una pila.

otras, llamadas prerrequisitos. Este plan de estudios forma un grafo dirigido, donde los nodos son las asignaturas y existe una arista de la asignatura A a la asignatura B, si y sólo si A es prerrequisito de B. Notemos, además, que el grafo ha de ser acíclico: si A es prerrequisito de B (directa o indirectamente) no debe existir ninguna secuencia de prerrequisitos que permitan deducir que B es prerrequisito de A (directa o indirectamente).

```
{P ≅ g es un grafo dirigido no etiquetado}
función recorrido_en_anchura (g es grafo) devuelve lista_vértices es
var S es cjt_vértices; v es vértice; l es lista_vértices fvar
    S := Ø; l := LISTA_INTERÉS.crea
    para todo v dentro de vértice hacer
            {I ≅ es_recorrido_anchura(l, subgrafo(g, S)) ∧
                ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ⇒ ∀w: w∈descendientes(g, u): w∈l)}
        si v ∉ S entonces visita_componente(g, v, S, l) fsi
    fpara todo
devuelve l
{Q ≅ es_recorrido_anchura(l, g)}     -- v. ejercicio 6.31

{P ≅ v ∉ S ∧ es_recorrido_anchura(l, subgrafo(g, S)) ∧
     ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ⇒ ∀w: w∈descendientes(g, u): w∈l)}
acción privada visita_componente (ent g es grafo; ent v es vértice;
                                  ent/sal S es cjt_vértices; ent/sal l es lista_vértices) es
var c es cola_vértices; u, w son vértice fvar
    S := S ∪ {v}; c := COLA.encola(COLA.crea, v)
    mientras ¬COLA.vacía?(c) hacer
            {I ≅ es_recorrido_anchura(l, subgrafo(g, S)) ∧ (∀u: u∈vértice: u∈l ⇔ u∈S) ∧
                (∀u: u∈c: ∀w: w∈ascendientes(g, u)-{u}: w∈descendientes(g, v) ⇒ w∈l) ∧
                (∀u: u∈l: ∀w: w∈suc(g, u): w∈l ∨ w∈c) }
        w := COLA.cabeza(c); c := COLA.desencola(c)
        l := LISTA_INTERÉS.inserta(l, w)
        para todo u dentro de suc(g, v) hacer
            si u ∉ S entonces S := S ∪ {u}; c := COLA.encola(c, u) fsi
        fpara todo
    fmientras
facció
{Q ≅ v ∈ S ∧ es_recorrido_anchura(l, subgrafo(g, S)) ∧
     ∀u: u∈vértice: (u∈l ⇔ u∈S) ∧ (u∈l ⇒ ∀w: w∈descendientes(g, u): w∈l)}
```

Fig. 6.17: algoritmo de recorrido en anchura de un grafo dirigido.

Cuando un alumno se matricula en este plan de estudios, ha de cursar diversas asignaturas (es decir, ha de recorrer parte del grafo del plan de estudios) respetando las reglas:

- Al comenzar, sólo puede cursar las asignaturas (es decir, visitar los nodos) que no tengan ningún prerrequisito (es decir, que no tengan ningún predecesor en el grafo).
- Sólo se puede cursar una asignatura si han sido cursados todos sus prerrequisitos previamente (es decir, si han sido visitados todos sus predecesores).

Así, pues, un estudiante se licencia cuando ha visitado un número suficiente de nodos respetando estas reglas, las cuales conducen a un recorrido en ordenación topológica.

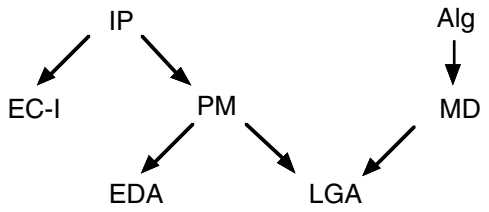


Fig. 6.18: parte del plan de estudios de la Ingeniería en Informática impartida en la Facultat d'Informàtica de Barcelona de la Universitat Politècnica de Catalunya.

La descripción del funcionamiento del algoritmo se presenta en la fig. 6.19: se van escogiendo los vértices sin predecesores, que son los que se pueden visitar y, a medida que se incorporan a la solución se borran las aristas que salen de ellos; la función nodos del invariante devuelve todos los elementos pertenecientes a una lista. Los recorridos obtenidos de esta forma responden a la especificación que se propone en el ejercicio 6.31.

```
{P ≅ g es un grafo dirigido no etiquetado y acíclico}
función ordenación_topológica (g es grafo) devuelve lista_vértices es
var l es lista_vértices; v, w son vértices fvar
   l := LISTA_INTERÉS.crea
   mientras queden nodos por visitar hacer
         {I ≅ es_ordenación_topológica(l, subgrafo(g, nodos(l))) }
      escoger v ∉ l  tal que todos sus predecesores estén en l
      l := LISTA_INTERÉS.inserta(l, v)
   fmientras
devuelve l
{Q ≅ es_ordenación_topológica(l, g)}        -- v. ejercicio 6.31
```

Fig. 6.19: presentación del algoritmo de ordenación topológica.

Obviamente la implementación directa de esta descripción es costosa, ya que hay que buscar vértices sin predecesores repetidas veces. Una alternativa consiste en calcular previamente cuántos predecesores tiene cada vértice, almacenar el resultado en un vector y actualizarlo siempre que se incorpore un nuevo vértice a la solución. En la fig. 6.20 se presenta una implementación de esta última opción, que usa un vector de vértices núm_pred, el cual asocia el número de predecesores que todavía no están en la solución a cada vértice, y un conjunto ceros donde se guardan todos los vértices que se pueden incorporar en el paso siguiente del algoritmo. La inicialización de la estructura, ciertamente rebuscada a simple vista, está diseñada para ser asintóticamente óptima para cualquier representación del grafo; con este objetivo se evita el uso de la operación etiqueta, que exige búsquedas lineales en caso de listas[11]. Nótese el uso de una operación sobre conjuntos, escoger_uno_cualquiera, que devuelve un elemento cualquiera del conjunto y lo borra, y que debe ser ofrecida por la especificación del TAD de los conjuntos o bien por un enriquecimiento.

Para calcular el coste temporal del algoritmo se suman los costes de la inicialización de las estructuras y del bucle principal. En la inicialización, el primer bucle queda $\Theta(n)$, mientras que el segundo exige un examen de todas las aristas del grafo y queda, pues, $\Theta(a+n)$ en el caso de listas de adyacencia y $\Theta(n^2)$ en caso de matriz de adyacencia. En lo que respecta al bucle principal, un examen poco cuidadoso podría llevar a decir que es $\Theta(n^2)$, dado que el bucle se ejecuta n veces (en cada paso se añade un vértice a la solución) y en cada iteración se obtienen los sucesores del nodo escogido, operación que es $\Theta(n)$ en el caso peor (supongamos que escoger_uno_cualquiera es de orden constante). Ahora bien, notemos que la iteración interna exige, a lo largo del algoritmo, el examen de todas las aristas que componen el grafo con un tratamiento constante de cada una, lo que, en realidad, es $\Theta(a)$ y no $\Theta(n^2)$ si el grafo está implementado por listas de adyacencia. Como la operación escoger_uno_cualquiera se ejecuta un total de n-1 veces y es $\Theta(1)$, se puede concluir que la eficiencia temporal del algoritmo es, una vez más, $\Theta(a+n)$ con listas de adyacencia y $\Theta(n^2)$ con matriz de adyacencia y, por ello, la representación por listas de adyacencia es preferible si el grafo es disperso, mientras que si es denso la implementación no afecta a la eficiencia.

[11] Otra opción sería disponer de dos versiones del algoritmo en dos universos diferentes, una para implementación mediante matriz y otra para listas o multilistas. No obstante, preferimos esta opción para simplificar el uso del algoritmo.

```
{P ≅ g es un grafo dirigido no etiquetado y acíclico [12]}
función ordenación_topológica (g es grafo) devuelve lista_vértices es
var l es lista_vértices; v, w son vértice
    ceros es cjt_vértices; núm_pred es vector [vértice] de nat
fvar
        {inicialización de las estructuras en dos pasos}
    ceros := Ø
    para todo v dentro_de vértice hacer
        núm_pred[v] := 0; ceros := ceros ∪ {v}
    fpara todo
    para todo v dentro_de vértice hacer
        para todo w dentro_de suc(g, v) hacer
            núm_pred[w] := núm_pred[w] + 1; ceros := ceros - {w}
        fpara todo
    fpara todo
        {a continuación se visitan los vértices del grafo}
    l := LISTA_INTERÉS.crea
    mientras ceros ≠ Ø hacer
            {I ≅ es_ordenación_topológica(l, subgrafo(g, nodos(l))) ∧
                ∀w: w∈ vértice: núm_pred[w] = ||{<u, w>∈ A_g / u∉ l }|| ∧
                                    w∈ ceros ⇒ núm_pred[w] = 0}
        <ceros, v> := escoger_uno_cualquiera(ceros)
        l := LISTA_INTERÉS.inserta(l, v)
        para todo w dentro_de suc(g, v) hacer
            núm_pred[w] := núm_pred[w] - 1
            si núm_pred[w] = 0 entonces ceros := ceros ∪ {w} fsi
        fpara todo
    fmientras
devuelve l
{Q ≅ es_ordenación_topológica(l, g)}
```

Fig. 6.20: implementación del algoritmo de ordenación topológica.

[12] Como alternativa, se podría controlar la condición de aciclicidad del grafo en el propio algoritmo y entonces desaparecería de la precondición.

6.4 Búsqueda de caminos mínimos

En esta sección nos ocupamos de la búsqueda de caminos minimales dentro de grafos etiquetados no negativamente que sean lo más cortos posible, donde la longitud de un camino, también conocida como coste, se define como la suma de las etiquetas de las aristas que lo componen. Para abreviar, hablaremos de "suma de aristas" con significado "suma de las etiquetas de las aristas" y, por ello, hablaremos de "caminos mínimos" en vez de "caminos minimales". Concretamente, estudiamos un algoritmo para encontrar la distancia mínima de un vértice al resto y otro para encontrar la distancia mínima entre todo par de vértices.

Dada la funcionalidad requerida, es necesario formular algunos requisitos adicionales sobre el dominio de las etiquetas, que se traducen en nuevos parámetros formales de los universos que encapsulen estos algoritmos. Concretamente, consideramos E como un dominio ordenado, donde < es la operación de ordenación y = la de igualdad; para abreviar, usaremos también la operación $\leq$, definible a partir de éstas. También se precisa de una operación conmutativa + para sumar dos etiquetas, que tiene una constante 0, a la cual denominaremos etiqueta de coste nulo, como elemento neutro. Por último, nos interesa que el valor indefinido esp represente la etiqueta más grande posible, por motivos que quedarán claros a lo largo de la sección; para mayor claridad, en los algoritmos que vienen a continuación supondremos que esp se renombra por ∞. Las etiquetas negativas no forman parte del género, tal como establece la última propiedad del universo.

```
universo ELEM_ESP_<_=_+ caracteriza
    usa BOOL
    tipo elem
    ops 0, esp: → elem
        _<_, _≤_, _=_, _≠_: elem elem → bool
        _+_: elem elem → elem
    ecns ...la igualdad y desigualdad se especifican de la manera habitual
        [x ≠ esp] ⇒ x < esp = cierto          {esp es cota superior}
        x < x = falso                         {estudio de la reflexividad...}
        ((x < y) ⇒ (y < x)) = falso           {..de la simetría...}
        ((x < y ∧ y < z) ⇒ (x < z)) = cierto   {...y de la transitividad de <}
        x ≤ y = (x < y) ∨ (x = y)
        x + y = y + x                         {conmutatividad}
        x + 0 = x; x + esp = esp              {elementos neutro e idenpotente}
        (x+y < x) = falso                     {etiquetas no negativas}
funiverso
```

Fig. 6.21: nuevos requerimientos sobre el dominio de las etiquetas.

6.4.1 Camino más corto de un nodo al resto

Dado un grafo $g \in \{f : V \times V \rightarrow E\}$ con etiquetas no negativas, se trata de calcular el coste del camino mínimo desde un vértice dado al resto (ing., single-source shortest paths)[13]. La utilidad de un procedimiento que solucione esta cuestión es clara: el caso más habitual es disponer de un grafo que represente una distribución geográfica, donde las aristas den el coste (en precio, en distancia o similares) de la conexión entre dos lugares y sea necesario averiguar el camino más corto para llegar a un punto partiendo de otro (es decir, determinar la secuencia de aristas para llegar a un nodo a partir del otro con un coste mínimo).

La solución más eficiente a este problema es el denominado algoritmo de Dijkstra, en honor a su creador, E.W. Dijkstra. Formulado en 1959 en "A note on two problems in connexion with graphs", Numerical Mathematica, 1, pp. 269-271, sobre grafos dirigidos (su extensión al caso no dirigido es inmediata y queda como ejercicio para el lector), el algoritmo de Dijkstra (v. fig. 6.22) es un algoritmo voraz[14] que genera uno a uno los caminos de un nodo v al resto por orden creciente de longitud; usa un conjunto S de vértices donde, a cada paso del algoritmo, se guardan los nodos para los que ya se sabe el camino mínimo y devuelve un vector indexado por vértices, de manera que en cada posición w se guarda el coste del camino mínimo que conecta v con w. Cada vez que se incorpora un nodo a la solución, se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él. Por convención, se supone que el camino mínimo de un nodo a sí mismo tiene coste nulo. Un valor infinito en la posición w del vector indica que no hay ningún camino entre v y w.

El invariante asegura que, en cada momento del algoritmo, el vector contiene caminos mínimos formados íntegramente por nodos de S (excepto el último nodo), y que los nodos de S corresponden a los caminos mínimos más cortos calculados hasta el momento. A tal efecto, utiliza diversas funciones: caminos, que da el conjunto de caminos que se pueden encontrar dentro del grafo entre dos nodos; coste, que da el coste de un camino; máximo y mínimo, que dan los elementos mayor y menor de un conjunto, respectivamente. La especificación de la primera se propone en el ejercicio 6.1; la especificación de las otras es sencilla y queda como ejercicio para el lector. Abusando de la notación, se usa el operador de intersección entre un conjunto y un camino con el significado intuitivo.

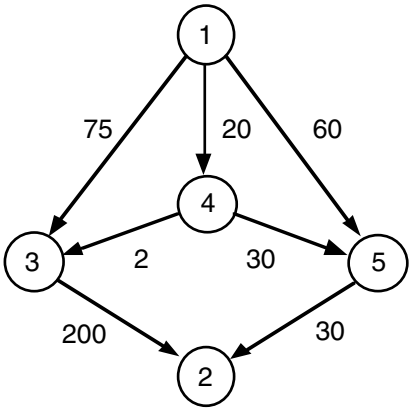
La especificación del algoritmos se propone en el ejercicio 6.32, donde se generaliza el algoritmo, pues el resultado es una tabla de vértices y etiquetas; la versión de la fig. 6.22 es una particularización de este caso, donde la tabla se implementa con un vector porque los vértices son un tipo escalar; el caso general se propone en el mismo ejercicio.

[13] Se podría pensar en el caso particular de encontrar el camino más corto entre dos vértices, pero el algoritmo es igual de costoso que éste más general.

[14] Los algoritmos voraces (ing., greedy) son una familia de algoritmos que se estructuran como un bucle que, en cada paso, calcula una parte ya definitiva de la solución; para estudiar en profundidad sus características puede consultarse, por ejemplo, [BrB87].

```
{P ≅ g es un grafo dirigido etiquetado no negativamente}
función Dijkstra (g es grafo; v es vértice) devuelve vector [vértice] de etiq es
var S es cjt_vértices; D es vector [vértice] de etiq fvar
    ∀w: w∈ vértice: D[w] := etiqueta(g, v, w)
    D[v] := 0; S := {v}
    mientras S no contenga todos los vértices hacer
        {I ≅ ∀w: w∈ vértice: D[w] = mínimo({coste(C) / C∈ caminos(g, v, w) ∧ C∩(S∪{w}) = C})
            ∧ ¬ ( ∃w: w∉ S:  D[w] < máximo({D[u] / u∈ S}) }
        elegir w ∉ S tal que D[w] es mínimo; S := S ∪ {w}
        ∀u: u∉ S: actualizar distancia mínima comprobando si por w hay un atajo
    fmientras
devuelve D
{Q ≅ D = camino_mínimo(g, v)}      -- v. ejercicio 6.32
```

Fig. 6.22: descripción del algoritmo de Dijkstra.



	D				S
0	∞	75	20	60	{1}
0	∞	22	20	50	{1, 4}
0	222	22	20	50	{1, 4, 3}
0	80	22	20	50	{1, 4, 3, 5}
0	80	22	20	50	{1, 4, 3, 5, 2}

Fig. 6.23: ejemplo de funcionamiento del algoritmo de Dijkstra, donde 1 es el nodo inicial.

En la fig. 6.24 se detalla una posible codificación del algoritmo. Notemos que el uso del valor indefinido de las etiquetas como la etiqueta más grande posible permite denotar la inexistencia de caminos, porque cualquier camino entre dos nodos es más pequeño que ∞. Asímismo, notemos que el algoritmo no trabaja con el conjunto S sino con su complementario, T, que debe presentar operaciones de recorrido para aplicarle un bucle "para todo". El número n de vértices es un parámetro formal más, definido en ELEM_ORDENADO. El bucle principal se ejecuta tan sólo n-2 veces, porque el último camino queda calculado después del último paso (no quedan vértices para hallar atajos).

```
{P ≅ g es un grafo dirigido etiquetado no negativamente}
función Dijkstra (g es grafo; v es vértice) devuelve vector [vértice] de etiq es
var T es cjt_vértices; D es vector [vértice] de etiq; u, w son vértices fvar
    T := Ø
    para todo w dentro_de vértice hacer
        D[w] := etiqueta(g, v, w); T := T ∪ {w}
    fpara todo
    D[v] := ETIQUETA.0; T := T - {v}
    hacer n-2 veces {quedan n-1 caminos por determinar}
          {I ≅ ∀w: w∈vértice: D[w] = mínimo({coste(C) / C∈caminos(g, v, w) ∧ (C-{w})∩T = Ø})
               ∧ ¬ ( ∃w: w∈T:  D[w] < máximo({D[u] / u∉T}) }
          {selección del mínimo w : w∈T ∧ (∀u: u∈T: D[w] ≤ D[u])}
        val := ETIQUETA.∞
        para todo u dentro_de T hacer
            si D[u] ≤ val entonces w := u; val := D[u] fsi
            {como mínimo, siempre hay un nodo que cumple la condición}
        fpara todo
            {se marca w como vértice tractado}
        T := T - {w}
            {se recalculan las nuevas distancias mínimas}
        para todo u dentro_de T hacer
            si D[w] + etiqueta(g, w, u) < D[u] entonces D[u] := D[w] + etiqueta(g, w, u) fsi
        fpara todo
    fhacer
devuelve D
{Q ≅ D = caminos_mínimos(g, v)}
```

Fig. 6.24: una codificación posible del algoritmo de Dijkstra.

Se analiza a continuación el tiempo de ejecución, suponiendo que las operaciones sobre conjuntos están implementadas en tiempo constante, excepto la creación (por ejemplo, y aprovechando que los vértices son un tipo escalar, mediante un vector de booleanos). Distinguimos cuatro partes en el algoritmo: inicialización, selección del mínimo, marcaje de vértices y recálculo de las distancias mínimas. Si la representación es mediante matriz de adyacencia, la ejecución de etiqueta es constante y se obtiene:

- Inicialización: creación del conjunto y ejecución n veces de diversas operaciones constantes; queda $\Theta(n)$.
- Selección: las instrucciones del interior del bucle de selección son $\Theta(1)$. Por lo que respecta al número de ejecuciones del bucle, en la primera vuelta se consultan n-1 vértices, en la segunda n-2, etc., ya que la dimensión de T decrece en una unidad en

cada paso; es decir, a lo largo de las n-2 ejecuciones del bucle, sus instrucciones se ejecutarán $n(n-1)/2 - 1$ veces y, así, la eficiencia total del paso de selección será $\Theta(n^2)$.

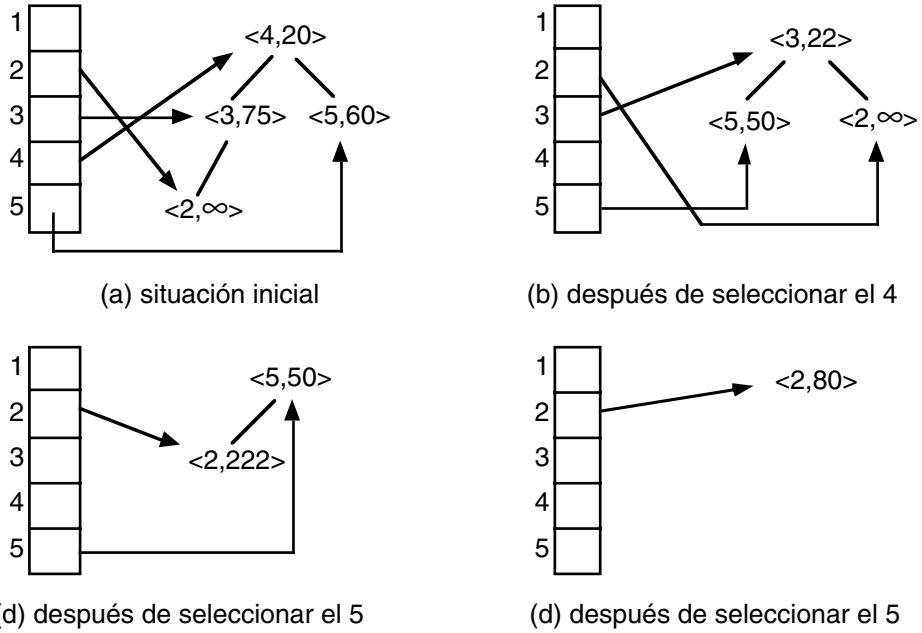
- Marcaje: n supresiones a lo largo del algoritmo conducen a $\Theta(n)$.
- Recálculo de las distancias mínimas: queda $\Theta(n^2)$ por los mismos razonamientos que el paso de selección.

El coste total del algoritmo es la suma de las cuatro etapas, es decir, $\Theta(n^2)$.

En la representación por listas o multilista de adyacencias, lo único que parece cambiar es que la operación etiqueta deja de tener un coste constante. Ahora bien, un análisis más exhaustivo demuestra que en realidad la sentencia "si ... fsi" del paso de recálculo sólo se tiene que ejecutar a veces a lo largo del algoritmo, si se sustituye el bucle sobre los vértices de T por otro bucle sobre los vértices del conjunto de sucesores de w (evitando el uso reiterado de etiq). Como siempre se cumple que $a < n^2$, esta observación puede ser significativa, ya que, si también se consiguiera rebajar el coste del paso de selección, el algoritmo de Dijkstra para grafos no densos representados con estructuras encadenadas sería más eficiente que la representación mediante matriz.

Para alcanzar este objetivo basta con organizar astutamente el conjunto de vértices todavía no tratados, teniendo en cuenta que se selecciona el elemento mínimo. Por ejemplo, se puede sustituir el conjunto por una cola prioritaria, siendo sus elementos pares <w, et> de vértice (todavía no tratado) y etiqueta, que juega el papel de prioridad, con significado "la distancia mínima de v a w es et" (et sería el valor D[w] de la fig. 6.22); las operaciones de obtención y supresión del mínimo encajan perfectamente dentro del algoritmo. Ahora bien, el recálculo de las distancias del último paso puede exigir cambiar la prioridad de un elemento cualquiera de la cola. Por este motivo se define una nueva modalidad de cola con prioridad que permite el acceso directo a cualquier elemento y que, además de las operaciones habituales, incorpora tres más: la primera, para cambiar la etiqueta asociada a un vértice cualquiera por una más pequeña, reorganizando la cola si es necesario; otra, para obtener el valor asociado a un vértice (que valdrá ∞ si el vértice no está en la cola); y una tercera, para comprobar la presencia en la cola de un vértice dado. En la fig. 6.26 se muestra la signatura resultante de este tipo, aplicada al contexto del algoritmo, junto con su coste asintótico[15], y en la fig. 6.25 se muestra un ejemplo de su funcionamiento suponiendo que, para mayor eficiencia, la cola se implementa con un montículo junto con un vector indexado por vértices, de forma que cada posición apunte al nodo del montículo que contenga la etiqueta asociada al vértice; la especificación e implementación del tipo quedan como ejercicio para el lector.

[15] En esta signatura se observa la ausencia de la función organiza de transformación de un vector en una cola (v. apartado 5.5.2), más eficiente que la creación de la cola por inserciones individuales. Dicha ausencia se debe a que el algoritmo queda más claro sin que su coste asintótico se vea afectado. Si se considera conveniente, el algoritmo puede adaptarse fácilmente a esta modificación.



(a) situación inicial (b) después de seleccionar el 4

(d) después de seleccionar el 5 (d) después de seleccionar el 5

Fig. 6.25: evolución de la cola prioritaria aplicando el algoritmo sobre el grafo de la fig. 6.23.

crea: → colapr_aristas; Θ(1)
inserta: colapr_aristas vértice etiq → colapr_aristas; Θ(logn)
menor: colapr_aristas → vértice_y_etiq; Θ(1)
borra: colapr_aristas → colapr_aristas; Θ(logn)
sustituye: colapr_aristas vértice etiq → colapr_aristas; Θ(logn)
valor: colapr_aristas vértice → etiq; Θ(1)
está?: colapr_aristas vértice → bool; Θ(1)
vacía?: colapr_aristas → bool; Θ(1)

Fig. 6.26: signatura para las colas prioritarias extendidas con las operaciones necesarias.

El algoritmo resultante se muestra en la fig. 6.27. La inicialización requiere dos bucles para evitar el uso de la operación etiqueta. Se podría modificar el algoritmo para que las etiquetas infinitas no ocupasen espacio en el montículo y mejorar así el tiempo de ejecución de las operaciones que lo actualizan. El análisis de la eficiencia temporal da como resultado:

- La inicialización es $\Theta(\Sigma k: 1 \leq k \leq n\text{-}1: \log k) + \Theta(n\log n) = \Theta(n\log n) + \Theta(n\log n) = \Theta(n\log n)$, dado que el paso k-ésimo del primer bucle ejecuta una inserción en una cola de tamaño k y todos los pasos del segundo bucle (n-1 como máximo) modifican una cola de tamaño n.

- Las n-1 selecciones del mínimo son constantes, y su supresión, logarítmica sobre n, y queda, pues, $\Theta(n) + \Theta(\Sigma k: 1 \leq k \leq n\text{-}1: \log k) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$.
- El bucle interno examina todas las aristas del grafo a lo largo del algoritmo y, en el caso peor, efectúa una sustitución por arista, de coste $\Theta(\log n)$, y así queda $\Theta(a \log n)$.

En definitiva, el coste temporal es $\Theta((a+n)\log n)$, mejor que la versión anterior si el grafo es disperso; no obstante, si el grafo es denso, el algoritmo original es más eficiente. El espacio adicional continua siendo asintóticamente lineal sobre n.

```
{P ≅ g es un grafo dirigido etiquetado no negativamente}
función Dijkstra (g es grafo; v es vértice) devuelve vector [vértice] de etiq es
var D es vector [vértice] de etiq; A es colapr_aristas; u, w son vértices; et, val son etiq fvar
        {creación de la cola con todas las aristas <v, w>}
    A := COLA_EXTENDIDA.crea
    para todo w dentro_de vértice hacer A := COLA_EXTENDIDA.inserta(A, w, ∞) fpara todo
    para todo <w, et> dentro_de suc(g, v) hacer
        A := COLA_EXTENDIDA.modif(A, w, et)
    fpara todo
    mientras ¬COLAPR_EXTENDIDA.vacía?(A) hacer
        {I ≅ ∀w: w∈vértice: x = mínimo({coste(C) / C∈caminos(g, v, w) ∧
                                                ∀u: u∈C: u ≠ w ⇒ ¬está?(A, u)})
              ∧ menor(A).et ≥ máximo({D[u] / ¬ está?(A, u)})
               definiendo x: está?(A, w) ⇒ x = valor(A, w) ∧ ¬está?(A, w) ⇒ x = D[w] }
        <w, val> := COLAPR_EXTENDIDA.menor(A)   {selección de la arista mínima}
        D[w] := val; A := COLAPR_EXTENDIDA.borra(A)  {marcaje de w  como tratado}
            {recálculo de las nuevas distancias mínimas}
        para todo <u, et> dentro_de suc(g, w) hacer
            si COLAPR_EXTENDIDA.está?(A, u) entonces
                si val+et < COLAPR_EXTENDIDA.valor(A, u) entonces
                    A := COLAPR_EXTENDIDA.sustituye(A, u, val + et)
                fsi
            fsi
        fpara todo
    fmientras
    D[v] := ETIQ.0
devuelve D
{Q ≅ D = caminos_mínimos(g, v)}
```

Fig. 6.27: el algoritmo de Dijkstra usando colas prioritarias.

La eficiencia temporal se puede mejorar aún más si estudiamos el funcionamiento del montículo. Destaquemos que el algoritmo efectúa exactamente n-1 hundimientos de elementos y, por otro lado, hasta un máximo de a flotamientos de manera que, en el caso general, hay más flotamientos que hundimientos. Estudiando estos algoritmos (v. fig. 5.42), se puede observar que un hundimiento requiere un número de comparaciones que depende tanto de la aridad del árbol como del número de niveles que tiene, mientras que un flotamiento depende tan sólo del número de niveles: cuantos menos haya, más rápido subirá el elemento hacia arriba. Por tanto, puede organizarse el montículo en un árbol k-ario y no binario, de manera que la altura del árbol se reduce. En [BrB87, pp. 93-94] se concluye que, con esta estrategia, el coste del algoritmo queda, para grafos densos, $\Theta(n^2)$ y, para otros tipos, se mantiene $\Theta(a \log n)$, de manera que la versión resultante es la mejor para cualquier grafo representado por listas de adyacencia, independientemente de su número de aristas[16].

Notemos que las diferentes versiones del algoritmo de Dijkstra presentadas aquí no devuelven la secuencia de nodos que forman el camino mínimo. La modificación es sencilla y queda como ejercicio para el lector: notemos que, si el camino mínimo entre v y w pasa por un vértice intermedio u, el camino mínimo entre v y u es un prefijo del camino mínimo entre v y w, de manera que basta con devolver un vector C, tal que C[w] contenga el nodo anterior en el camino mínimo de v a w (que será v si está directamente unido al nodo de partida, o si no hay camino entre v y w). Este vector tendrá que ser actualizado al encontrarse un atajo en el camino (en realidad, el vector representa un árbol de expansión -v. sección 6.5- con apuntadores al padre, donde v es la raíz). Es necesario también diseñar una nueva acción para recuperar el camino a un nodo dado, que tendría como parámetro C.

6.4.2 Camino más corto entre todo par de nodos

Se trata ahora de determinar el coste del camino más corto entre todo par de vértices de un grafo etiquetado (ing., all-pairs shortest paths); de esta manera, se puede generalizar la situación estudiada en el apartado anterior, y queda plenamente justificada su utilidad. Una primera solución consiste en usar repetidamente el algoritmo de Dijkstra variando el nodo inicial; ahora bien, también se dispone del llamado algoritmo de Floyd, que proporciona una solución más compacta y elegante pensada especialmente para esta situación. El algoritmo de Floyd, definido por R.W. Floyd en 1962 sobre grafos dirigidos (su extensión al caso no dirigido es inmediata y queda como ejercicio para el lector) en "Algorithm 97: Shortest path", Communications ACM, 5(6), p. 345, es un algoritmo dinámico[17] que se estructura como un

[16] Hay implementaciones asintóticamente más rápidas que usan una variante de montículo llamada montículo de Fibonacci (ing., Fibonacci heap), introducida por M. Fredman y R. Tarjan en 1987, que por su complejidad no se explica en este texto; v., por ejemplo, [HoS94, pp. 488-494].

[17] La programación dinámica (ing., dynamic programming) es una técnica que estructura los algoritmos como bucles que, en cada paso, se acercan más a la solución, pero sin asegurar la obtención de ninguna parte definitiva antes de la finalización del algoritmo; para más detalles consultar, por ejemplo, [BrB87].

bucle, que trata un vértice denominado pivote en cada iteración, y que usa un vector bidimensional D indexado por pares de vértices para guardar la distancia mínima. Cuando u es el pivote, se cumple que D[v, w] es la longitud del camino más corto entre v y w formado íntegramente por pivotes de pasos anteriores (excluyendo posiblemente las extremidades). La actualización de D consiste en comprobar, para todo par de nodos v y w, si la distancia entre ellos se puede reducir pasando por el pivote u mediante el cálculo de la fórmula D[v, w] = min(D[v, w], D[v, u] + D[u, w]). Así pues, hay un par de diferencias en el funcionamiento de este algoritmo y el de Dijkstra: por un lado, los vértices se tratan en un orden independiente de las distancias y, por el otro, no se puede asegurar que ninguna distancia mínima sea definitiva hasta que acaba el algoritmo. Debe notarse que el algoritmo funciona incluso con aristas negativas, siempre que no haya ciclos negativos (es decir, que la suma de las etiquetas de las aristas que los componen no sea negativa), motivo por el que se podría relajar la restricción de aristas no negativas.

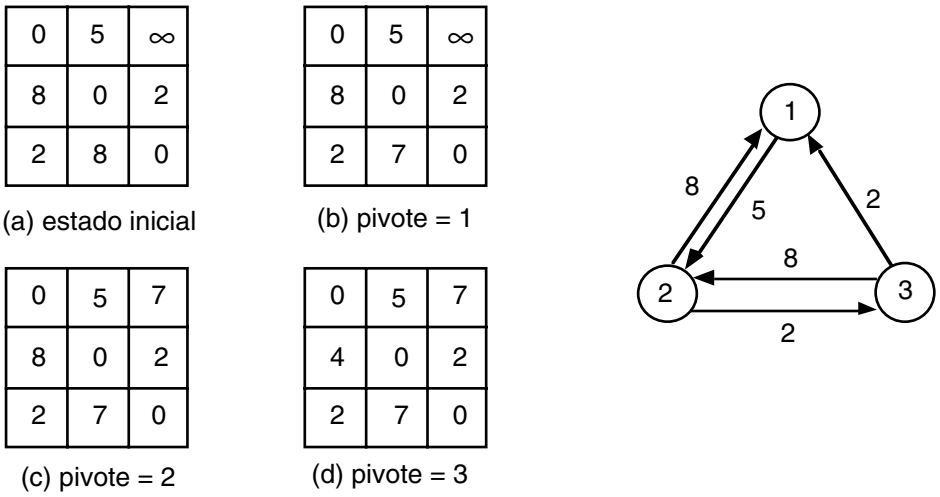


Fig. 6.28: ejemplo de funcionamiento del algoritmo de Floyd.

La especificación del problema se propone en el ejercicio 6.32; los comentarios son los mismos que en la especificación del algoritmo de Dijkstra. La implementación se presenta en la fig. 6.29. Notemos que la diagonal de D se inicializa a cero y nunca cambia. Por otra parte, observemos que tanto la fila como la columna del vértice u tratado en un paso del algoritmo permanecen invariables en este paso, por lo que se puede trabajar con una única matriz para actualizar las distancias mínimas. La inicialización de la matriz es ineficiente en caso de que el grafo esté implementado por (multi)listas de adyacencia, pero se deja así porque no afecta al coste total del algoritmo; si se considera conveniente, se puede modificar de manera similar al algoritmo de Dijkstra de la fig. 6.27, sustituyendo las n^2 llamadas a etiqueta por n bucles sobre los sucesores, de coste total a. El invariante usa una función pivotes(u), que devuelve

el conjunto de vértices que han sido pivotes en pasos anteriores del algoritmo, lo que se puede determinar a partir del pivote del paso actual dado que los vértices son un tipo escalar (si no, el invariante necesitaría mantener un conjunto de los vértices que han sido pivotes).

```
{P ≅ g es un grafo dirigido etiquetado que no contiene ciclos negativos}
función Floyd (g es grafo) devuelve vector [vértice,vértice] de etiq es
var D es vector [vértice, vértice] de etiq; u, v, w son vértice fvar
        {inicialmente la distancia entre dos vértices tiene el valor de la arista que
         los une; las diagonales se ponen a cero}
    para todo v dentro de vértice hacer
        para todo w dentro de vértice hacer
            D[v, w] := etiqueta(g, v, w)   {∞ si no hay arco}
        fpara todo
        D[v, v] := ETIQUETA.0
    fpara todo
    para todo u dentro de vértice hacer
        {I ≅ ∀v: v∈vértice: ∀w: w∈vértice:
                D[v, w] = mínimo({coste(C) / C∈caminos(g, v, w) ∧
                                              C∩(pivotes(u)∪{v, w}) = C})
        para todo v dentro de vértice hacer
            para todo w dentro de vértice hacer
                si D[v, u]+D[u, w] < D[v, w] entonces D[v, w]:= D[v, u] + D[u, w] fsi
            fpara todo
        fpara todo
    fpara todo
devuelve D
{Q ≅ D = todos_caminos_mínimos(g, v)}   -- v. ejercicio 6.32
```

Fig. 6.29: una codificación del algoritmo de Floyd.

La eficiencia temporal del algoritmo de Floyd es $\Theta(n^3)$, independientemente de la representación, igual que la aplicación reiterada de Dijkstra trabajando con una representación del grafo mediante matriz de adyacencia. Ahora bien, es previsible que Floyd sea más rápido (dentro del mismo órden asintótico) a causa de la simplicidad de cada paso del bucle y, como mínimo, siempre se puede argumentar que Floyd es más simple y elegante. Si el grafo está implementado por listas o multilistas de adyacencia y el algoritmo de Dijkstra se ayuda de colas prioritarias, el coste queda $\Theta(a \log n)n = \Theta(an \log n)$. La comparación entre este orden de magnitud y n^3 determina la solución más eficiente. Por último, debe notarse que el algoritmo de Floyd exige un espacio adicional $\Theta(1)$, mientras que cualquier versión de Dijkstra precisa una estructura $\Theta(n)$.

El algoritmo original de Floyd tampoco almacena los vértices que forman el camino; su modificación sigue la misma idea que en el caso del algoritmo de Dijkstra: si el camino mínimo de m a n pasa primero por p y después por q, la secuencia de vértices que forman el camino mínimo de p a q forma parte de la secuencia de vértices que forman el camino mínimo de m a n. Para implementar esta idea, se puede definir un vector bidimensional C indexado por vértices, de modo que C [v, w] contiene un nodo u que forma parte del camino mínimo entre v y w. La concatenación de los caminos que se obtienen aplicando el mismo razonamiento sobre C [v, u] y C [u, w] da el camino mínimo entre v y w. La modificación del algoritmo queda como ejercicio.

6.5 Árboles de expansión minimales

Los algoritmos de la sección anterior permiten encontrar la conexión de coste mínimo entre dos vértices individuales de un grafo etiquetado. No obstante, a veces no interesa tanto minimizar conexiones entre nodos individuales como obtener un nuevo grafo que sólo contenga las aristas imprescindibles para una optimitzación global de las conexiones entre todos los nodos. Generalmente, esta optimización global implicará que diversos pares concretos de nodos no queden conectados entre ellos con el mínimo coste posible dado el grafo de partida.

Una aplicación inmediata es nuevamente la resolución de problemas que tienen que ver con distribuciones geográfica. Por ejemplo, supongamos que se dispone de un conjunto de ordenadores distribuidos geográficamente entre diferentes ciudades (posiblemente en diferentes países), a los que se quiere conectar para que puedan intercambiar datos, compartir recursos, etc. Hay muchas maneras de implementar esta conexión, pero supongamos que aquí se opta por pedir a las compañias telefónicas respectivas los precios del alquiler de la línea entre las ciudades implicadas. Una vez conocida esta información, para escoger las líneas que soportarán la red seguiremos una política austera, que asegure que todos los ordenadores se puedan comunicar entre ellos minimizando el precio total de la red. Pues bien, este problema puede plantearse en términos de grafos considerando que los ordenadores son los vértices y las líneas telefónicas son las aristas, formando un grafo no dirigido y etiquetado, del cual se quiere seleccionar el conjunto mínimo de aristas que permitan conectar todos los vértices con el mínimo coste posible. Para formular la resolución de este problema supondremos en el resto de esta sección que las etiquetas cumplen los mismos requerimientos que en la sección anterior.

Para plantear en términos generales los algoritmos que resuelven este tipo de enunciado, introduciremos primero unas definiciones (v. ejercicio 6.33 para su especificación):

- Un árbol libre (ing., free tree) es un grafo no dirigido conexo acíclico. Puede demostrarse por reducción al absurdo que todo árbol libre con n vértices presenta exactamente n -1 aristas; si se añade una arista cualquiera a un árbol libre se introduce un ciclo mientras que, si se borra, quedan vértices no conectados. También puede demostrarse que en un árbol libre cualquier par de vértices está unido por un único camino simple. Este tipo de árbol generaliza los tipos vistos en el capítulo 5, que se llaman por contraposición árboles con raíz (ing., rooted tree). De hecho, un árbol con raíz no es más que un árbol libre en el que un vértice se distingue de los otros y se identifica como raíz.
- Sea $g \in \{f : V \times V \rightarrow E\}$ un grafo no dirigido conexo y etiquetado no negativamente; entonces, un subgrafo g' de g, $g' \in \{f : V \times V \rightarrow E\}$, es un árbol de expansión para g (ing., spanning tree ; también llamado árbol de recubrimiento o árbol generador) si es un árbol libre.
- Se define el conjunto de árboles de expansión para el grafo $g \in \{f : V \times V \rightarrow E\}$ no dirigido, conexo y etiquetado no negativamente, denotado por ae(g), como:

  $ae(g) = \{g' \in \{f : V \times V \rightarrow E\} / g'$ es árbol de expansión para g$\}$.
- Sea $g \in \{f : V \times V \rightarrow E\}$ un grafo no dirigido conexo y etiquetado no negativamente; entonces, el grafo no dirigido y etiquetado $g' \in \{f : V \times V \rightarrow E\}$ es un árbol de expansión de coste mínimo para g (ing., minimum cost spanning tree; para abreviar, los llamaremos árboles de expansión minimales), si g' está dentro de ae(g) y no hay ningún otro g'' dentro de ae(g), que la suma de las aristas de g'' sea menor que la suma de las aristas de g'.
- Se define el conjunto de árboles de expansión minimales para el grafo $g \in \{f : V \times V \rightarrow E\}$ no dirigido conexo y etiquetado no negativamente, aem(g), como:

  $aem(g) = \{g' \in \{f : V \times V \rightarrow E\} / g'$ es árbol de expansión minimal para g$\}$.

Dadas estas definiciones, se puede formular la situación del inicio del apartado como la búsqueda de un árbol de expansión minimal para un grafo no dirigido, conexo y etiquetado no negativamente. Hay varios algoritmos que resuelven este problema desde que, en el año 1926, O. Boruvka formuló el primero; en este texto, examinaremos dos que dan resultados satisfactorios, a pesar de que hay otros que pueden llegar a ser más eficientes y que, por su dificultad, no explicamos aquí (v., por ejemplo, [Tar83, cap. 6] y [CLR90, cap. 24]). Ambos algoritmos se basan en una misma propiedad que llamaremos propiedad de los árboles de expansión minimales: sea g un grafo no dirigido conexo y etiquetado no negativamente, $g \in \{f : V \times V \rightarrow E\}$, y sea U un conjunto de vértices tal que $U \sqsubset V$, $U \neq \emptyset$; si <u, v> es la arista más pequeña de g tal que $u \in U$ y $v \in V\text{-}U$, existe algun $g' \in aem(g)$ que la contiene, es decir que cumple g'(u, v) = g(u, v). La demostración por reducción al absurdo queda como ejercicio para el lector (v. [AHU83, pp. 234]).

6.5.1 Algoritmo de Prim

El algoritmo de Prim, definido por V. Jarník en el año 1930 y redescubierto por R.C. Prim y E.W. Dijkstra en 1957 (el primero en "Shortest connection networks and some generalizations", Bell System Technical Journal, 36, pp. 1389-1401, y el segundo en la misma referencia dada en el apartado 6.4.1), es un algoritmo voraz que aplica reiteradamente la propiedad de los árboles de expansión minimales, incorporando a cada paso una arista a la solución. El algoritmo (v. fig. 6.30) dispone de un conjunto U de vértices tratados (inicialmente, un vértice v cualquiera, porque todo vértice aparece en el árbol de expansión) y de su complementario, de manera que, de acuerdo con la propiedad de los árboles de expansión minimales, se selecciona la arista mínima que une un vértice de U con otro de su complementario, se incorpora esta arista a la solución y se añade su vértice no tratado dentro del conjunto de vértices tratados. La noción de arista se implementa como un trío <vértice, vértice, etiqueta>. El invariante usa la operación subgrafo cuya especificación se propone en el ejercicio 6.1. Su coste es $\Theta(na)$, que puede llegar hasta $\Theta(n^3)$ si el grafo es denso.

```
{P ≅ g es un grafo no dirigido conexo etiquetado no negativamente}
función Prim (g es grafo) devuelve grafo[18] es
var U es cjt_vértices; gres es grafo; u, v son vértice; x es etiq fvar
    gres := crea; U := {un vértice cualquiera}
    mientras U no contenga todos los vértices hacer
            {I ≅ subgrafo(gres, U) ∈ aem(subgrafo(g, U))}
        seleccionar <u, v, x>  mínima tal que u ∈ U, v ∉ U
        gres := añade(gres, u, v, x); U := U ∪ {v}
    fmientras
devuelve gres
{Q ≅ gres ∈ aem(g)}
```

Fig. 6.30: presentación del algoritmo de Prim.

La versión preliminar se puede refinar y obtener el algoritmo de la fig. 6.32 de coste $\Theta(n^2)$, en general mejor que la anterior (nunca peor, porque la precondición garantiza que a vale como mínimo n-1). Esta nueva versión usa un vector arista_mín, indexado por vértices, que contiene información significativa para todos los vértices que todavía no forman parte de la solución. En concreto, si $v \notin U$, la posición arista_mín[v] contiene el par <w, g(v, w)>, tal que <v, w> es la arista más pequeña que conecta v con un vértice $w \in U$. Para obviar U en la codificación, convenimos que si arista_mín[v] vale <v, ∞>, entonces $v \in U$. La constante un_vértice_cualquiera puede ser un parámetro formal del universo que encapsule el algoritmo o bien un parámetro de la función.

[18] También se puede devolver un árbol general o el conjunto de aristas del grafo resultante.

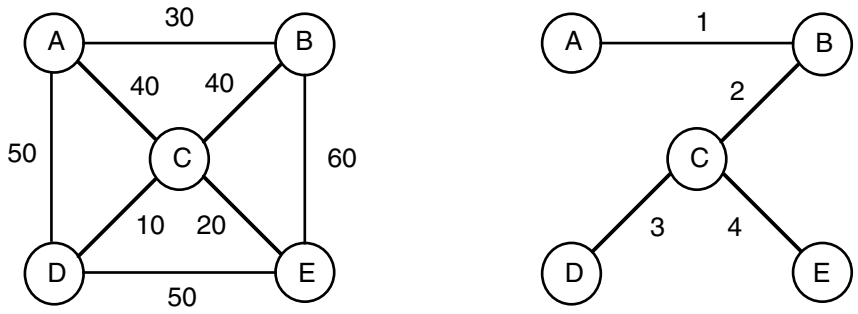


Fig. 6.31: aplicación del algoritmo de Prim: a la izquierda, un grafo no dirigido etiquetado acíclico; a la derecha, árbol resultante de aplicar Prim, tomando A como vértice inicial y numerando las aristas según el orden de inserción en la solución.

El cálculo de la eficiencia temporal de esta nueva versión queda:

- La inicialización es lineal en caso de matriz y cuadrática en caso de (multi)listas. Si se desea, se puede evitar el uso de etiqueta de manera similar a algunos algoritmos ya vistos, para reducir este último a lineal, aunque veremos que esta optimización no influye en el coste asintótico global del algoritmo.
- En el bucle principal hay algunas operaciones de orden constante, que no influyen en el coste total, el añadido de arista en el grafo y dos bucles internos. El añadido será constante usando una matriz y, en el caso peor, lineal usando (multi)listas, pero tampoco esto afectará al coste global. El bucle de selección es incondicionalmente $\Theta(n)$, ya que examina todas las posiciones del vector y, como se ejecuta n-1 veces, lleva a un coste $\Theta(n^2)$, mientras que el bucle de reorganización depende de la representación del grafo: en caso de matriz de adyacencia, calcular los adyacentes es $\Theta(n)$ y el coste total queda $\Theta(n^2)$, mientras que con listas, como siempre, su ejecución a lo largo del algoritmo lleva a un coste $\Theta(a+n)$, que en general es mejor, a pesar de que en este caso no afecta al orden global.

El coste asintótico total queda, en efecto, $\Theta(n^2)$, independientemente de la representación. Por lo que respecta al coste espacial, el espacio adicional empleado es $\Theta(n)$.

```
{P ≅ g es un grafo no dirigido conexo etiquetado no negativamente}
función Prim (g es grafo) devuelve grafo es
var arista_mín es vector [vértices] de vértice_y_etiq
    gres es grafo; primero, mín, v, w son vértice; x es etiq
fvar
        {inicialización del resultado}
    primero := un_vértice_cualquiera
    para todo v dentro de vértice hacer
        arista_mín[v] := <primero, etiqueta(g, primero, v)>
    fpara todo
        {a continuación se aplica el método}
    gres := crea
    hacer n-1 veces
            {I ≅ subgrafo(gres, U)∈ aem(subgrafo(g, U)) ∧
                ∀v: v∉ U: arista_mín[v] = mínimo{etiqueta(g, v, u) / u∈ U},
                    donde U = {u / arista_mín[u] =<u, ∞>}  }
            {primero se selecciona la arista mínima}
        mín := primero     {centinela, pues así arista_mín[min].et vale ∞}
        para todo v dentro de vértice hacer
            <w, x> := arista_mín[v]
            si x < arista_mín[mín].et entonces mín := v  fsi {como mínimo habrá uno}
        fpara todo
            {a continuación se añade a la solución}
        gres := añade(gres, mín, arista_mín[mín].v, arista_mín[mín].et)
            {se añade mín  al conjunto de vértices tratados}
        arista_mín[mín] := <min, ∞>
            {por último, se reorganiza el vector comprobando si la arista mínima de
             los vértices todavía no tratados los conecta a mín}
        para todo <v, x> dentro de adyacentes(g, mín) hacer
            si (arista_mín[v].v = v) ∧ (x < arista_mín[v].et) entonces
                arista_mín[v] := <mín, x>
            fsi
        fpara todo
    fhacer
devuelve gres
{Q ≅ gres∈ aem(g)}
```

Fig. 6.32: implementación eficiente del algoritmo de Prim.

6.5.2 Algoritmo de Kruskal

Como el algoritmo de Prim, el algoritmo ideado por J.B. Kruskal en el año 1956 ("On the shortest spanning subtree of a graph and the traveling salesman problem", en Proceedings American Math. Society, 7, pp. 48-50) se basa en la propiedad de los árboles de expansión minimales: partiendo del árbol vacío, se selecciona a cada paso la arista de menor etiqueta que no provoque ciclo sin requerir ninguna otra condición sobre sus extremos.

El algoritmo se presenta en la fig. 6.33; el invariante usa la operación componentes, que devuelve el conjunto de subgrafos que forman componentes conexos de un grafo dado. Su implementación directa es ineficiente dado que hay que encontrar la arista mínima y comprobar si se forma un ciclo. El estudio del invariante, no obstante, da una visión alternativa del algoritmo que permite implementarlo eficientemente: se considera que, en cada momento, los vértices que están dentro de un componente conexo en la solución forman una clase de equivalencia, y el algoritmo se puede considerar como la fusión continuada de clases hasta obtener un único componente con todos los vértices del grafo.

```
{P ≅ g es un grafo no dirigido conexo etiquetado no negativamente}
función Kruskal (g es grafo) devuelve grafo es
var gres es grafo; u, v son vértice; x es etiq fvar
    gres := crea
    mientras gres no sea conexo hacer
        {I ≅ ∀g': g'∈{f: U x U → E} ∧ g'∈ componentes(gres): g'∈aem(subgrafo(g, U))}
        seleccionar <u, v, x> mínima, todavía no examinada
        si no provoca ciclo entonces gres := añade(gres, u, v, x) fsi
    fmientras
devuelve gres
{Q ≅ gres ∈ aem(g)}
```

Fig. 6.33: presentación del algoritmo de Kruskal.

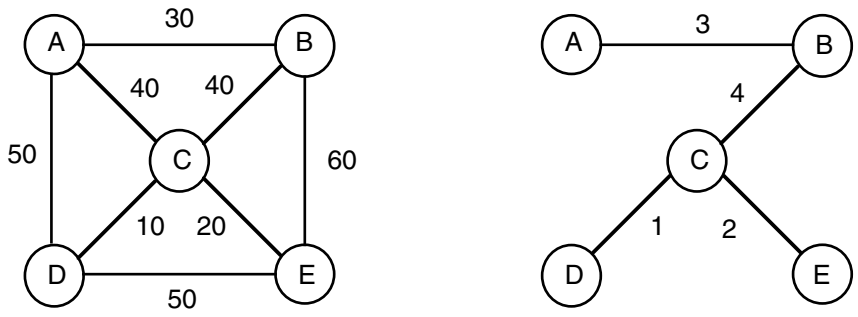


Fig. 6.34: aplicación del algoritmo de Kruskal sobre el grafo de la fig. 6.31.

En el grafo de la figura 6.34, la evolución de las clases es:

$$\{[A], [B], [C], [D], [E]\} \rightarrow \{[A], [B], [C], [D, E]\} \rightarrow \{[A], [B], [C, D, E]\} \rightarrow$$
$$\rightarrow \{[A, B], [C, D, E]\} \rightarrow \{[A, B, C, D, E]\}$$

Así, puede construirse una implementación eficiente del algoritmo de Kruskal, que utiliza la especificación y la implementación arborescente del TAD de las relaciones de equivalencia. El algoritmo resultante (v. fig. 6.35) organiza, además, las aristas dentro de una cola de prioridades implementada con un montículo, de manera que se favorece el proceso de obtención de la arista mínima a cada paso del bucle. El invariante establece las diversas propiedades de las estructuras usando las operaciones de los TAD que intervienen y algunas operaciones de grafos cuya especificación se propone en el ejercicio 6.1.

```
{P ≅ g es un grafo no dirigido conexo etiquetado no negativamente}
función Kruskal (g es grafo) devuelve grafo es
var T es colapr_aristas; gres es grafo; u, v son vértice; x es etiq
    C es releq_vértice; ucomp, vcomp son nat
fvar
    C := RELEQ.crea    {inicialmente cada vértice forma una única clase}
    gres := GRAFO.crea; T := COLAPR.crea
        {se colocan todas las aristas en la cola}
    para todo v dentro de vértice hacer
        para todo <u, x> dentro de adyacentes(g, v) hacer T := inserta(T, v, u, x) fpara todo
    fpara todo
    mientras RELEQ.cuántos?(C) > 1 hacer
          {I ≅ ∀g': g'∈{f: U x U → E} ∧ g'∈ componentes(gres): g'∈aem(subgrafo(g, U)) ∧
              ∀<u, v>: <u, v>∈A_gres: etiqueta(g, u, v) ≤ mínimo(T).et ∧
              ∀u,v: u,v∈vértice: clase(C, u) = clase(C, v) ⇔ conectados(gres, u, v) }
            {se obtiene y se elimina la arista mínima de la cola}
        <u, v, x> := COLAPR.menor(T); T := COLAPR.borra(T)
              {a continuación, si la arista no provoca ciclo se añade a la solución y se
               fusionan las clases correspondientes}
        ucomp := RELEQ.clase(C, u); vcomp := RELEQ.clase(C, v)
        si ucomp ≠ vcomp entonces
            C := RELEQ.fusiona(C, ucomp, vcomp)
            gres := añade(gres, u, v, x)
        fsi
    fmientras
devuelve gres
{Q ≅ gres∈ aem(g)}
```

Fig. 6.35: implementación eficiente del algoritmo de Kruskal.

El coste del algoritmo se puede calcular de la siguiente manera:

- La creación de la relación de equivalencia es $\Theta(n)$, y la del grafo resultado es $\Theta(n^2)$ usando una representación por matriz y $\Theta(n)$ usando listas de adyacencia.
- Las a inserciones consecutivas de aristas en la cola prioritaria quedan $\Theta(a \log a)$. Como a es un valor entre n-1 y n(n-1)/2, sustituyendo por estos valores en $\Theta(a \log a)$ obtenemos $\Theta(a \log n)$ en ambos casos, habida cuenta que $\Theta(a \log n^2) = \Theta(2a \log n)$. El coste se puede reducir hasta $\Theta(a)$ usando la función organiza de los pilones para convertir un vector con todas las aristas en la cola prioritaria correspondiente. También debe considerarse que la obtención de las aristas es $\Theta(n^2)$ con matriz y $\Theta(a)$ con listas.
- Como mucho, hay a consultas y supresiones de la arista mínima; el coste de la consulta es constante y el de la supresión logarítmico y, por ello, este paso queda $\Theta(a \log n)$ en el caso peor. Ahora bien, si la configuración del grafo conlleva un número pequeño de iteraciones (siempre n-1 como mínimo, eso sí) el coste es realmente menor, hasta $\Theta(n \log n)$; si la cola prioritaria se crea usando organiza, este hecho puede ser significativo. Notemos que el caso peor es aquél en que la arista de etiqueta mayor del grafo es necesaria en todo árbol de expansión minimal asociado al grafo, mientras que el caso mejor es aquél en que las n-1 aristas más pequeñas forman un grafo acíclico.
- El coste de las operaciones sobre las relaciones ha sido estudiado en la sección 5.4: averiguar cuántas clases hay es constante mientras que fusiona se ejecuta n-1 veces y clase entre 2(n-1) y 2a veces, por lo que el coste total al término del algoritmo es $\Theta(n)$ en el caso mejor y $\Theta(a)$ en el caso peor, usando la técnica de compresión de caminos.
- Las n-1 inserciones de aristas quedan $\Theta(n)$ con matriz y $\Theta(n^2)$ con listas. En este algoritmo, al contrario que en el anterior, este hecho es importante, por lo que debe evitarse el uso de una implementación encadenada del grafo resultado. Otra alternativa es eliminar la comprobación de existencia de la arista en la función añade, pues la mecánica del algoritmo de Kruskal garantiza que no habrán inserciones repetidas de aristas; como resultado, la función quedaría constante siempre y cuando las listas no se ordenaran a cada inserción sino en un paso posterior una vez obtenidas todas las aristas del resultado[19] .

Así, y teniendo en cuenta el razonamiento sobre la implementación del grafo resultado, el coste total del algoritmo es: en el caso peor (se examinan todas las aristas), $\Theta(a \log n)$ con listas de adyacencia y $\Theta(n^2+a \log n)$ con matriz; en el caso mejor (n-1 aristas examinadas), $\Theta(a+n \log n)$ con listas y $\Theta(n^2)$ con matriz. La comparación de estas magnitudes con $\Theta(n^2)$ indica la conveniencia de usar Prim o Kruskal para obtener el árbol de expansión minimal (a tal efecto, v. ejercicio 6.23). Al tomar esta decisión, también puede influir que Kruskal necesita más memoria auxiliar a causa del montículo y de la relación, incluso asintóticamente hablando, porque el montículo exige un espacio $\Theta(a)$, por $\Theta(n)$ del vector auxiliar de Prim.

[19] Recordemos que la ordenación de las listas de adyacencia es necesaria para que la implementación sea correcta con respecto a la especificación.

Ejercicios

Nota preliminar: en los ejercicios de este capítulo, debe calcularse el coste de todos los algoritmos que se pidan.

6.1 Escribir la especificación de los grafos: **a)** dirigidos no etiquetados; **b)** no dirigidos y etiquetados; **c)** no dirigidos y no etiquetados. Enriquecer estos universos y el de los grafos dirigidos y etiquetados con las operaciones:

caminos: grafo vértice vértice → cjt_sec_vértices, devuelve el conjunto de caminos simples entre los dos vértices dados; cada camino se representa por una secuencia de vértices.

antecesores, descendientes: grafo vértice → cjt_vértices, devuelve todos los vértices que son antecesores o descendientes del vértice dado, respectivamente.

ciclo: grafo vértice → bool, responde si el vértice dado forma parte de algún ciclo.

conectados: grafo vértice vértice → bool, indica si hay algún camino entre los vértices dados.

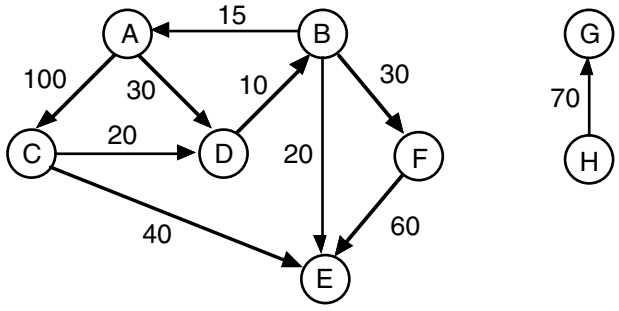
componentes: grafo → cjt_grafos, devuelve un conjunto con todos los componentes conexos del grafo; cada componente se representa mediante un grafo.

conexo, acíclico: grafo → bool, indican si el grafo es conexo o acíclico, respectivamente.

subgrafo: grafo cjt_vértices → grafo, tal que subgrafo(g, U) devuelve el grafo que tiene como nodos los elementos de U y como arcos aquéllos de g que conectan nodos de U.

6.2 Hacer una comparación exhaustiva de las diversas representaciones de grafos en lo que se refiere a espacio y tiempo de ejecución de las operaciones del TAD, estudiando para qué operaciones y características son más adecuadas.

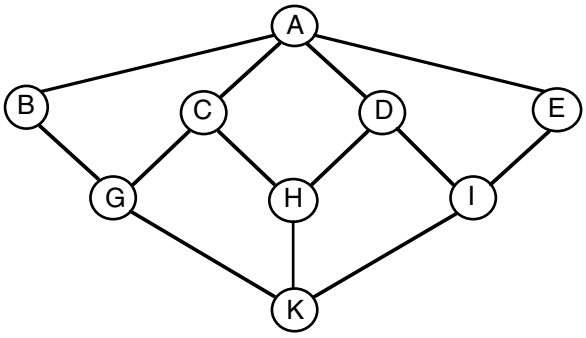
6.3 Representar de todas las formas posibles el grafo:



6.4 Diseñar un par de algoritmos para transformar un grafo representado mediante matriz de adyacencia en una representación mediante listas de adyacencia y viceversa.

6.5 En un ordenador cualquiera, los números enteros y los punteros se representan con 32 bits y los booleanos con un único bit. Calcular el número mínimo de aristas que debe tener un grafo dirigido no etiquetado de 100 vértices, suponiendo que los vértices se identifican mediante enteros, con el objetivo de que su representación mediante matriz de adyacencia ocupe menos bits que mediante listas de adyacencia. Repetir el ejercicio suponiendo la presencia de enteros como etiquetas.

6.6 Suponiendo que los sucesores de un nodo dado siempre se obtienen en orden alfabético, recorrer el grafo siguiente en profundidad y en anchura empezando por A:



6.7 Usar algún algoritmo conocido para decidir si un grafo no dirigido es conexo. Modificar el algoritmo para que, en caso de que no lo sea, encuentre todos los componentes conexos.

6.8 Escribir un algoritmo a partir de algún recorrido para calcular la clausura transitiva (ing., transitive clousure) de un grafo no dirigido, es decir, que determine para todo par de nodos si están conectados o no. Comparar con el resultado del ejercicio 6.17.

6.9 Escribir un algoritmo que, dado un grafo dirigido y dos vértices de este grafo, escriba todos los caminos simples de un vértice al otro.

6.10 Escribir un algoritmo para enumerar todos los ciclos elementales dentro de un grafo no dirigido. ¿Cuál es el máximo número de ciclos elementales que se pueden encontrar?

6.11 Modificar el algoritmo de recorrido en anchura para obtener un programa que detecte si un grafo no dirigido presenta algún ciclo.

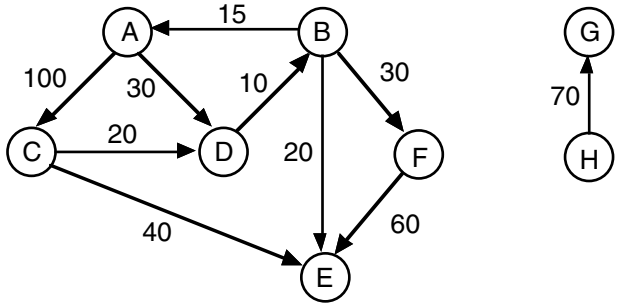
6.12 Formular el algoritmo de ordenación topológica a partir del algoritmo de recorrido en profundidad.

6.13 Una expresión aritmética puede representarse con un grafo dirigido acíclico. Indicar exactamente cómo, y qué algoritmo debe aplicarse para evaluarla. Escribir un algoritmo que transforme una expresión representada con un árbol en una expresión representada con un

grafo dirigido acíclico.

6.14 Los árboles de expansión no necesariamente minimales son un concepto útil para analizar redes eléctricas y obtener un conjunto independiente de ecuaciones. No obstante, en lugar de aplicar los algoritmos presentados en la sección 6.5, que tienen un coste elevado, se pueden adaptar los algoritmos de recorrido de grafos presentados en la sección 6.3, que permiten construir árboles de expansión con las aristas usadas para visitar los nodos del grafo, y entonces añadir consecutivamente el resto de aristas para formar el conjunto de ecuaciones (para más detalles, v. [HoS94, pp. 335-337] y [Knu68, pp. 393-398]). Modificar alguno de estos recorridos para utilizarlo en este contexto.

6.15 **a)** Encontrar el coste de los caminos mínimos entre todo par de nodos del grafo:



i) aplicando reiteradamente el algoritmo de Dijkstra, y **ii)** aplicando el algoritmo de Floyd. **b)** Ahora, además, calcular cuáles son los nodos que forman los caminos mínimos entre toda pareja de nodos.

6.16 Razonar por qué el algoritmo de Dijkstra no siempre funciona para grafos con etiquetas no necesariamente positivas. Mostrar un grafo de tres nodos con alguna arista negativa para el cual no funcione el algoritmo.

6.17 Modificar el algoritmo de Floyd para que calcule la clausura transitiva de un grafo. El resultado es el denominado algoritmo de Warshall que, curiosamente, es anterior al de Floyd (v. "A theorem on boolean matrices", S. Warshall, Journal of the ACM, 9(1), pp.11-12, 1962). Comparar con el resultado del ejercicio 6.8.

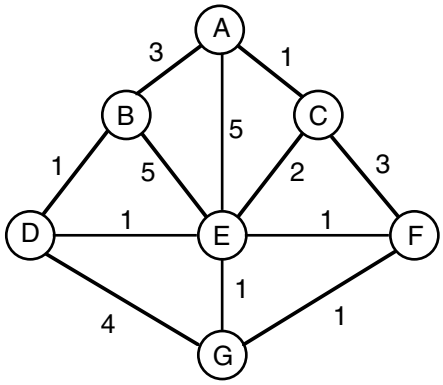
6.18 Modificar los algoritmos de Dijkstra y Floyd para que también devuelvan una estructura que almacene de forma compacta los caminos minimales. Diseñar e implementar algoritmos recursivos que recuperen los caminos minimales a partir de la información obtenida.

6.19 Para todas las situaciones que a continuación se enumeran, decir si afectan o no al funcionamiento normal del algoritmo de Dijkstra; en el caso de que lo hagan, indicar si existe alguna manera de solucionar el problema: **a)** el grafo presenta algún ciclo; **b)** el grafo es no

dirigido; **c)** las aristas pueden ser negativas; **d)** puede haber aristas de un nodo a sí mismo; **e)** pueden haber varias aristas de un nodo a otro; **f)** hay nodos inaccesibles desde el nodo inicial; **g)** hay varios nodos unidos con el nodo inicial por caminos igual de cortos.

6.20 Dado un grafo dirigido determinar, en función de su representación y de la relación entre el número de aristas y el número de nodos, si es mejor el algoritmo de Floyd o el algoritmo de Dijkstra reiterado para determinar la conexión entre toda pareja de nodos. En el segundo caso, especificar claramente cómo hay que modificar el algoritmo para adaptarlo al problema. Decir si hay alguna situación concreta que aconseje aplicar algún otro algoritmo conocido sobre el grafo.

6.21 Dado el grafo siguiente:



indicar, a ojo de buen cubero, todos los posibles árboles de expansión minimales asociados. A continuación, aplicarles los algoritmos de Kruskal y de Prim, estudiando todas las resoluciones posibles de las indeterminaciones que presente. Comparar los resultados.

6.22 El coste espacial adicional del algoritmo de Prim de la fig. 6.20 es $\Theta(n)$ asintóticamente y no se puede mejorar; ahora bien, las dos estructuras empleadas están fuertemente interrelacionadas, como muestra claramente el invariante, y se pueden fusionar, dado que dentro del conjunto sólo hay vértices de cero predecesores no tratados. Codificar esta nueva versión.

6.23 Estudiar la posibilidad de organizar el vector arista_mín del algoritmo de Prim como una cola prioritaria. ¿Qué coste queda? ¿Influye en este coste la configuración del grafo? (Esta implementación fue propuesta por D.B. Johnson el año 1975 en "Priority queues with update and finding minimum spanning trees", Information Processing Letters, 4, pp. 53-57.)

6.24 Razonar cómo afectan la implementación del grafo y la relación entre el número de nodos y aristas en la elección entre Prim y Kruskal.

6.25 Para todas las situaciones que se enumeran a continuación, decir si afectan o no al funcionamiento normal de los algoritmos de Prim y Kruskal; en caso de que lo hagan, indicar si hay alguna forma posible de solucionar el problema: **a)** el grafo presenta algún ciclo; **b)** el grafo es dirigido; **c)** las aristas pueden ser negativas; **d)** puede haber aristas reflexivas; **e)** puede haber varias aristas de un nodo a otro; **f)** hay nodos inaccesibles desde el nodo inicial; **g)** hay varios nodos unidos con el nodo inicial con una arista igual de corta.

6.26 Diseñar un algoritmo que decida si un grafo no dirigido dado contiene algún subgrafo completo de k vértices, usando la signatura que más convenga.

6.27 La región de Aicenev, al sur de Balquistán, es famosa en el mundo entero por su sistema de comunicaciones basado en canales que se ramifican. Todos los canales son navegables y todas las localidades de la región están situadas cerca de algún canal.

a) Una vez al año, los alcaldes de la región se reunen en la localidad de San Macros desplazándose cada uno en su propio vaporetto oficial. Proponer a los alcaldes un algoritmo que las permita llegar a San Macros de forma que entre todos ellos hayan recorrido la mínima distancia.

b) A consecuencia de la crisis energética, los alcaldes han decidido compartir los vaporettos de manera que si, durante su trayecto, un vaporetto pasa por una ciudad donde hay uno o más alcaldes, los recoge, formándose entonces grupos que se dirigen a San Macros. Discutir la validez de la solución anterior. Si es necesario, proponer otra que reduzca la distancia recorrida por los vaporettos (pero no necesariamente por los alcaldes).

6.28 Una empresa dispone de un conjunto de ordenadores distribuidos en un ámbito geográfico. Varias parejas de estos ordenadores se podrían conectar físicamente mediante una línea cualquiera de comunicación (bidireccional). La empresa hace un estudio previo del coste de transimisión por unidad de información que tendría cada conexión en función de algunos parámetros (longitud de la línea, calidad de la señal, etc.) y, a partir de este resultado, quiere calcular qué conexiones tiene que usar para que se cumpla que: **a)** dos ordenadores predeterminados del conjunto estén comunicados entre sí con el mínimo coste de transmisión por unidad de información posible y **b)** todos los ordenadores estén comunicados, de manera que el coste total de la transmisión por unidad de información sea mínimo, cumpliendo la condición del punto anterior. Proponer un algoritmo que resuelva el problema.

6.29 Una multinacional tiene agencias en numerosas ciudades dispersas por todo el planeta. Suponiendo que se conoce el coste de un minuto de conversación telefónica entre todo par de ciudades, se quiere encontrar la forma de decidir a qué agencias debe llamarse desde toda agencia para que un mensaje originario de cualquiera de ellas llegue al destinatario con un coste total mínimo. Pensar cuál es la información mínima imprescindible que debe almacenarse en cada agencia.

6.30 a) El ayuntamiento de Villatortas de Arriba proyecta construir una red metropolitana. Previamente, el consistorio decide la ubicación de cada estación y calcula la distancia entre todo par de ellas, de manera que tan sólo queda determinar qué tramos (es decir, qué conexiones entre estaciones) se construyen. La política elegida consiste en reducir la longitud total de la red con la condición de que todas las estaciones queden conectadas entre sí (es decir, que desde cualquier estación se pueda llegar a cualquier otra). El resultado puede implicar la existencia de diferentes líneas de metro y, en este caso, existen estaciones que pertenecen a más de una línea y que permiten hacer transbordos (es decir, cambios de línea). Asociar esta situación a un modelo y a un algoritmo conocido y explicar cómo debería interpretarse la salida de dicho algoritmo para encontrar una posible configuración de la red (en el caso general, habrá diversas alternativas; devolver una cualquiera).

b) Al construir la red metropolitana, el ayuntamiento decide no seguir los resultados del estudio anterior sino que, movido por intereses oscuros, prefiere aceptar la propuesta de la coalición en el poder. A continuación, el consistorio quiere disponer de aquellas máquinas que, dadas dos estaciones, encuentran la manera de ir de una a otra reduciendo la distancia recorrida (suponer que un transbordo entre líneas tiene distancia 0). Justificar qué algoritmo debe implementarse en una máquina instalada en la estación E en los supuestos siguientes:

i) Que pueda pedirse el trayecto más corto entre cualquier par de estaciones.

ii) Que sólo pueda pedirse el trayecto más corto desde E a cualquier otra estación.

(En caso que haya más de una alternativa, devolver una cualquiera.)

c) Después de un estudio de los técnicos municipales, se considera que un criterio mejor para programar las máquinas consiste en reducir el número de estaciones de paso para ir de una estación a otra (suponer que un transbordo entre líneas no cuenta ningún paso). Explicar qué modelo y qué algoritmo se necesita para programar las máquinas. (En caso de que haya más de una alternativa, devolver una cualquiera.)

d) En las siguientes elecciones municipales, el principal partido de la oposición basa su campaña en sustituir estas máquinas por otras que, dadas dos estaciones, encuentran la manera de ir de una a otra con el número mínimo de transbordos. ¿Es posible? En caso afirmativo, decir qué modelo y qué algoritmo se necesita. ¿Qué debe modificarse en la solución si, en caso de haber diversas alternativas con el mínimo número de transbordos, se pidiera la que reduzca el número de estaciones de paso, siendo dicho cálculo lo más eficiente posible?

6.31 En este ejercicio nos centramos en la especificación de los diferentes recorridos sobre grafos estudiados en la sección 6.3. Considerando que hay muchos recorridos concretos asociados a una estrategia particular y que, además, la especificación de la obtención de uno (o todos) de estos recorridos, dentro del marco de la semántica inicial, resultaría prácticamente en una implementación por ecuaciones (debido al conocido problema de la sobreespecificación que presenta esta semántica), se ha optado por introducir diferentes predicados que comprueben si una lista con punto de interés es un recorrido válido para un grafo dirigido no etiquetado según cada estrategia.

a) Especificar la función están_todos: lista_vértices → bool que comprueba si una lista contiene todos los vértices del grafo, lo que es un requisito indispensable para que la lista se pueda considerar como un recorrido válido según cualquier estrategia. Definir los parámetros formales necesarios sobre los vértices que permitan obtener todos los valores del tipo. Si es necesario, enriquecer las listas con alguna operación.

b) Sea g un grafo dirigido no etiquetado y sea l una lista de vértices. Se quiere especificar la operación es_recorrido_profundidad: lista_vértices grafo → bool; l será un recorrido en profundidad de g si, para todo vértice v de la lista, el recorrido avanza por alguno de los caminos que salen de él y llega tan lejos como puede (es decir, hasta un vértice ya visitado o hasta un vértice sin sucesores). Los caminos se definen como una secuencia de vértices. Especificar las operaciones:

- todos_caminos_no_cíclicos: grafo lista_vértices cjt_vértices → cjt_secs_vértices, que genera el conjunto de caminos no cíclicos que salen de alguno de los vértices dados a la lista; el conjunto de vértices representa los vértices por los que ya ha pasado el camino y es necesario precisamente para controlar si es cíclico o no. Si es necesario, enriquecer las listas con alguna operación.
- sigue_alguno: cjt_secs_vértices lista_vértices → bool, que comprueba si hay algún camino de los generados anteriormente que se solape total o parcialmente con los vértices que hay en la lista, a partir de la posición actual; en este caso, se puede afirmar que efectivamente se ha seguido este camino. Si el solapamiento no es total, es indispensable que los vértices del camino que no se solapan ya hayan sido visitados previamente, porque este es el único motivo que justifica la interrupción en el recorrido de un camino.
- prof_pro: lista_vértices grafo → bool, que comprueba que todos los vértices de la lista cumplen la condición del recorrido en profundidad.

De esta manera, la especificación de la operación es_recorrido_profundidad? queda:

es_recorrido_profundidad(l, g) = están_todos(l) ∧ prof_pro(principio(l), g)

c) Sea g un grafo dirigido no etiquetado y sea l una lista de vértices. Se quiere especificar la operación es_recorrido_anchura: lista_vértices grafo → bool; l será un recorrido en anchura de g si, para todo vértice v de la lista, el recorrido explora inmediatamente todos sus sucesores todavía no visitados. Especificar las operaciones:

- subsec: sec_vértices lista_vértices → sec_vértices, que genera la subsecuencia del recorrido que tiene como primer elemento el primer no sucesor de v que aparece detrás de v. La lista de entrada representa los vértices sucesores y la secuencia contiene, inicialmente, los nodos que hay a partir del elemento actual, sin incluirlo.
- no_hay_ninguno: sec_vértices lista_vértices → bool, que comprueba que, en la subsecuencia obtenida con subsec, no hay ningún sucesor de v. Es necesario pasar la lista de vértices sucesores.
- anchura: lista_vértices grafo → bool, que comprueba que todos los vértices de la lista cumplen la condición de recorrido en anchura.

Así, la especificación de la operación es_recorrido_anchura? queda:

es_recorrido_anchura(l, g) = están_todos(l) $\wedge$ anchura(principio(l), g)

d) Sea g un grafo dirigido no etiquetado y sea l una lista de vértices. Se quiere especificar la operación es_ordenación_topológica?: lista_vértices grafo $\rightarrow$ bool; l será un recorrido en ordenación topológica de g, si todo vertice v aparece en la lista después de todos sus antecesores. Especificar la operación ord_top: lista_vértices grafo $\rightarrow$ bool, que comprueba que, para cada arista del grafo, se cumple la propiedad dada. Así, la especificación de la operación es_ordenación_topológica? queda:

es_ordenación_topológica(l, g) = están_todos(l) $\wedge$ ord_top(principio(l), g)

En todos los apartados, detallar las instancias efectuadas y también cuáles son los parámetros formales de la especificación.

6.32 En este ejercicio se quiere especificar la obtención de caminos mínimos en grafos.

a) Sea g un grafo dirigido y etiquetado no negativamente y sea v un vértice. Especificar la operación camino_mínimo: grafo vértice $\rightarrow$ tabla_vértices_y_etiqs; camino_mínimo(g, v) devuelve una tabla T, tal que consulta(T, w) es igual al coste del camino mínimo entre v y w.

b) Sea g un grafo dirigido y etiquetado no negativamente. Especificar la operación caminos_mínimos: grafo $\rightarrow$ tabla_parejas_vértices_y_etiqs; caminos_mínimos(g) devuelve una tabla T, tal que consulta(T, <v, w>) es igual al coste del camino mínimo entre v y w.

En todos los apartados, detallar las instancias efectuadas y también cuáles son los parámetros formales de la especificación. Si es conveniente, usar alguna de las operaciones especificadas en el ejercicio 6.1.

6.33 En este ejercicio se quiere especificar el concepto de árbol de expansión minimal asociado a un grafo siguiendo varios pasos.

a) Sea g un grafo no dirigido y etiquetado no negativamente. Especificar la operación no_cíclicos: grafo $\rightarrow$ cjt_grafos, tal que no_cíclicos(g) devuelve el conjunto de subgrafos de g que no presentan ningún ciclo.

b) Sea C un conjunto de grafos no dirigidos, acíclicos y etiquetados no negativamente. Especificar la operación maximales: cjt_grafos $\rightarrow$ cjt_grafos tal que maximales(C) devuelve un conjunto D de grafos conexos, D $\subseteq$ C. Notar que, por la propiedad de los árboles libres, o bien D es el conjunto vacío (si no había ningún grafo conexo en C), o bien todos sus elementos tienen exactamente n - 1 aristas, siendo n el número de vértices de los grafos.

c) Sea g un grafo no dirigido y etiquetado no negativamente. Especificar la operación ae: grafo $\rightarrow$ cjt_grafos tal que ae(g) devuelve el conjunto de grafos que son árboles de expansión para g.

d) Sea C un conjunto de grafos, resultado de un cálculo de ae. Especificar la operación minimales: cjt_grafos $\rightarrow$ cjt_cjt_grafos tal que minimales(S) devuelve un conjunto D de

conjuntos de grafos que son minimales (respecto la suma de las etiquetas de sus aristas) dentro del conjunto C, D $\subseteq$ C.

e) Finalmente, sea g un grafo no dirigido y etiquetado no negativamente. Especificar la operación aem: grafo $\rightarrow$ cjt_cjt_aristas que devuelve el conjunto de árboles de expansión minimales asociados al grafo.

En todos los apartados, detallar las instancias efectuadas y también cuáles son los parámetros formales de la especificación.