6 GRAFOS

Hemos considerado los árboles como una generalización del concepto de lista porque permiten que un elemento tenga más de un sucesor. Los grafos aparecen como una extensión del concepto de árbol, ya que en este nuevo tipo de estructuras cada elemento puede tener, además de más de un sucesor, varios elementos predecesores.

Esta propiedad hace a los grafos las estructuras más adecuadas para representar situaciones donde la relación entre los elementos es completamente arbitraria, como pueden ser mapas de carreteras, sistemas de telecomunicaciones, circuitos impresos o redes de ordenadores. Aunque hay estructuras más complejas que los grafos, no las veremos en este curso.

Los grafos se pueden clasificar en diferentes tipos dependiendo de cómo se defina la relación entre los elementos: podemos encontrar grafos dirigidos o no dirigidos y etiquetados o no etiquetados. También se pueden combinar ambas categorías.

6.1 Conceptos previos y terminología.

Formalmente, un grafo *G* consiste en dos conjuntos finitos *N* y *A*. *N* es el conjunto de elementos del grafo, también denominados *vértices* o *nodos*. *A* es el conjunto de *arcos*, que son las conexiones que se encargan de relacionar los nodos para formar el grafo. Los arcos también son llamados *aristas* o *líneas*.

Los nodos suelen usarse para representar objetos y los arcos para representar la relación entre ellos. Por ejemplo, los nodos pueden representar ciudades y los arcos la existencia de carreteras que las comunican.

Cada arco queda definido por un par de elementos n_1, $n_2 \in N$ a los que conecta. Aunque habitualmente los elementos son distintos, permitiremos que sean el mismo nodo ($n_1 = n_2$). Representaremos gráficamente un arco como una línea que une los dos nodos asociados.

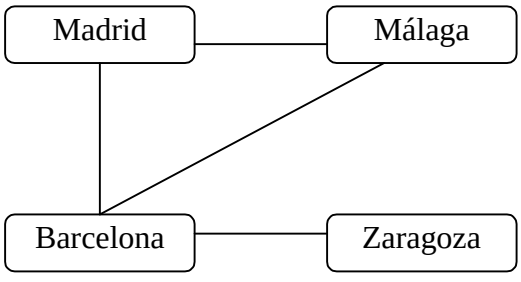


Figura 6. 1. Grafo representativo de la conexión aérea de ciudades.

Si queremos representar mediante un grafo la red de vuelos de una compañía aérea entre diferentes ciudades, tendríamos el siguiente grafo $G = \{N, A\}$; N = {Málaga, Zaragoza, Madrid, Barcelona}; A = {(Madrid, Málaga), (Madrid, Barcelona), (Málaga, Barcelona), (Zaragoza, Barcelona)} la representación gráfica sería la de la figura 6.1.

Se dice que dos nodos son *adyacentes* o *vecinos* si hay un arco que los conecta. Los nodos adyacentes pueden ser representados por pares (a, b).

Un *camino* es una secuencia de nodos $n_1, n_2, ..., n_m$ tal que $\forall i, 1 \leq i \leq (m-1)$, cada par de nodos (n_i, n_{i+1}) son adyacentes. Se dice que un camino es *simple* si cada uno de sus nodos, excepto tal vez el primero y el último, aparece sólo una vez en la secuencia.

La *longitud* de un camino es el número de arcos de ese camino. Se puede considerar como caso especial un nodo por sí mismo como un camino de longitud 0.

Un *ciclo* es un camino simple en el que el primer y último nodos son el mismo ($n_1 = n_m$). Si un camino desde un nodo a él mismo no contiene otros nodos entonces decimos que es un *ciclo degenerado*.

Un grafo sin ciclos se dice *acíclico*.

Si en un grafo $G = \{N,A\}$, N está formado por dos o más subconjuntos disjuntos de nodos (no hay arcos que conecten nodos de un subconjunto con nodos de otro subconjunto) entonces se dice que el grafo es *desconectado* o *inconexo*, en otro caso se dice que es *conectado* o *conexo*.

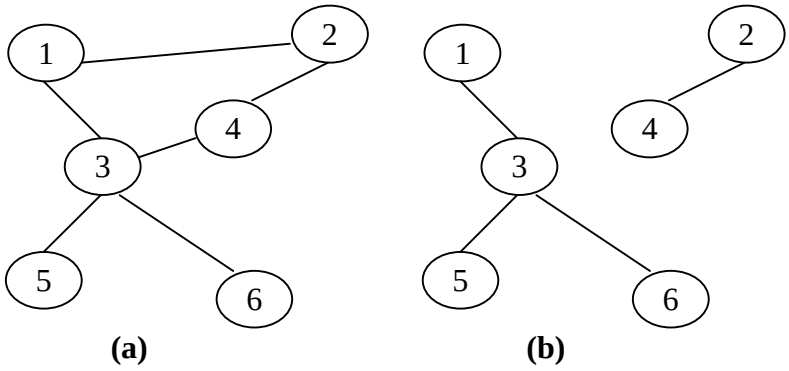


Figura 6. 2. Grafos (a) conexo y (b) inconexo.

6.1.1 Subgrafos.

Sea $G = (N, A)$ un grafo con un conjunto de nodos N y un conjunto de arcos A. Un *subgrafo* de G es un grafo $G' = (N', A')$ tal que:

1. N' es un subconjunto de N.
2. todos los elementos que aparecen en arcos de A' pertenecen a N'.

Si, además, A' consta de todos los arcos (n, m) de A, tal que n y m están en N', entonces G' es un *subgrafo inducido* de G.

6.1.2 Grafos dirigidos y no dirigidos.

Hasta ahora hemos supuesto que un arco conecta dos nodos en ambos sentidos igualmente. Un *grafo dirigido* es aquel en el que los arcos tienen un único sentido. En este caso, un arco se dirige desde el nodo origen hasta el nodo destino. Se dice que el nodo origen *precede* al nodo destino, y que éste *sucede* al origen. Los arcos de un grafo dirigido se representan gráficamente con flechas.

Un grafo no dirigido es un grafo donde los arcos conectan a los nodos en ambos sentidos.

Un grafo dirigido se podría usar para representar bloques de un programa mediante los nodos y la transferencia del flujo de control mediante los arcos. Un grafo que representara el sentido del tráfico entre diferentes plazas podría ser:

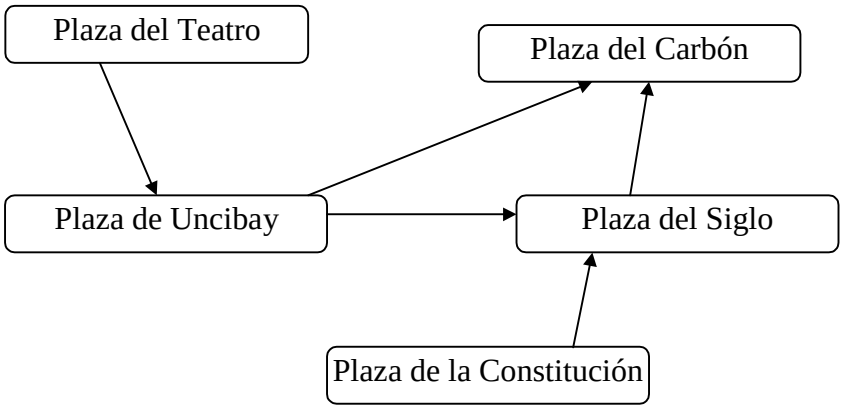


Figura 6. 3. Grafo representativo del sentido del tráfico.

Un nodo *N* se dice alcanzable desde un nodo *M* si y sólo si existe un camino desde *M* hasta *N*. Más formalmente, un nodo *W* se dice alcanzable desde un nodo *M* si:

(1) *N* y *M* son el mismo nodo, o

(2) *N* es alcanzable desde algún nodo que sea sucesor de *M*.

Para cada nodo de un grafo existe un conjunto de nodos alcanzables desde ese nodo, denominado *conjunto alcanzable*.

Un nodo *N* se dice directamente alcanzable desde un nodo *M* si y sólo si son adyacentes y *N* es el sucesor de *M*.

Un grafo conexo acíclico no dirigido es un árbol libre. Un árbol libre puede convertirse en un árbol general si se elige cualquier nodo deseado como raíz y se orienta el cada arco desde ella.

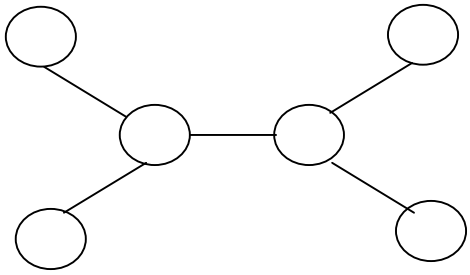


Figura 6. 4. Árbol libre

Los árboles libres tienen dos propiedades interesantes:

1. Todo árbol libre con n nodos tiene exactamente n-1 arcos.
2. Si se agrega cualquier arco a un árbol libre, se genera un ciclo.

6.1.3 Grafos etiquetados y no etiquetados.

En ciertos casos es necesario asociar información a los arcos del grafo. Esto se puede lograr mediante una etiqueta que contenga cualquier información útil relativa al arco, como el nombre, peso, coste o un valor de cualquier tipo de datos dado. En este caso hablamos de *grafos etiquetados*. Esta etiqueta podría significar el tiempo que se tarda el vuelo entre dos ciudades o indicar cuáles son los parámetros de entrada y de salida en la llamada a un subprograma.

Un grafo no etiquetado es un grafo donde los arcos no tienen etiquetas.

En el caso del grafo que representa el sentido del tráfico se pueden etiquetar los arcos con el nombre de las calles.

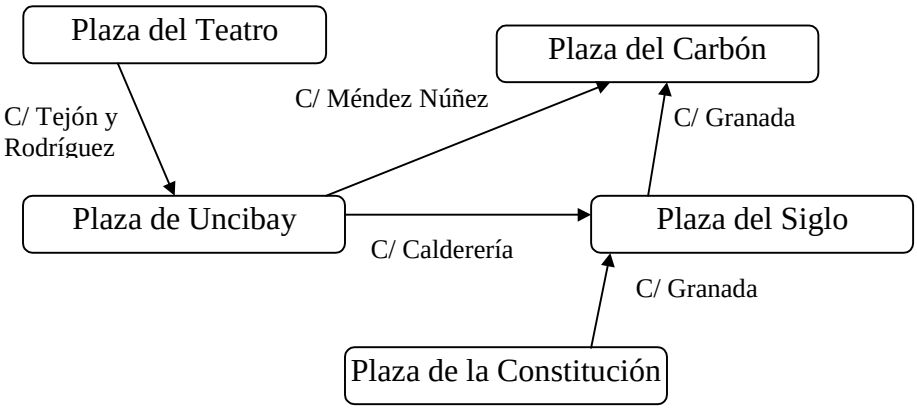


Figura 6. 5. Sentido de tráfico representado por un grafo etiquetado

6.2 Especificaciones de los distintos TADs grafo.

Si consideramos grafos no dirigidos, la especificación del T.A.D. debe incluir:

1. constructores:

- **crear un grafo vacío,** operación constante que devuelve un grafo vacío (sin nodos ni arcos)**.**
- **añadir un nodo**, que devuelve un grafo con todos los nodos y arcos del grafo inicial junto con el nuevo nodo si no estaba ya o el grafo original si el nodo ya estaba incluido en el grafo y
- **añadir una arista**, que devuelve un grafo con todos los nodos y arcos del grafo inicial y un arco nuevo entre dos nodos del grafo o bien el grafo original si ese arco ya existía. Esta operación tiene como precondición que ambos nodos pertenezcan al grafo.
- **devolver un grafo sin un determinado nodo**, que devuelve un grafo con los nodos del grafo inicial excepto el que se borra y sin los arcos que contenían al nodo borrado o el grafo inicial si el nodo no pertenecía al grafo.
- **o sin una determinada arista**, que devuelve el grafo que resulta de eliminar la arista indicada si existe o el grafo original si no existe.

2. funciones selectoras:

- **comprobar si un grafo es vacío**,
- **comprobar si un nodo pertenece al grafo**, y
- **si dos nodos son adyacentes**.

tipo Grafo
dominios Grafo, Elemento, BOOLEAN
generadores
Crear : ⟶ Grafo
Añadir_Nodo : Grafo × Elemento ⟶ Grafo
Añadir_Arista : Grafo × Elemento × Elemento —/→ Grafo
constructores
Borrar_Nodo : Grafo × Elemento ⟶ Grafo
Borrar_Arista : Grafo × Elemento × Elemento ⟶ Grafo
selectores
Es_Vacio? : Grafo ⟶ BOOLEAN
Contiene? : Grafo × Elemento ⟶ BOOLEAN
Son_Adyacentes?: Grafo × Elemento × Elemento ⟶ BOOLEAN
Nodo: Grafo ⟶ Elemento
Arista: Grafo ⟶ Elemento × Elemento
auxiliares
Hay_Arista? : Grafo ⟶ BOOLEAN
precondiciones G: Grafo n, m: Elemento
pre: Añadir_Arista(G, n, m) = Contiene?(G, n) AND Contiene?(G, m)
pre: Nodo(G) = NOT Es_Vacio?(G)
pre: Arista(G) = Hay_Arista?(G)
ecuaciones G:Grafo; n,m,p,q:Elemento;
Añadir_Nodo(Añadir_Nodo(G, n), m) == SI (n = m) ENTONCES
Añadir_Nodo(G, n);
SI NO

```
                                                    Añadir_Nodo(Añadir_Nodo(G, m), n)
    Añadir_Arista(G, m, n) ==    Añadir_Arista(G, n, m)
    Añadir_Arista(Añadir_Arista(G, m, n), p, q) ==
                                        SI (p = m) AND (q = n) ENTONCES
                                              Añadir_Arista(G, m, n)
                                        SI NO
                                             Añadir_Arista(Añadir_Arista(G,p, q), m, n)
    Añadir_Nodo(Añadir_Arista(G, m, n), p) == Añadir_Arista(Añadir_Nodo(G,p), m, n)
    Borrar_Nodo(Crear, m) == Crear
    Borrar_Nodo(Añadir_Nodo(G, n), m) ==    SI (n = m) ENTONCES
                                                  Borrar_Nodo(G, m)
                                            SI NO
                                                Añadir_Nodo(Borrar_Nodo(G, m), n)
    Borrar_Nodo(Añadir_Arista(G, p, q), m) == SI (m = p) OR (m = q) ENTONCES
                                                  Borrar_Nodo(G, m)
                                            SI NO
                                              Añadir_Arista(Borrar_Nodo(G, m), p, q)
    Borrar_Arista(Crear, n, m) == Crear
    Borrar_Arista(Añadir_Nodo(G, p), n, m) == Añadir_Nodo(Borrar_Arista(G, n, m), p)
    Borrar_Arista(Añadir_Arista(G, p, q), n, m) ==
                          SI (n = p AND m = q) OR (n = q AND m = p) ENTONCES
                                Borrar_Arista(G, n, m)
                          SI NO
                                Añadir_Arista(Borrar_Arista(G, n, m), p, q)
    Es_Vacio?(Crear) == VERDADERO
    Es_Vacio?(Añadir_Nodo(G, n)) == FALSO
    Es_Vacio?(Añadir_Arista(G, n, m)) == FALSO
    Hay_Arista?(Crear) == FALSO
    Hay_Arista?(Añadir_Nodo(G, n)) == Hay_Arista?(G)
    Hay_Arista?(Añadir_Arista(G, n, m)) == VERDADERO
    Contiene?(Crear, m) == FALSO
    Contiene?(Añadir_Nodo(G, n), m) == (n = m) OR Contiene?(G, m)
    Contiene?(Añadir_Arista(G, p, q), m) == Contiene?(G, m)
    Son_Adyacentes?(Crear, n, m) == FALSO
    Son_Adyacentes?(Añadir_Nodo(G, p), n, m) ==
    Son_Adyacentes?(G, n, m)
    Son_Adyacentes?(Añadir_Arista(G, p, q), n, m) ==        (n = p AND m = q) OR
                                                            (n = q AND m = p)  OR
                                                            Son_Adyacentes?(G, n, m)
    Contiene?(G, Nodo(G)) == VERDADERO
    Son_Adyacentes(G, Arista(G)) = VERDADERO
fin
```

En esta especificación se ha incluido como precondición de la operación Añadir_Arista que ambos nodos pertenezcan al grafo. Otra opción posible era tomar la operación Añadir_Arista como total, eliminando su precondición. Así, si se añade un arco y alguno de los nodos que lo forman no está presente en el grafo, éste se agregaría de forma implícita. En este caso, en un grafo en el que no hayamos hecho Añadir_Nodo(G, 1) es legal hacer Añadir_Arista(G, 2, 1).

Sin embargo esta opción tiene un comportamiento que puede resultar confuso. Si sobre G = Añadir_Arista(Añadir_Nodo(Crear, 2), 2, 1) hacemos Borrar_Nodo(G,2), como efecto lateral estaremos borrando el nodo 1 del grafo. El mismo problema aparece si lo que queremos es borrar el arco: al hacer Borrar_Arista(G,2,1) tendremos como efecto lateral la pérdida del nodo 1. Este tipo de efectos laterales suele pasar desapercibido al diseñador y producirse situaciones inesperadas.

Si queremos especificar un grafo dirigido, tenemos que introducir unas pequeñas modificaciones:

a) Definir BorrarArista(Añadir_Arista(G, p, q), n, m) como
   Borrar_Arista(Añadir_Arista(G, p, q), n, m) ==
      SI (n = p AND m = q) ENTONCES
         Borrar_Arista(G, n, m)
      SI NO
         Añadir_Arista(Borrar_Arista(G, n, m), p, q)
   Simplemente se quita la segunda parte de la condición para no considerar los arcos como bidireccionales.
b) Eliminar la ecuación Añadir_Arista(G, m, n) == Añadir_Arista(G, n, m), ya que ahora los arcos sólo tienen un sentido.
c) Sustituir la operación SonAdyacentes por las dos siguientes operaciones:

EsPredecesor? : Grafo × Elemento × Elemento ⟶ BOOLEAN
EsSucesor? : Grafo × Elemento × Elemento ⟶ BOOLEAN

EsPredecesor? devolverá VERDADERO si y sólo si existe un arco del primer nodo al segundo en el grafo que se le pasa de parámetro. Por su parte, EsSucesor? devolverá VERDADERO si y sólo si existe un arco del segundo nodo al primero en el grafo que se le pasa de parámetro.

```
EsPredecesor?(Crear, n, m) == FALSO
EsPredecesor?(AñadirNodo(G, n), p, q) == EsPredecesor?(G, p, q)
EsPredecesor?(AñadirArista(G, n, m), p, q) ==
                                    SI (n=p y m=q) ENTONCES
                                       VERDADERO
                                    SI NO
                                       EsPredecesor?(G, p, q)
EsSucesor?(Crear, n, m) == FALSO
EsSucesor?(AñadirNodo(G, n), p, q) ==    EsSucesor?(G, p, q)
EsSucesor?(AñadirArista(G, n, m), p, q) == SI (n=q y m=p) ENTONCES
                                       VERDADERO
                                    SI NO
                                       EsSucesor?(G, p, q)
```

6.3 Implementación de grafos.

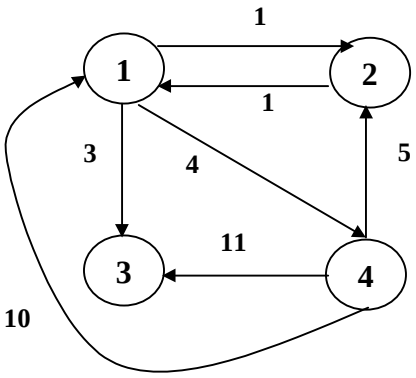
En este apartado veremos diferentes implementaciones del tipo grafo: una forma acotada (con una matriz de adyacencias) y dos no acotadas (con listas y multilistas de adyacencia).

6.3.1 Implementación mediante matrices de adyacencia.

Con la matriz de adyacencia, los arcos de un grafo $G = \{N, A\}$se representan como elementos de una matriz $n \times n$, donde n es Card(N).

Para un grafo no etiquetado, el elemento (i, j) de la matriz es 1 (o VERDADERO) si el arco (i, j) pertenece a A. Si el arco no pertenece, el elemento valdrá 0 (o FALSO). Si el grafo es etiquetado, se sustituyen los 1's por el valor de las etiquetas y los 0's por un valor distinguido (∞).

```
CONST
        MAXNODOS = (* Máximo número de nodos *)
TYPE
        GRAFO =     POINTER TO ESTRUCTURA;
        ARCO =      RECORD
                            existe : BOOLEAN;
                            peso : CARDINAL;
                    END;
        NODO = [1..N]
        MATRIZ =    ARRAY [NODO, NODO] OF ARCO;
        ESTRUCTURA =    RECORD
                                ult_lleno: [0 .. MAXNODOS]
                                matriz: MATRIZ;
                        END;
```



	1	2	3	4
1	∞	1	∞	4
2	1	∞	∞	∞
3	3	∞	∞	∞
4	10	5	11	∞

Figura 6. 6. Matriz de adyacencia de un grafo dirigido.

Como el tipo de los nodos de un grafo puede ser cualquiera, cuando el tipo base de N no es un subrango o un enumerado, no nos basta con definir NODO = [1..N], sino que hay que definir un array que contenga los datos sobre cada uno de los nodos del grafo y los valores de NODO = [1..N] actuarán como índice del array de nodos. La información de la i-

ésima fila de la matriz será la correspondiente al elemento que ocupe la i-ésima posición del array de elementos.

6.3.2 Implementación mediante listas de adyacencias.

En esta opción definimos una lista enlazada con un nodo por cada elemento del grafo. Cada nodo contendrá además una lista enlazada con sus elementos adyacentes.

```
TYPE
    GRAFO =     POINTER TO LISTA_NODOS:
    LISTA_ARCOS =   POINTER TO NODO_ARCO;
    LISTA_NODOS =   RECORD
                        nodo: NODO;
                        arcos: LISTA_ARCOS;
                        signodo: GRAFO;
                    END;
    NODO_ARCO =     RECORD
                        destino: NODO;
                        peso: CARDINAL;
                        sigarco: LISTA_ARCOS;
                    END;
```

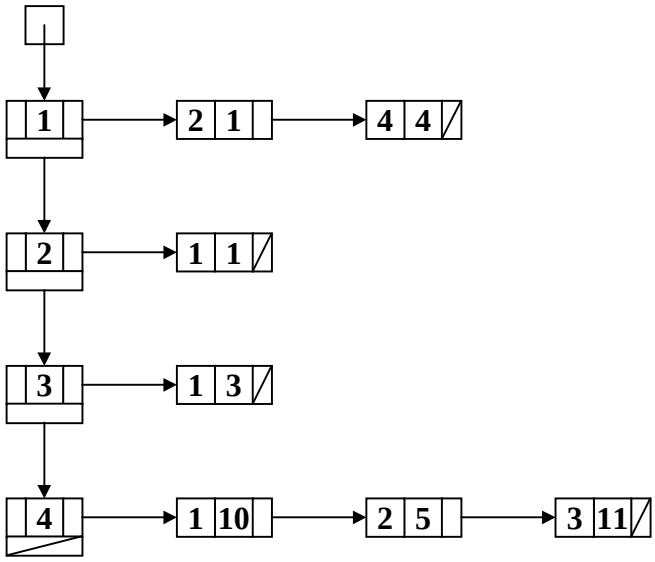


Figura 6. 7. Grafo implementado mediante listas de adyacencia.

En el caso de los grafos no dirigidos también se puede adoptar la estrategia de guardar sólo aquellos arcos donde el nodo origen sea menor o igual que el destino.

6.3.3 Implementación mediante multilistas de adyacencias.

Cuando aplicamos el criterio de ahorro de memoria o trabajamos con grafos dirigidos, el problema está en ver cuáles son los nodos origen de los arcos que llegan a un determinado nodo.

Para conseguir una implementación eficiente de dicha función se usan las multilistas de adyacencia.

```
TYPE
    GRAFO =      POINTER TO LISTA_NODOS:
    LISTA_ARCOS =   POINTER TO NODO_ARCO;
    LISTA_NODOS =   RECORD
                        nodo: NODO;
                        arcos_origen: LISTA_ARCOS;
                        arcos_destino: LISTA_ARCOS;
                        signodo: GRAFO;
                    END;
    NODO_ARCO =     RECORD
                        origen: NODO;
                        destino: NODO;
                        peso: CARDINAL;
                        sig_origen: LISTA_ARCOS;
                        sig_destino: LISTA_ARCOS;
                    END;
```

En esta representación se tiene por cada nodo una lista enlazada con los nodos destinos y otra lista enlazada con los nodos origen. El campo arcos_origen es la lista de arcos en los que el nodo es el origen y arcos_destino es la lista de arcos donde el arco es destino.

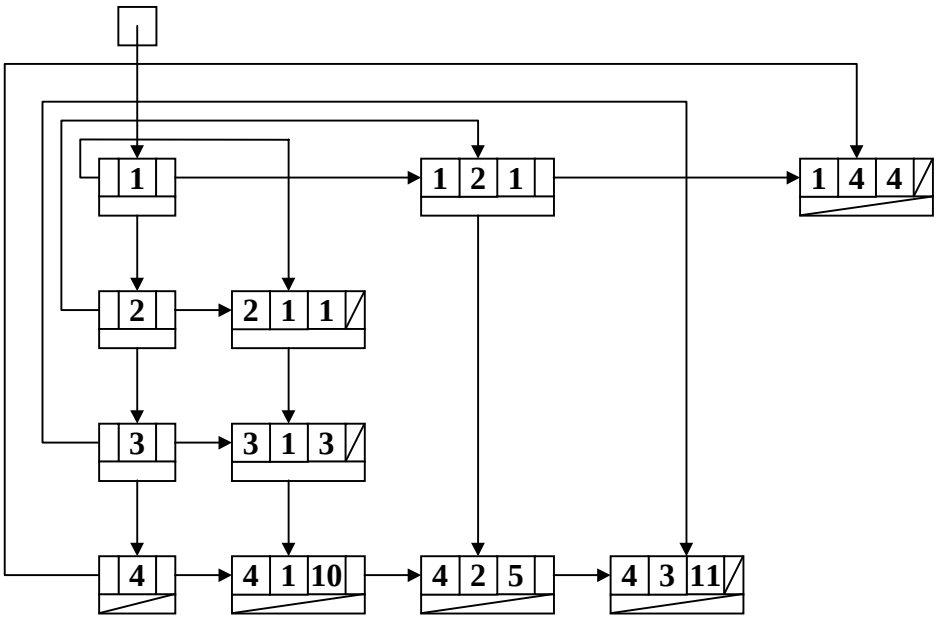


Figura 6. 8. Grafo implementado mediante multilistas de adyacencias.

Cuando exista un arco de un nodo a sí mismo, se ha de decidir si se guarda en la lista de sucesores, en la de predecesores o en ambas.

Esta representación no añade ninguna ventaja sobre las listas de adyacencia en el caso de grafos no dirigidos, pero pueden usarse sin problemas. En este caso también tenemos la posibilidad de ahorrar memoria representando cada arco en un solo sentido.

6.4 Recorridos sobre grafos.

Las funciones de recorrido de grafos son extensiones de las que se vieron para árboles. El objetivo es visitar todos los nodos del grafo exactamente una vez. Cuando se visita, es con la intención de realizar algún cálculo sobre él, ya sea simplemente imprimirlo u obtener algún resultado a partir de los valores de los nodos visitados.

Si el objetivo de un recorrido es simplemente pasar por todos los nodos del grafo, bastaría con hacer una pasada por el conjunto de nodos. En cualquiera de las implementaciones propuestas en el apartado anterior es una trivial. Pero queremos recorrer el grafo teniendo en cuanta su topología, es decir, lo recorreremos siguiendo los caminos definidos en él por los arcos. Vamos a describir dos estrategias de recorrido diferentes: el recorrido *primero en profundidad* y el recorrido *primero en amplitud*.

La diferencia fundamental es que hemos de tener en cuenta la posible existencias de ciclos en un grafo. Si en un recorrido llegamos otra vez al nodo origen, entramos en un bucle infinito. Para evitarlo se han de marcar los nodos por los que ya se ha pasado.

Los algoritmos que vamos a tratar visitan todos los nodos que son alcanzables desde un nodo origen dado. Para imprimir todos los nodos de un grafo inconexo, habría que ejecutar los algoritmos desde un nodo de cada subgrafo disjunto.

6.4.1 Recorrido primero en profundidad.

En el recorrido primero en profundidad se procesa un nodo inicial *n*. Después se procesa uno de los nodos adyacentes que no haya sido previamente visitado iniciando un recorrido en profundidad desde él. Cuando se alcanza un nodo cuyos nodos adyacentes han sido todos visitados el algoritmo vuelve atrás al último nodo visitado al que le quede aún algún sucesor por procesar. Finalmente, llegaremos a un punto donde todos los nodos alcanzables habrán sido visitados.

El código en pseudolenguaje sería el siguiente:

```
Algoritmo PrimeroEnProfundidad(G: Grafo; n: Elemento);
Variables
        p: Pila
        l, m: Elemento
Inicio
        Apilar(p, n);
        MIENTRAS NOT Vacía(p) HACER
                Cima(p, m)
                Desapilar(c)
                SI NOT Marcado(m) ENTONCES
```

```
                Marcar(m)
                Procesar(m)
            FIN SI
            PARA TODO nodo l sin marcar adyacente a m HACER
                Apilar(c, l)
            FIN PARA
        FIN MIENTRAS
Fin PrimeroEnProfundidad
```

Veamos cómo funciona el algoritmo con un ejemplo concreto:

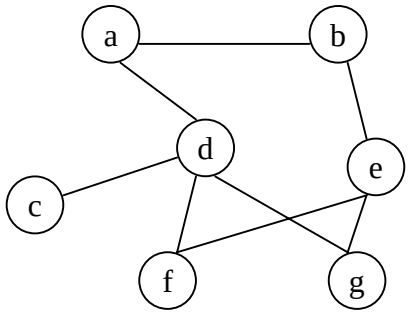


Figura 6. 9. Grafo no dirigido.

Supongamos que el elemento inicial es el nodo a. Se guarda en la pila. Como la pila no está vacía, precisamente contiene el nodo a, sacamos el primer elemento. Al no estar marcado, lo marcamos y lo procesamos. Podemos suponer que, en este caso, el procesamiento consiste en escribir su valor por pantalla. Los dos elementos adyacentes a a son b y d. Ambos están sin marcar y los incluimos en la pila en, por ejemplo, ese orden.

La pila no está vacía por lo que sacamos el elemento superior, d, que no está marcado. Se marca y se escribe por pantalla. Los nodos c, f y g son adyacentes a d y no están marcados. Se apilan los tres. En la cima de la pila está g, el cual sacamos. No está marcado, por lo dejamos marcado y lo escribimos por pantalla. El único nodo adyacente a g sin marcar es e, porque d ya está marcado, por lo que sólo apilamos e, que pasa a ser la nueva cima. Sacamos e de la pila, lo marcamos y lo escribimos por pantalla. Los sucesores de e que no han sido marcados aún son b y f, por tanto los apilamos. Aunque b y f ya están en la pila, no han sido procesados. Como se hemos llegado a ellos por dos caminos, los sacaremos de la pila dos veces pero sólo los procesaremos una, porque antes de procesarlos se pregunta si han están marcados y después de la primera visita ya lo estarán.

En este momento, la cima de la pila es f. La sacamos y, al no estar marcada, la marcamos y la escribimos. Los tres nodos adyacentes a f son d, e, y g y todos están marcados, por lo que no se apilan. Sacamos la cima de la pila, el nodo b que no está marcado. Lo marcamos y lo procesamos. Sus nodos adyacentes, a y e, no se apilan al estar ya marcados. Sacamos otra vez la cima, que resulta ser la otra aparición de f. Como f ya está marcado, no se procesa. Volvemos a sacar la cima, que esta vez es c, que está aún sin marcar. Pon tanto, se marca y se procesa. Como todos sus nodos adyacentes están marcados, no se apila ninguno. Y finalmente sacamos la otra aparición de b en la pila, que es el único

elemento que queda. Como está marcada, no se hace nada. El resultado por pantalla sería entonces: a, d, g, e, f, b, c.

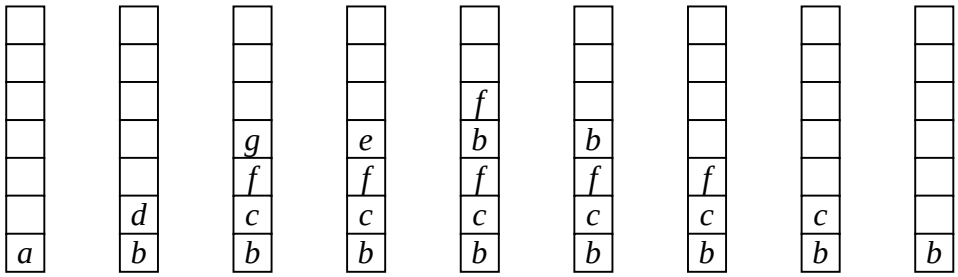


Figura 6. 10. Contenido de la pila en el recorrido primero en profundidad.

En este algoritmo hemos hecho uso de una pila. Podemos sustituir esta versión iterativa por una recursiva más elegante que use implícitamente la pila del sistema.

```
Algoritmo PrimeroEnProfundidad(G: Grafo; n: Elemento);
Inicio
        Marcar n como visitado;
        MIENTRAS quede algún nodo m adyacente a n no haya sido visitado HACER
                PrimeroEnProfundidad(G, m)
        FIN MIENTRAS
Fin PrimeroEnProfundidad
```

Hay que hacer notar que en el recorrido hemos supuesto implícitamente un orden entre los nodos adyacentes a uno dado en el sentido de que hemos tenido que elegir entre los posible nodos adyacentes para procesarlos secuencialmente. Si el orden escogido hubiera sido otro, el recorrido hubiera sido diferente. Si en vez del *orden alfabético* que hemos seguido en el ejemplo hubiéramos seguido el orden inverso, el recorrido resultante hubiera sido: a, b, e, f, d, c, g.

6.4.2 Recorrido primero en amplitud.

El recorrido en amplitud es similar al recorrido por niveles de un árbol, si consideramos como un nivel el conjunto de los nodos adyacentes. En este recorrido, tras visitar un nodo, se visitan todos sus nodos adyacentes. Una vez visitados todos los nodos de este primer nivel, se repite el proceso visitando todos los nodos adyacentes del primer vecino de nivel recién recorrido y repitiéndolo para todos los nodos hasta que no quede ninguno por visitar.

En el recorrido en profundidad usábamos una pila como estructura auxiliar para recorrer el grafo. Aunque en el recorrido en amplitud no hacemos vuelta atrás también necesitamos almacenar temporalmente los nodos. Para guardarlos en el orden adecuado, usaremos una cola en vez de una pila.

El código en pseudolenguaje sería el siguiente:

```
Algoritmo PrimeroEnAmplitud(G: Grafo; n: Elemento);
```

```
Variables
        c: Cola
        l, m: NODO
Inicio
        Insertar(c,n);
        MIENTRAS NOT Vacía(c) HACER
                Frente(c, m)
                Extraer(c)
                Marcar(m)
                PARA TODO nodo l sin marcar adyacente a m HACER
                        Insertar(c, l)
                FIN PARA
        FIN MIENTRAS
Fin PrimeroEnAmplitud
```

Si hacemos PrimeroEnAmplitud(G, a), siendo G el nodo del ejemplo anterior tendremos la siguiente secuencia. Insertamos a en la cola. Como la cola no está vacía, extraemos el primer elemento, que es a, por ser el único. Insertamos los nodos adyacentes a a sin marcar: b y d. Como la cola no está vacía, sacamos el primer elemento de la cola, b, e insertamos los nodos adyacentes no visitados. Sólo insertamos e. Ahora el frente de la cola es d. Lo sacamos y guardamos sus sucesores: c, f y g. El siguiente elemento que sale de la cola es e, al que no le queda ningún vecino sin marcar. Igual pasa cuando sacamos c, f y finalmente g de la cola. Llegados a este punto, la cola se ha quedado vacía y se acaba el algoritmo. El recorrido ha sido: a, b, c, d, e, f , g.

6.4.3 Especificación de los algoritmos de recorrido.

Tras la definición de los dos recorridos y su implementación en pseudocódigo, vamos a ver cómo es su especificación:

PrimeroEnProfundidad:	Grafo	$\longrightarrow$	Lista
PEP_2:	Grafo × Lista × Elemento × Pila	$\longrightarrow$	Lista
Quitar:	Grafo × Lista	$\longrightarrow$	Grafo
Hermanos:	Conjunto × Grafo	$\longrightarrow$	Conjunto
Meter:	Pila × Conjunto	$\longrightarrow$	Pila
ConsFinal:	Lista × Elemento	$\longrightarrow$	Lista

```
PrimeroEnProfundidad(G) ==
                SI Es_Vacío(G) ENTONCES
                        Crear_L
                SI NO SEA   L = PEP2(G, Crear_L, Nodo(G), Crear_P)
                            G' = Quitar(G, L)
                EN
                        SI NOT Es_Vacío(G') ENTONCES
                                Concat(L, PEP2(G', Crear_L, Nodo(G'), Crear_P)
                        SI NO
                                L

PEP2 (G, L, n, p) ==  SI Está(L, n) ENTONCES
```

```
                                SI Es_Vacía(p)= ENTONCES
                                        L
                                SI NO
                                        PEP2(G, L, Cima(p), Desapilar(p))
                        SI NO SEA p' = Meter(p, Hermanos(n, G))
                        EN      SI Es_Vacía(p') ENTONCES
                                        ConsFinal(L, n)
                                SI NO
                                        PEP2(G, ConsFinal(L,n),Cima(p),Desapilar(p'))

Quitar(G, CrearL) ==  G
Quitar(G, Cons(e, L)) ==        Quitar(Borrar_Nodo(G, e), L))

Hermanos(n, CrearG) == CrearC
Hermanos(n, Añadir_Arista(G, n1, n2)) ==   SI  n1 = n ENTONCES
                                                Añadir(n2, Hermanos(n, G))
                                           SI NO
                                                SI n2 = n ENTONCES
                                                        Añadir(n1,Hermanos(n,G))
                                                SI NO
                                                        Hermanos(n, G)
Hermanos(n, Añadir_Nodo(G, m)) ==       Hermanos (n, G)

Meter(p, c) ==          SI Es_Vacío(c) ENTONCES
                                p
                        SI NO
                                Apilar(Elemento(c), Meter(p, Eliminar(c, Elemento(c))))

ConsFinal(CrearL, n) ==         Cons(n, CrearL)
ConsFinal(Cons(e, l), n) ==     Cons(e, ConsFinal(l, n))
```

Para el recorrido primero en amplitud, basta con cambiar las operaciones de Pila por Cola: Apilar por Insertar, Desapilar por Extraer y Cima por Frente y sustituir la Pila por una Cola en las operaciones PEP_2 y Meter.

Esta especificación está pensada para un grafo no dirigido. Si el grafo es dirigido, hay que eliminar

```
SI v = n ENTONCES
        Añadir(u, Hermanos(n, G))
```

de la ecuación Hermanos(n, Añadir_Arista(G,n_1,n_2))

6.5 Árboles de extensión de coste mínimo.

En un grafo es habitual encontrar caminos redundantes. Es decir, es posible llegar de un nodo a otro a través de diferentes caminos. Si en grafo conexo eliminamos los caminos redundantes, seguirá siendo conexo, se podrá llegar de cualquier nodo a cualquier otro.

Dado un grafo $G = (N, A)$, conexo con los arcos etiquetados con su coste, un *árbol de extensión* asociado a G es un árbol libre que conecta todos los nodos de N. El *coste* del árbol

es la suma de los costes de todos los arcos del árbol libre. En este apartado veremos dos algoritmos para encontrar el árbol de extensión de coste mínimo.

Una aplicación típica de este algoritmo es el diseño de una red de comunicaciones (telefónica o de Internet, por ejemplo). El árbol de extensión de coste mínimo nos dará la opción de conexión de todas las ciudades al menor coste, ya sea éste el precio del tendido de los cables, el mantenimiento, etc.

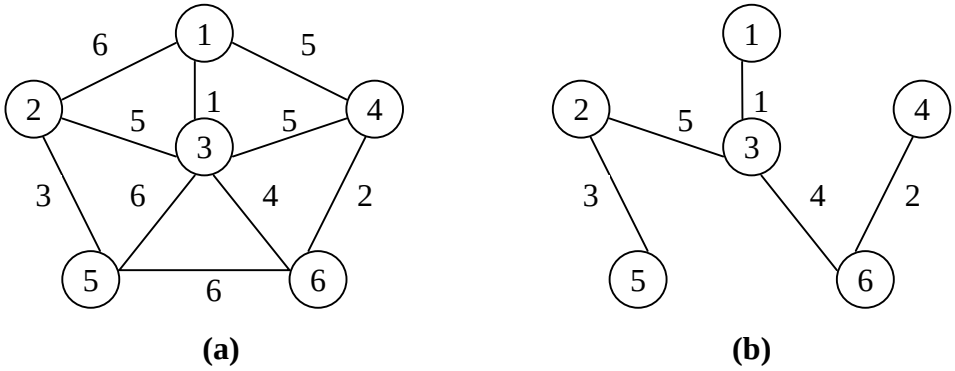


Figura 6. 11. (a) Grafo etiquetado y (b) su árbol de expansión de coste mínimo.

6.5.1 La propiedad AAM.

Los dos algoritmos que describiremos aprovechan la propiedad *AAM* de los árboles de extensión de coste mínimo. Esta propiedad establece que, en un grafo $G = (N, A)$, con los arcos etiquetados con su coste, con $U \subset N$, si (u, n) es un arco de coste mínimo tal que $u \in U$ y $n \in N$, entonces, existe un árbol de extensión de coste mínimo que incluye (u, n) entre sus arcos.

6.5.2 El algoritmo de Prim.

Supongamos un grafo $G = (N, A)$, con los arcos etiquetados con su coste. El algoritmo de Prim va construyendo un subconjunto de nodos U, aumentándolo hasta que contenga todos los nodos de N. Inicialmente, U contiene un único nodo de N, que puede ser cualquiera. En cada paso, se localiza el arco más corto (u, n) tal que $u \in U$ y $n \in N$, repitiendo el proceso hasta que $U = N$.

```
Algoritmo Prim(G: Grafo; VAR T; Conjunto de Arcos);
Variables
        U: Conjunto de arcos
        u, n: arcos
Inicio
        T := {}
        U := {Cualquier n ∈ N}
        MIENTRAS  U <> n HACER
                Sea (u, n) el arco de coste mínimo tal que u ∈ U y n ∈ N.
                T := T ∪ {(u, n)}
                U := U ∪ {n}
```

FIN MIENTRAS
Fin Prim

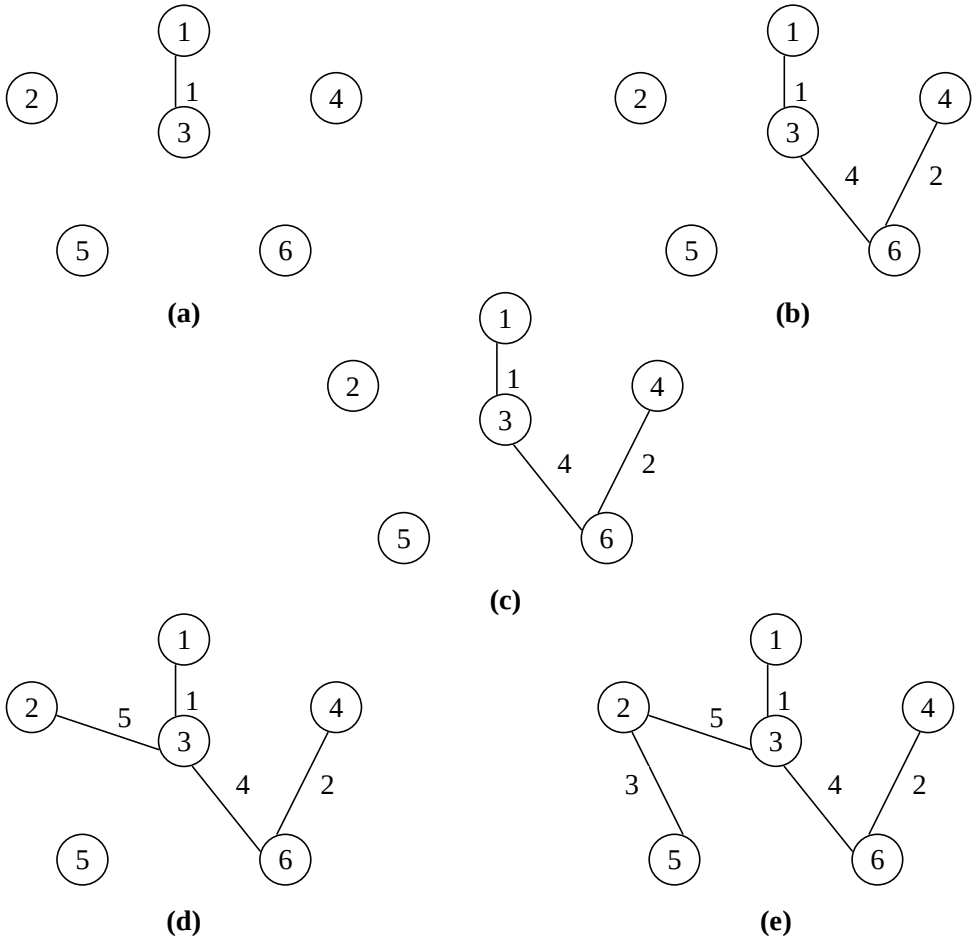


Figura 6. 12. Secuencia de aristas añadidas por el algoritmo de Prim.

Para encontrar fácilmente el arco de menor coste entre *U* y *N* - *U*, mantendremos dos matrices, *MAS_CERCANO[i]*, que devuelve el nodo en *U* más cercano a *i* en *N* - *U* y *MENOR_COSTE[i]* que devuelve el coste del arco *(i, MAS_CERCANO[i])*. En cada paso, se escoge *k* tal que $k \in N - U$ es el nodo el más cercano a *U* y se actualizan ambas matrices teniendo en cuenta que ahora $k \in U$. Para evitar que k vuelva a ser seleccionado, se asigna $MENOR\_COSTE[k] = \infty$.

La especificación algebraica del algoritmo de Prim es la siguiente:

operaciones

Prim: Grafo $\longrightarrow$ Grafo
Prim2: Grafo $\times$ Grafo $\longrightarrow$ Grafo
P: Grafo $\times$ Grafo $\longrightarrow$ BOOLEAN
Min_Arco: Grafo $\times$ Grafo $\longrightarrow$ N
Es_V_E_A: Grafo $\longrightarrow$ BOOLEAN
Min: Elemento $\times$ Elemento $\times$ N $\times$ Elemento $\times$ Elemento $\times$ N $\longrightarrow$
Elemento $\times$ Elemento $\times$ N

precondiciones

pre Min_Arco(G, G') : P(G, G')

ecuaciones

P(Crear, G') == FALSO

P(Añadir_Nodo(G, a), G') == P(G, G')

P(Añadir_Arco(G, u, v, n), G') == (Esta(u, G') AND NOT Esta(v, G')) OR
(Esta(v, G') AND NOT Esta(u, G')) OR
P(G, G')

Prim(G) == SI Es_Vacío(G) ENTONCES
Crear(G)
SI NO
$Prim_2$(G, Añadir_Nodo($Crear_G$, Elemento(G))

$Prim_2$(G, G') =
SI NOT P(G, G') ENTONCES
G'
SI NO
SEA n_1, n_2, c = Min_Arco(G, G')
EN
$Prim_2$(Borrar_Arco(G, n_1, n_2),
Añadir_Arco(Añadir_Nodo(Añadir_Nodo(G',n_2), n_1), n_1, n_2, c))

Min_Arco(Añadir_Nodo(G, n), G') == Min_Arco(G, G')

Min_Arco(Añadir_Arco(G_1, n_1, n_2, c), G') ==
SEA n_1, n_2, c = Arco(G_1)
G = Borrar_Arco(G_1, n_1, n_2)
EN
SI NOT P(G) ENTONCES
n_1, n_2, c
SI NO SI (Esta(n_1, G') AND NOT Esta(n_2, G')) OR
(Esta(n_2, G') AND NOT Esta(n_1, G')) ENTONCES
Min(n_1, n_2, c, Min_Arco(G, G')
SI NO
Min_Arco(G, G')

Es_V_E_A(CrearG) == VERDADERO

Es_V_E_A(Añadir_Nodo(G, n)) == Es_V_E_A(G)

Es_V_E_A(Añadir_Arco(G, n_1, n_2, c)) == FALSO

Min(n_1, n_2, c, n_1', n_2', c') == SI c < c' ENTONCES
n_1, n_2, c
SI NO
n_1', n_2', c'

6.5.2 El algoritmo de Kruskal.

El algoritmo de Kruskal resuelve el problema del árbol de extensión de coste mínimo utilizando una estrategia distinta a la usada en el algoritmo de Kruskal. En este caso, dividiremos el grafo de partida en diferentes subgrafos e iremos conectando los subgrafos entre sí por el arco de menor coste posible. Al unir los subgrafos tendremos en cuenta que no se formen ciclos, que no están permitidos en un árbol de expansión. Cuando todos los subgrafos estén unidos, habremos encontrado el árbol de expansión de coste mínimo. Para

asegurar que en todo momento escogemos el arco de menor coste, siempre buscaremos en orden ascendente de coste dentro del conjunto de arco.

Dado un nodo $G = \{N, A\}$, se define un grafo T = {N, ∅}. T es, por tanto, un grafo con todos los nodos de G pero sin ninguna arista. Podemos considerar que en T todos los nodos son subgrafos conexos. Durante el algoritmo se mantendrá siempre un conjunto de subgrafos conexos y se añadirán los arcos que conecten estos subgrafos en otros más grandes. Estos arcos son los que formarán el árbol de extensión de coste mínimo.

Para añadir los arcos examinamos el conjunto *A* de arcos en orden creciente según su coste. Si el arco inspeccionado conecta dos nodos de subgrafos diferentes, se añade al conjunto de nodos de T puesto que formará parte del árbol de extensión y consideraremos esos dos subgrafos como un solo subgrafo conexo. Si los dos arcos pertenecen al mismo subgrafo, no se añade para evitar la aparición de ciclos en el árbol de extensión. Cuando todos los nodos de T estén conectados, habremos encontrado el árbol de extensión buscado.

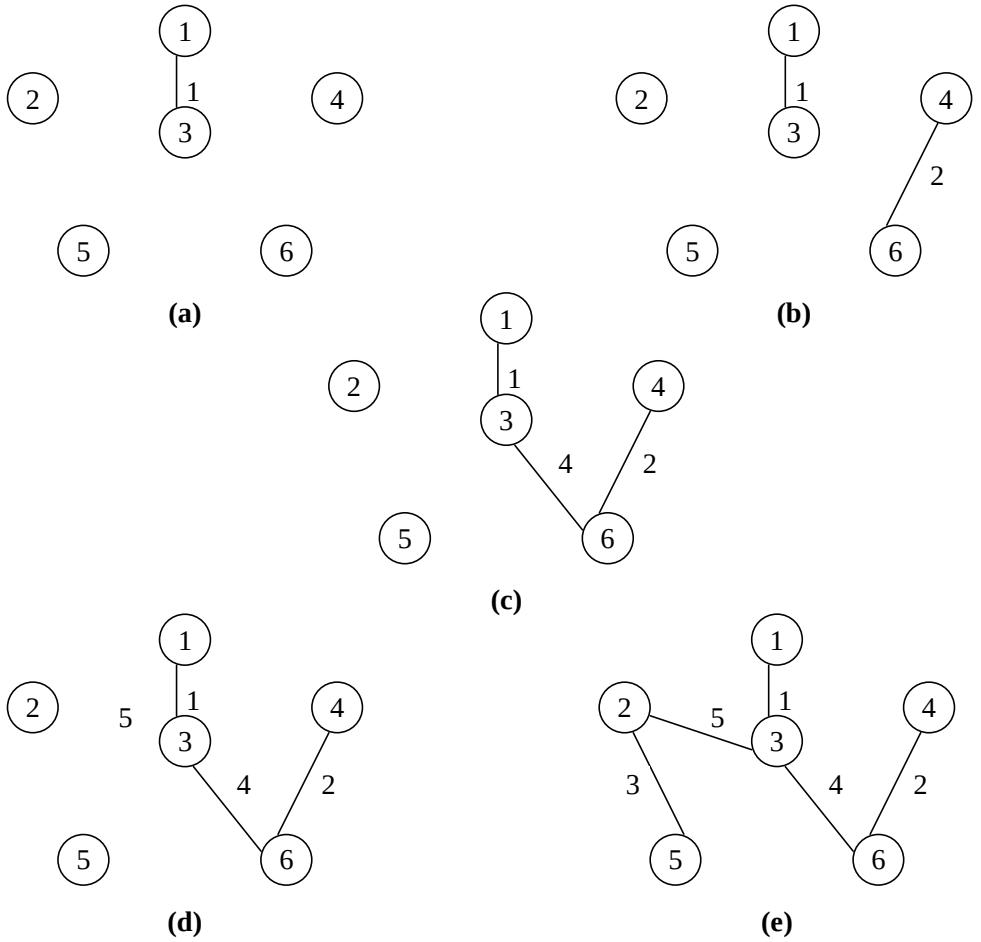


Figura 6. 13. Secuencia de aristas añadidas por el algoritmo de Kruskal.

```
Algoritmo Kruskal(G: Grafo; VAR T: Conjunto de Arcos)
Variables
        arcos:ColaPrio
        subgrafos: Conjunto de Conjunto de Nodos
        a: Arco
```

```
        n, m: Nodo
Inicio
        T := Crear_CONJ
        subgrafos := Crear_CONJ
        arcos := Crear_COLAPRIO

        (* Insertamos todos los nodos de N como    *)
        (*subgrafos de un solo elemento.           *)
        PARA TODO n ∈ N HACER
                Insertar({n}, subgrafos)
        FIN PARA
        (* Insertamos todos los arcos de A en un cola de prioridades      *)
        (* ordenadas de forma ascendente por su coste                     *)
        PARA TODO a ∈ A HACER
                Insertar(a, arcos)
        FIN PARA
        (* Sacamos arcos de la cola de prioridad hasta que sólo quede     *)
        (* un grafo. Si no provoca ciclos, lo añadimos al árbol.          *)
        MIENTRAS Card(subgrafo) >1 HACER
                a := Frente(arcos)
                Extraer(arcos)
                Sea a = (n, m)
                c_n = SubgrafoQueContiene(n)
                c_m = SubgrafoQueContiene(m)
                SI (c_n ≠ c_m) ENTONCES
                        Inserta(a, T)
                        c_u := Union(c_n, c_m)
                        Extraer(subgrafos, c_n)
                        Extraer(subgrafos, c_m)
                        Insertar(subgrafos, c_u)
                FINSI
        FIN MIENTRAS
Fin Kruskal
```

6.6 Búsquedas de caminos.

En este apartado vamos a ver una serie de algoritmos de búsquedas de caminos en el grafo. Estos algoritmos nos permitirán encontrar:

- un camino de un nodo S a otro nodo T,
- el camino más corto entre S y T,
- el camino de menor peso entre S y T y
- los caminos de menor peso entre S y los demás nodos del grafo.

6.6.1 Búsqueda de un camino entre un nodo S y otro nodo T.

El problema de encontrar un camino entre dos nodos S y T es un problema muy sencillo que se resuelve fácilmente realizando un recorrido en profundidad partiendo de S y parando cuando se encuentra T.

6.6.2 Búsqueda del camino más corto desde un nodo S.

En la actualidad no se han desarrollado algoritmos en los cuales encontrar el camino entre un nodo S y otro nodo T sea más rápido que encontrar el camino más corto de S a los demás nodos, por lo que describiremos el algoritmo para éste último problema.

Supondremos un grafo dirigido y un nodo inicial en el grafo. Para ilustrar el algoritmo veremos un ejemplo concreto.

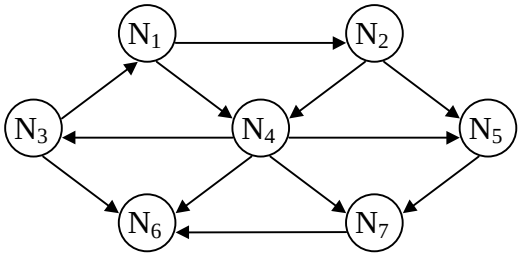


Figura 6. 14. Búsqueda de caminos mínimos en un grafo.

La estrategia a seguir en este algoritmo será realizar un recorrido primero en amplitud registrando información adicional al paso de cada nodo, información que posteriormente será tratada para conocer el camino real.

Deseamos conocer los caminos más cortos desde el nodo N_3. De entrada, el camino más corto desde N_3 a N_3 es de longitud cero. A continuación se buscan todos los nodos que estén a distancia uno, o sea, que sean adyacentes. N_1 y N_6 están a una arista de distancia. Ahora buscamos los nodos que estén a distancia exactamente dos y que no hayan sido visitados aún. Éstos serán los nodos adyacentes a N_1 y N_6: N_2 y N_4. Continuando de la misma manera, exploramos lo nodos adyacentes a N_2 y N_4 no visitados, que son N_5 y N_7. A éstos el camino es de longitud tres. Así se continua hasta que se hayan visitado todos los nodos.

Con esta información sólo sabremos la longitud del camino. Si se desea conocer el camino real, es decir, la sucesión de nodos que nos lleva al destino, es necesario conservar información adicional. Con esta finalidad, el algoritmo ha de ir rellenando una tabla con la siguiente información:

NODO	MARCA	DISTANCIA	CAMINO
N_1			
N_2			

N_3			
N_4			
N_5			
N_6			
N_7			

La entrada MARCA se pone a VERDADERO después de procesar un nodo. Al principio, todas son FALSO. Cuando un nodo está e VERDADERO, se tiene la certeza de que no existe un camino más corto y por tanto su proceso ha terminado.

La entrada DISTANCIA tendrá un valor muy grande (∞) para todos los nodos excepto para el inicial, que será 0.

La entrada CAMINO indica el nodo desde el que se ha llegado y nos sirve para conocer el camino real.

6.6.3 Búsqueda del camino de menor peso desde un nodo S.

La búsqueda de caminos de menor peso tiene sentido cuando hablamos de grafos etiquetados. El problema de encontrar el camino más corto puede considerarse el caso particular de un grafo etiquetado donde todas las etiquetas tienen peso 1. El método general para resolver este problema es el llamado algoritmo de Dijkstra.

En este algoritmo se maneja la misma información que en el del apartado anterior. Cada nodo se marca a su paso con una distancia provisional para cada nodo. Esta distancia resulta ser la longitud del camino más corto desde el origen a N_i usando como nodos intermedios sólo nodos marcados. En CAMINO se almacena el último nodo que ha ocasionado un cambio en el valor de DISTANCIA.

En cada etapa, el algoritmo selecciona entre los nodos no visitados aquel que tiene el camino más corto al origen. El primer nodo seleccionado es el nodo origen, que tiene una camino de peso 0. Los demás tienen un camino de peso inicial ∞.

Una vez seleccionado el nodo, comprueba si se puede mejorar la distancia a sus nodos adyacentes no marcados. Si el peso del camino hasta el nodo seleccionado más el peso del arco hasta el nodo adyacente es menor que el peso del camino provisional hasta el nodo adyacente, modificamos su distancia, adaptándola a la del nuevo camino. El campo CAMINO también se actualizará a este nuevo nodo. El algoritmo acaba una vez que se hayan visitado todos los nodos.

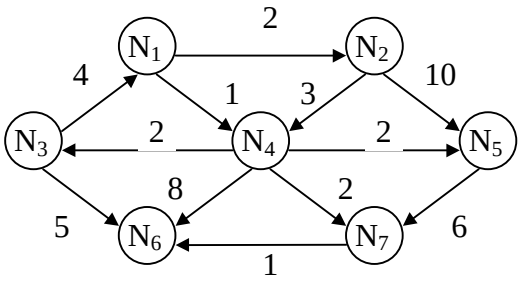


Figura 6. 15. Búsqueda de caminos de peso mínimo en un grafo.

Sigamos la ejecución del algoritmo de Dijsktra con el grafo de la figura 6.11 usando como nodo inicial el nodo N_1. El nodo N_1 será por tanto el primer nodo seleccionado. En la tabla de marca como visitado y se inspeccionan los nodos adyacentes, N_2 y N_4. En ambos casos el coste mínimo hasta ese momento es infinito, por lo que hemos encontrado un camino mejor pasando por N_1. El coste de este camino es el coste del camino a N_1 más el del arco que los une. Para N_2 es $d_2 = d_1 + c_{1,2} = 0 + 2 = 2$. Para N_4 es $d_4 = d_1 + c_{1,4} = 0 + 1 = 1$. En ambos casos, el campo CAMINO se actualiza a N_1. De estos dos nodos, se selecciona N_4 por ser el que tiene del camino de menor coste. Se marca y se analizan los nodos adyacentes a N_4, que son N_3, N_5, N_6 y N_7. En todos los casos, como no había un camino previo, se toma como mejor camino de forma provisional el camino a través de N_4, actualizándose los campos DISTANCIA y CAMINO.

El siguiente nodo seleccionado en N_2, que tiene como nodos adyacentes N_4 y N_5. Como N_4 está marcado como visitado, y tiene por tanto un camino mejor que N_2, no es trabaja con él. Como el camino a N_5 a través de N_2 tiene de peso $d_5 = d_2 + c_{2,5} = 2 + 10 = 12$ no es mejor que el que ya tenía N_5, por lo que no se modifica.

El resto de los nodos se escogen en el siguiente orden: N_5, N_3, N_7 y N_6, quedando la tabla con los siguientes valores finales:

NODO	MARCA	DISTANCIA	CAMINO
N_1	VERDADERO	0	-
N_2	VERDADERO	2	N_1
N_3	VERDADERO	3	N_4
N_4	VERDADERO	1	N_1
N_5	VERDADERO	3	N_4
N_6	VERDADERO	6	N_7
N_7	VERDADERO	5	N_4

6.6.3.1. Especificación algebraica del algoritmo de Dijkstra.

Vamos a ver en este apartado la especificación algebraica del algoritmo de búsqueda de caminos de peso mínimo aplicado a grafos dirigidos.

Dijkstra: Elemento × Grafo —/→ Grafo
$Dijkstra_2$: Elemento × Grafo × Grafo ⟶ Grafo
MinDistancia: Grafo × Grafo × Elemento —/→ Elemento × Elemento × Peso × Peso
Distancia: Elemento × Elemento × Grafo —/→ Peso
MinArco: Elemento × Elemento × Peso × Peso ×
Elemento × Elemento × Peso × Peso ⟶
Elemento × Elemento × Peso × Peso
Es_Conexo: Grafo ⟶ BOOLEAN
Están_Conectados: Elemento × Elemento × Grafo ⟶ BOOLEAN
P: Grafo × Grafo ⟶ BOOLEAN
Incluido_En_Nodos: Grafo × Grafo ⟶ BOOLEAN

precondiciones

Dijkstra (a, G): Está(a, G)
MinDistancia(G, G', a): Está(a, G') AND Es_Conexo(G') AND P(G, G')
Distancia(n, m, G): Están_Conectados(n, m, G)

ecuaciones

Dijkstra(a, G) == $Dijkstra_2$(a, G, Añadir_Nodo(a, Crear))
$Dijkstra_2$(a, G, G') ==SI P(G, G') ENTONCES
SEA n, m, c, t = MinDistancia(a, G', G) EN
$Dijkstra_2$(a, G, Añadir_Arco(Añadir_Nodo(G', m),n,m,c))
SI NO
G'
MinDistancia(a, G', G) ==
SEA n, m, c == Arco(G), G_1 = Borrar_Arco(G, n, m) EN
SI Está(n, G') AND NOT Está(m, G') ENTONCES
SI P(G_1, G') ENTONCES
MinArco(n, m, c Distancia(a, m, G')+c, MinDistancia(a, G', G_1))
SI NO
n, m, c Distancia(a, m, G') + c
SI NO
SI Está(m, G') AND NOT Está(n, G') ENTONCES
SI P(G_1, G') ENTONCES
MinArco(n,m,c,Distancia(a,v,G')+c, MinDistancia(a,G', G_1)
SI NO
n, m, c, Distancia(a, m, G') + c
SI NO
MinDistancia(a, G', G_1)
Distancia(n, m, G) =
SI n = m ENTONCES
0
SI NO
SEA r, s, c = Arco(G), G' = Borrar_Arco(G, r, s) EN
SI Están_Conectados(n, r, G') AND

Están_Conectados(s, m, G') ENTONCES
SI Están_Conectados(n, m, G') ENTONCES
MIN(Distancia(n, r, G') + Distancia(s, m, G') + c,
Distancia(n, m, G'))
SI NO
Distancia(n, r, G') + Distancia(s, m, G') + c
SI NO
Distancia(n, m, G')

MinArco(n, m, c , d, n', m', c', d') == SI d < d' ENTONCES
n, m, c, d
SI NO
n', m', c', d'

EsConexo(G) == Incluido_En_Nodos(Prim(G), G)

Incluido_En_Nodos(Crear, G) == VEREDADERO

Incluido_En_Nodos(Añadir_Nodo(G', a), G) == Está(a, G) AND
Incluido_En_Nodos(G', G)

Incluido_En_Nodos(Añadir_Arco(G', n, m, c), G) == Incluido_En_Nodos(G', G)

Están_Conectados(n, m, G) == SI NOT Está(n, G) ENTONCES
FALSO
SI NO
SEA G' = $Prim_2$(G, Añadir_Nodo(n, Crear)) EN
Está(m, G')

P(Crear, G') == FALSO

P(Añadir_Nodo(G, n), G') == P(G, G')

P(Añadir_Arco(G, n, m, c), G') == (Está(n, G') AND NOT Está(m, G')) OR
(Está(m, G') AND NOT Está(n, G')) OR
P(G, G')

6.7 Implementación del tipo grafo.

A continuación se ofrece la implementación en MODULA-2 del tipo grafo

```
DEFINITION MODULE Grafo;

TYPE
    GRAFO;
    NODO = CARDINAL;
    ERROR_GRAFO = (No_Error, No_Creado, Sin_Memoria, Nodo_Inexistente);
VAR
    Error_Grafo : ERROR_GRAFO;

PROCEDURE Crear() : GRAFO;
PROCEDURE AddNodo(VAR g : GRAFO; n : NODO);
PROCEDURE AddArista(VAR g : GRAFO; n, m : NODO);
PROCEDURE BorrarNodo(VAR g : GRAFO; n : NODO);
PROCEDURE BorrarArista(VAR g : GRAFO; n, m : NODO);
PROCEDURE Contiene(g : GRAFO; n : NODO) : BOOLEAN;
PROCEDURE SonAdyacentes(g : GRAFO; n, m : NODO) : BOOLEAN;
PROCEDURE EsVacio(g : GRAFO) : BOOLEAN;
PROCEDURE Destruir(VAR g : GRAFO);

END Grafo.

IMPLEMENTATION MODULE Grafo;
```

```
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

CONST
    MAX = 20;
TYPE
    GRAFO = POINTER TO ESTRUCTURA;
    ESTRUCTURA = RECORD
                     nodos : ARRAY [1..MAX] OF NODO;
                     matriz : ARRAY [1..MAX],[1..MAX] OF BOOLEAN;
                     num_nodos : [0..MAX];
                 END;

PROCEDURE Posicion(g : GRAFO; n : NODO) : CARDINAL;
(* Proc. interno que dado un grafo ya creado, nos dice en que posici¢n de
   la matriz de adyacencia se encuentra un nodo determinado.
   Si el nodo no se encuentra, se retorna un 0 *)
VAR
      i : [1..MAX];
BEGIN
      i := 1;
      WHILE (i <= g^.num_nodos) AND (g^.nodos[i] <> n) DO
            INC(i);
      END;
      IF (i > g^.num_nodos) THEN
            RETURN 0;
      ELSE
            RETURN i;
      END;
END Posicion;

PROCEDURE Crear() : GRAFO;
VAR
      g : GRAFO;
BEGIN
      Error_Grafo := No_Error;
      IF NOT Available(SIZE(g^)) THEN
            Error_Grafo := Sin_Memoria;
            g := NIL;
      ELSE
            NEW(g);
            g^.num_nodos := 0;
      END;
      RETURN(g)
END Crear;

PROCEDURE AddNodo(VAR g : GRAFO; n : NODO);
VAR
  i : CARDINAL;
BEGIN
(* Comprobaci¢n de que el nodo ha sido efectivamente creado *)
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      ELSIF g^.num_nodos = MAX THEN (* Comprobaci¢n de que la estructura
puede albergar otro nodo m s *)
            Error_Grafo := Sin_Memoria;
      ELSIF Posicion(g, n) = 0 THEN (* Comprobaci¢n de que el nodo no est
ya *)
(* Inicializaci¢n de las aristas con ese nodo *)
            INC(g^.num_nodos);
            g^.nodos[g^.num_nodos] := n;
```

```
            FOR i := 1 TO g^.num_nodos DO
                  g^.matriz[g^.num_nodos, i] := FALSE;
                  g^.matriz[i, g^.num_nodos] := FALSE;
            END;
      END;
END AddNodo;

PROCEDURE AddArista(VAR g : GRAFO; n, m : NODO);
VAR
      i, j : [1..MAX];
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      ELSE
            i := Posicion(g, n);
            j := Posicion(g, m);
  (* Comprobaci¢n de que los nodos a conectar existen en el grafo *)
            IF (i = 0) OR (j = 0) THEN
                  Error_Grafo := Nodo_Inexistente;
            ELSE
                  g^.matriz[i, j] := TRUE;
                  g^.matriz[j, i] := TRUE;
            END;
      END;
END AddArista;

PROCEDURE BorrarNodo(VAR g : GRAFO; n : NODO);
VAR
      i, cont, cont2 : [0..MAX];
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      ELSE
            i := Posicion(g, n);
            IF i <> 0 THEN
                  FOR cont := i TO g^.num_nodos - 1 DO
                        g^.matriz[cont] := g^.matriz[cont + 1];
                        g^.nodos[cont] := g^.nodos[cont + 1];
                  END;
                  FOR cont := 1 TO g^.num_nodos - 1 DO
                        FOR cont2 := i TO g^.num_nodos - 1 DO
                              g^.matriz[cont, cont2] := g^.matriz[cont,
cont2 + 1]
                        END
                  END;
                  DEC(g^.num_nodos);
            END;
      END;
END BorrarNodo;

PROCEDURE BorrarArista(VAR g : GRAFO; n, m : NODO);
VAR
  i, j : [1..MAX];
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      ELSE
            i := Posicion(g, n);
```

```
            j := Posicion(g, m);
  (* Comprobaci¢n de que los nodos a conectar existen en el grafo *)
            IF (i = 0) OR (j = 0) THEN
                  Error_Grafo := Nodo_Inexistente;
            ELSE
                  g^.matriz[i, j] := FALSE;
                  g^.matriz[j, i] := FALSE;
            END;
      END;
END BorrarArista;

PROCEDURE Contiene(g : GRAFO; n : NODO) : BOOLEAN;
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      END;
      RETURN Posicion(n) <> 0;
END Contiene;

PROCEDURE SonAdyacentes(g : GRAFO; n, m : NODO) : BOOLEAN;
VAR
      i, j : [1..MAX];
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      ELSE
            i := Posicion(g, n);
            j := Posicion(g, m);
(* Comprobaci¢n de que los nodos a conectar existen en el grafo *)
            IF (i = 0) OR (j = 0) THEN
                  Error_Grafo := Nodo_Inexistente;
            END
      END;
      RETURN g^.matriz[i, j];
END SonAdyacentes;

PROCEDURE EsVacio(g : GRAFO) : BOOLEAN;
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      END;
      RETURN (g^.num_nodos = 0);
END EsVacio;

PROCEDURE Destruir(VAR g : GRAFO);
BEGIN
      Error_Grafo := No_Error;
      IF g = NIL THEN
            Error_Grafo := No_Creado;
      ELSE
            DISPOSE(g);
      END;
END Destruir;

END Grafo.
```

6.8 Ejercicios.

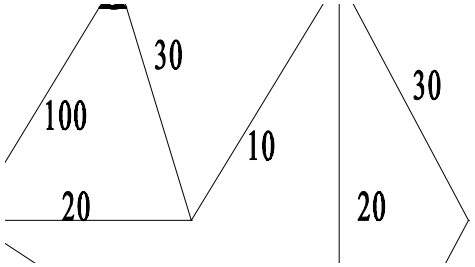
1.- Especificar la operación **LongitudCaminoMinimo: Item x Item x Grafo —/→ N**, que a partir de dos nodos de un grafo nos devuelve la longitud (medida en número de arcos) del camino más pequeño que los une.

2.- Especificar la operación **CaminoMinimo: Item x Item x Grafo ⟶ Lista**, que a partir de dos nodos de un grafo nos devuelve una lista con el camino más corto que los une.

3.- Especificar la operación **Caminos: Item x Item x Grafo ⟶ ListaDeListas**, que a partir de dos nodos de un grafo nos devuelve una lista con todos los caminos que los unen (hay que tener en cuenta que cada camino es, a su vez, una lista).

4.- Especificar la operación **ComponenteConexa: Item x Grafo ⟶ Grafo**, que a partir de un nodo **I** de un grafo **G** nos devuelve el subgrafo conexo de **G** más grande que contiene a **I**.

5.- Representar gráficamente la estructura del grafo de la figura, cuando se utiliza una implementación con:



a) Matriz de adyacencia.

b) Lista de adyacencia.

c) Multilista de adyacencia.

6.- (Franch 6.5) En un ordenador cualquiera, los números enteros y los punteros se representan con 32 bits, y los *booleanos* con un solo bit. Calcular el número mínimo de arcos que debe tener un grafo dirigido sin etiquetas, y con 100 nodos, para que su representación mediante matriz de adyacencia ocupe menos bits que mediante listas de adyacencia, suponiendo que los nodos se identifican mediante enteros.

Repetir el ejercicio suponiendo un grafo etiquetado, en el que las etiquetas son enteros.

7.- Especificar la operación **ListaCiclos: Grafo ⟶ ListaDeListas**, que a partir de un grafo no dirigido devuelve una lista con todos los ciclos simples. ¿Cuál es el máximo número de ciclos elementales que se pueden encontrar en un grafo de **n** nodos?

8.- Especificar la operación **ConexoN : Grafo x N ⟶ B** que indica si el grafo de partido contiene o no algún subgrafo con **n** elementos (el segundo parámetro) totalmente conexo.

9.- (Franch 6.27) La región montañosa de Putorana en el noreste de Siberia, es famosa en toda la estepa rusa por su sistema de comunicaciones estival, basado en canales que se ramifican. Todos los canales son navegables, y todas las poblaciones circundantes están situadas cerca de algún canal.

En el mes de julio, en pleno verano, estos canales se deshielan y su caudal facilita la navegación. En esta época, los alcaldes asisten a la reunión de la mancomunidad, que tiene lugar en la localidad de Magnitogorsk, y cada uno de ellos viaja en su propio barquito oficial,

dependiente del gobierno militar. Proponer a los alcaldes un algoritmo que les permita llegar a Magnitogorsk de manera que cada uno de ellos haya recorrido la mínima distancia posible.

10.- Siguiendo con el ejercicio anterior, y ante la crisis económica de toda la región, las milicias han decidido que los alcaldes compartan los barquitos, de manera que si, durante el trayecto, un barquito pasa por una población donde hay uno o más alcaldes, los recoge, formándose grupos que se dirigen a Magnitogorsk. Discutir la validez de la solución anterior. Si es necesario, proponer otra que reduzca la distancia recorrida por los barquitos (pero no necsariamente por los alcaldes).

11.- Especificar la operación **SubgrafosTotalmenteInconexos: Grafo** $\longrightarrow$ **ConjuntoDeSubgrafos** que, dado un grafo **G**, devuelve un conjunto con todos los subgrafos de **G**, formados exclusivamente por nodos, esto es, que no contienen ni un solo arco.

12.- A partir del ejercicio anterior, especificar la operación **Subgrafos: Grafo** $\longrightarrow$ **ConjuntoDeSubgrafos** que, dado un grafo **G**, devuelve un conjunto con todos los subgrafos de **G**.

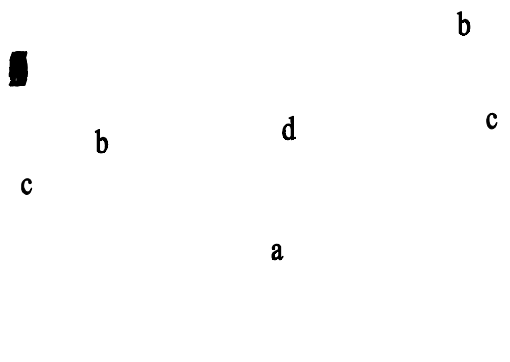
13.- Implementar los grafos mediante matrices de incidencias. Una matriz de incidencia es una tabla de doble entrada (n×a), en la que la casilla (n_i, a_j) es un **1** si el nodo n_i es uno de los extremos del arco a_j. Suponer que el número máximo de arcos es 3N donde N es el número de nodos.

14.- Diseñar un algoritmo para encontrar el camino de mayr peso entre dos nodos cualesquiera de un grafo etiquetado no dirigido y conexo.

15.- En un grafo dirigido, se llama nodo sumidero a todo tono que no posee arcos de salida, sino sólo arcos de llegada. Especificar una operación que, dado un grafo, devuelva un conjunto con todos sus nodos sumidero.

16.- El problema de los puentes de Königsberg. Dado un grafo no etiquetado y no dirigido, se pretende obtener un camino que pase por todos los nodos cuantas veces sea necesario, y que pase por todos los arcos una sola vez.

17.- Un grafo permite representar un autómata basado en estados, de manera que cada nodo representa un estado válido del autómata, y los arcos dirigidos representan las transiciones. Cada arco estará etiquetado con el valor de la entrada que se consume. Especificar un predicado que, dado un autómata finito en forma de grafo etiquetado y dirigido, un estado inicial, y una cadena de entrada en forma de lista, indique si la cadena es reconocida por el autómata o no. Se supone que tratamos con autómatas finitos deterministas. Por ejemplo, siguiendo el autómata de la figura, si partimos del estado 0, la entrada [a, a] es reconocida, pero la cadena [c, b, c] se rechaza.



18 .- A partir del ejercicio anterior, especificar una operación parcial que, dado un autómata finito en forma de grafo etiquetado y dirigido, un estado inicial, y una cadena de entrada en forma de lista, devuelva como resultado el estado final. Se supone que tratamos con autómatas finitos deterministas.

Por ejemplo, en la figura, si partimos del estado 0, y nos dan como entrada la lista [c, a, a, b], el estado final es el 4.

19.- Suponiendo que el autómata finito no es determinista (de un mismo nodo pueden salir varios arcos con la misma etiqueta), modificar la especificación anterior para que, en lugar de devolver el estado o nodo final, devuelva una lista con los posibles estados finales.

20.- Modificar las operaciones anteriores de manera que los estados que se devuelven sólo son válidos si son nodos sumidero, y no en caso contrario. Si una cadena de entrada no hace que se llegue a un estado sumidero, se supone que la cadena se rechaza.

El espacio de almacenamiento requerido es $O(n^2)$, independientemente del número de arcos del grafo. En grafos no dirigidos, las matrices de adyacencia son simétricas. Esto implica que es suficiente con $O(n^2/2)$ etiquetas para representar el grafo al poder guardar solamente los arcos donde el nodo origen sea menor o igual que el destino (es posible que un nodo se apunte a sí mismo).

La consulta de un arco es inmediata O(1), ya que sólo consiste en acceder al elemento correspondiente. La búsqueda de nodos adyacentes es O(n) y operar sobre todos los arcos es $O(n^2)$.

Supongamos que el elemento inicial es el nodo *a*. La primera llamada será entonces PrimeroEnProfundidad(G, a). Como *a* no ha sido visitado, se marca como visitado y se procesa su primer nodo adyacente, *b* en este caso. Iremos viendo el contenido de la pila para aclarar en lo posible la traza de la ejecución.

A la hora de procesar *b*, buscamos el primer nodo adyacente que esté sin marcar. Como *a* lo está, el único que queda es *e*. Pasamos pues a procesar el nodo *e*. El primer nodo adyacente a *e* sin marcar es *f*. Cuando procesamos *f*, el primer nodo accesible sin marcar es *d*, que es el próximo en procesar. Desde d podemos acceder a *c* que aún no ha sido visitado. Una vez en estamos procesando *c*, vemos que no tiene ningún nodo adyacente sin visitar (el único nodo adyacente es *d*, que es precisamente el nodo desde el que hemos accedido a *c*).

Es ahora cuando damos el primer paso atrás. Tras acabar el procesamiento de *c*, volvemos a acabar el del nodo *d*. Aún queda por visitar un nodo vecino de *d*, el nodo *g*. Con *g* tenemos la misma situación que con *c*, no hay ningún nodo adyacente, por lo que no avanzamos. Damos otro paso atrás y volvemos al nodo *d*, pero tampoco aquí queda ningún vecino sin marcar. El nodo anterior que se había quedado en mitad del proceso era *f*, al que tampoco le quedan nodos adyacentes que no hayan sido visitados, por lo que damos otro paso atrás al nodo *e*. Con el nodo *e* se repite la circunstancia y volvemos al nodo *b*, que está en idéntica situación. Sólo queda entonces acabar el procesamiento del nodo *a*, pero se nos encontramos con que el otro nodo adyacente a *a*, el *d*, ha sido ya visitado (entre el *f* y el *c*, concretamente) por lo que ya se ha acabado el recorrido, que habría sido: *a, b, e, f, d, c, g*.

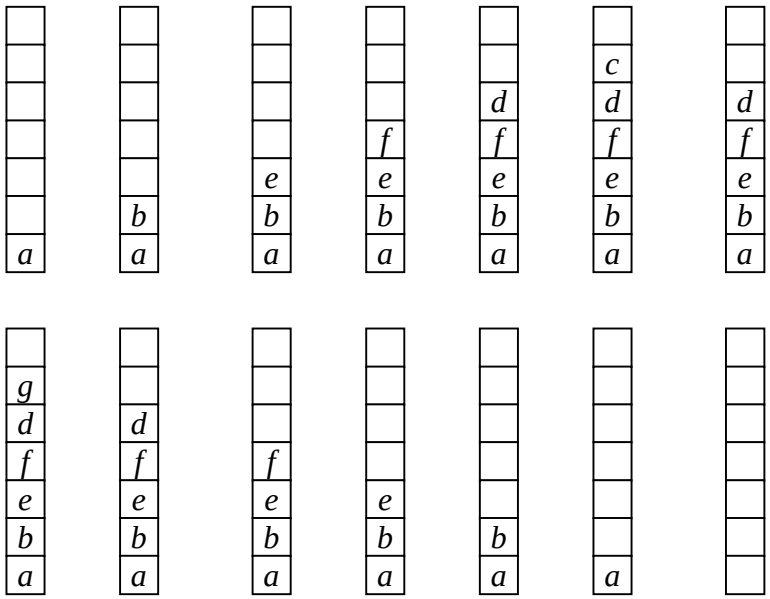


Figura 6. 16. Contenido de la pila en el recorrido primero en profundidad.

Demostración de la propiedad AAM de los árboles de extensión de coste mínimo. (Estructuras de Datos y Algoritmos. Aho, Hopcrof, Ullman. Página 234.