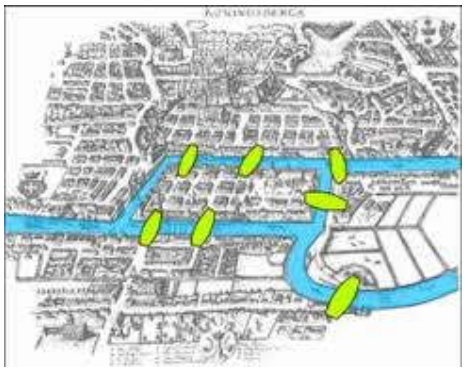
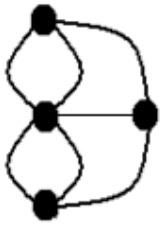
Grafos

El origen del concepto de grafo se remonta a la ciudad de Königsberg , en Prusia (actualmente, Kaliningrado, en Rusia). Por esa ciudad pasa el río Pregel y existen siete puentes, como muestra la figura. En épocas del matemático Euler, la gente del lugar se desafiaba a hacer un recorrido que permitiera atravesar cada puente una sola vez regresando al punto de partida.



Este problema, que tenía gran dificultad ya que nadie superaba el desafío, atrajo la atención de Euler, quien lo analizó empleando una técnica de graficado por la cual redujo a puntos la representación de las islas y las orillas, y a arcos los siete puentes.
El grafico resulto del siguiente modo. A los puntos se los denomino 'nodos' o 'vértices'. Los enlaces son 'aristas' o 'arcos'.



El problema original se volvió entonces ahora es análogo al de intentar dibujar el grafo anterior partiendo de un vértice, sin levantar el lápiz, sin pasar dos veces por el mismo arco y terminando en el mismo vértice donde se había comenzado.

Euler consideró que en un grafico de este tipo había nodos "intermedios" y de "inicio o finalización". Los nodos intermedios tienen un número par de aristas incidentes en ellos (ya que "si se llega" a uno de ellos se debe "volver a salir", puesto que son intermedios). Si hay un nodo de inicio y otro de finalización (es decir si el dibujo se comienza en uno y se termina en otro) ambos deben tener un

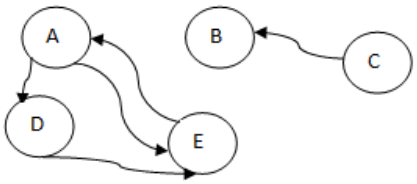
número impar de aristas incidentes en ellos, pero si el nodo de inicio coincide con el de finalización, el número de aristas incidentes en él debe ser par.

Analizando la situación de los puentes, se concluye que el problema de los puentes de Königsberg no tenía solución. Así comenzó la teoría de Grafos. (puede resultar interesante mirar este texto http://eulerarchive.maa.org//docs/originals/E053.pdf)

Los grafos constituyen una muy útil herramienta matemática para modelizar situaciones referidas a cuestiones tan diferentes como mapas de interrelación de datos, carreteras, cañerías, circuitos eléctricos, diagrama de dependencia, etc.

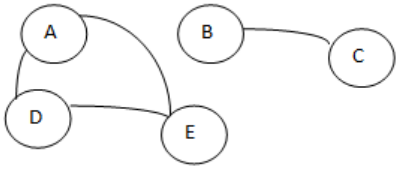
Definición matemática de Grafo

Un grafo dirigido o digrafo o grafo orientado consiste en una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares ordenados A (aristas orientadas) pertenecientes a VxV. (La relación establecida entre los vértices es antisimétrica)
En símbolos el grafo dirigido G es
G = (V ; A) donde A es un subconjunto de VxV (Aristas orientadas o dirigidas)
Ejemplo:



V={A, B, C, D, E}
A={ (A,D), (A,E), (E,A), (D,E), (C,B)}

Un grafo no dirigido (o no orientado) es una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares NO ordenados A (aristas no orientadas) pertenecientes a VxV. (La relación establecida entre los vértices es simétrica).
En símbolos el grafo dirigido G es
G = (V ; A) donde A es un conjunto de pares no ordenados de VxV (Aristas NO orientadas o NO dirigidas)
Ejemplo:



V={A, B, C, D, E}
A={ (A,D), (A,E), (D,E), (C,B)} (pares no ordenados)
En ambos casos, si (v,w) pertenece a A se dice que w es adyacente a v.

Algunas definiciones más:

Camino: serie alternada de vértices y aristas que inicia y finaliza con vértices y donde cada arista conecta el vértice que le precede con el que le sucede.

Longitud de Camino: Cantidad de Aristas del camino.

Camino Abierto: Camino donde el vértice inicial y final difieren.

Camino Cerrado: Camino donde el vértice inicial y final coinciden.

Recorrido: Camino que no repite aristas.

Ciclo: Camino Simple Cerrado (o bien, camino que conteniendo al menos tres vértices distintos tales que el primer vértice es adyacente al ultimo).

Árbol libre: es un grafo no dirigido conexo sin ciclos.

Grafo no dirigido conexo: un grafo no orientado es conexo sii para todo vértice del grafo hay un camino que lo conecte con otro vértice cualquiera del grafo

Grafo subyacente de un grafo: es el grafo no dirigido que se obtiene reemplazando cada arista (orientada) del mismo, por una arista no orientada.

Grafo dirigido fuertemente conexo: un grafo dirigido es fuertemente conexo sii entre cualquier par de vértices hay un camino que los une.

Grafo dirigido débilmente conexo: es aquel grafo dirigido que no es fuertemente conexo y cuyo grafo subyacente es conexo.

El TDA Grafo:

Es un TDA contenedor de un conjunto de datos llamados nodos y de un conjunto de aristas cada una de las cuales se determina mediante un par de nodos.

Crear grafo: esta primitiva genera un grafo vacío.
Precondición: ------
Poscondición: grafo generado vacío

Destruir grafo: esta primitiva destruye el grafo.
Precondición: que el grafo exista .
Poscondición: ------

Insertar nodo: esta primitiva inserta un nodo nuevo, recibido como argumento, en el grafo
Precondición: que el grafo exista y que el nodo no esté previamente
Poscondición: el grafo queda modificado por el agregado del nuevo nodo

Insertar arista: esta primitiva inserta una arista nueva, recibida como argumento, en el grafo
Precondición: que el grafo exista , que la arista no esté previamente y que existan en el grafo los nodos origen y destino de la arista.
Poscondición: el grafo queda modificado por el agregado de la nueva arista

Eliminar nodo: esta primitiva elimina un nodo, recibido como argumento, del grafo
Precondición: que el grafo exista y que el nodo a eliminar esté en él y no tenga aristas incidentes en él.
Poscondición: el grafo queda modificado por la eliminación del nodo

Eliminar arista: esta primitiva elimina una arista, recibida como argumento, del grafo
Precondición: que el grafo exista y la arista estén él.
Poscondición: el grafo queda modificado por la eliminación de la arista
Existe arista: esta primitiva recibe una arista y retorna un valor logico indicando si la arista existe en el grafo
Precondición: que el grafo exista
Poscondición: ------

Existe nodo: esta primitiva recibe una arista y retorna un valor logico indicando si el nodo existe en el grafo.
Precondición: que el grafo exista
Poscondición: ------

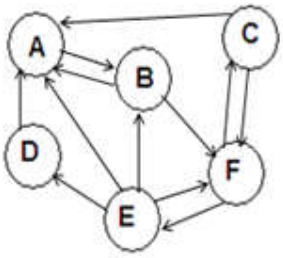
Pueden considerarse también las operaciones correspondientes a los recorridos como básicas en el TDA (o bien plantear las utilizando iteradores para navegar dentro del contenedor).

Estructuras de datos utilizadas para implementar grafos

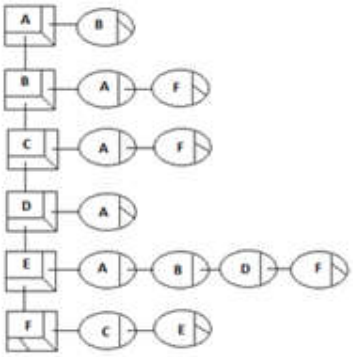
Se puede utilizar una Matriz de Adyacencia, la cual, para N nodos es de NxN.
Si M es la matriz de adyacencia del; grafo G entonces verifica que M[i][j] es true (o 1) si la arista (i,j) pertenece al grafo, o false (o bien 0) si no pertenece.

Otra posibilidad es usar listas de adyacencia. En este caso, la representación es mediante una lista de listas.
De este modo, se tiene una lista con nodos; cada uno de ellos conteniendo la información sobre un vértice y un puntero a una lista ligada a los vértices adyacentes al vértice indicado.

Ejemplo: para este grafo se muestra la correspondiente matriz de adyacencia y las listas de adyacencia.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	0	0	0	1
C	1	0	0	0	0	1
D	1	0	0	0	0	0
E	1	1	0	1	0	1
F	0	0	1	0	1	0



Recorridos de un grafo dirigido:

Los dos recorridos básicos en los grafos dirigidos son en profundidad y en anchura. Cada uno de ellos procesa o visita cada vértice del grafo, visitando una y solo una vez a cada uno.

Además, si se verifica que el grafo sea acíclico, se pueden llevar a cabo los recorridos topológicos.

Recorrido en profundidad (*depth first search* 0 'busqueda en profundidad',):

Consiste en, para cada vértice del grafo que no ha sido visitado previamente, visitarlo y luego pasar al primer adyacente no visitado, luego al primer adyacente del adyacente que no haya sido visitado, y así hasta llegar a un nodo que no tenga adyacentes no visitados. Entonces se retrocede, para pasar al siguiente adyacente del anterior vértice y realizar el mismo proceso hasta agotar los adyacentes no visitados (se retrocede cuando estos se agotan). Para implementar el algoritmo suele usarse una pila, o bien un algoritmo recursivo. Puede, además, numerarse los vértices a medida que se visitan, como se lleva a cabo en los algoritmos que se describen a continuación.

Recorrido en Profundidad recursivo con numeración de vértices

```
DFSNumerandoRec
{
Para cada vértice v
        {
        Marcar v como no visitado;
        Asignar 0 a numero de v;
        indice= 0;
        }

Para cada vértice v
        {
```

```
        Si (v no visitado) entonces
           {Num ( v, indice );}
         }
}

Num (u:vértice; var nd: entero )
{
   nd = nd+1;
   Asignar nd a número de u;
   Marcar u como visitado
   Para cada vértice w adyacente a u
          {
          Si (w no visitado )  entonces
              {Num( w, nd);}
                   }
  }
```

Análisis del coste temporal:

El coste depende básicamente de las estructuras que se usen para almacenar el grafo. Considerando un grafo de v vértices y a aristas, si se trabaja con una implementación de matriz de adyacencia, el ciclo que marca cada vértice como no visitado es O(v). Luego se ejecuta el ciclo que revisa si cada vértice ha sido o no visitado para invocar a Num. El ciclo externo tiene O(v) iteraciones y el coste del bloque depende del coste de Num.

Num comienza con unas sentencias O(1) y luego hay un ciclo que analiza cada adyacente a u para invocar a Num para cada adyacente que no haya sido visitado. Como para determinar los adyacentes a u hay que recorrer la fila correspondiente de la matriz, el ciclo tiene O(v) iteraciones, siendo n el número total de veces que se ejecuta Num. Entonces, el coste de la búsqueda en profundidad para el caso de una implementación con matriz de adyacencia pertenece a $O(v^2)$.

Si se ha implementado el grafo con listas de adyacencia, marcar cada vértice como no visitado es O(v). Num se invoca para cada vértice, en el peor caso, tantas veces como adyacentes tenga el vértice, por tanto, el ciclo que analiza cada vértice w adyacente a u realiza en total 2*a iteraciones. Entonces tenemos que el coste se puede expresar como O(v) + O(a), lo que generalmente se indica como O(v+a).

Recorrido en Profundidad iterativo con numeración de vértices:

```
DFSNumerandoIt
{
        Inicializar pila p a vacío;
        indice = 0;
```

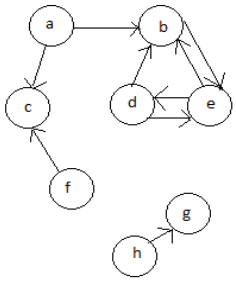
```
    Para cada vértice v
            {
       Marcar v como no visitado;
       Asignar 0 a numero de v;
       p.Apilar (v);
            }
    Mientras pila p no vacía
    {
       u:=cima(p);
       p:=desapilar(p);
       Si (u no visitado)
                {
                  marcar u como visitado
                  índice= índice +1;
                   asignar índice a numero de u;
                   Para cada w adyacente a u
                         Si (w no visitado)
                                {p. Apilar(w);}
                }
     }
}
```

Análisis del coste temporal:

Si se trabaja con una implementación de matriz de adyacencia, y considerando O(1) el coste de apilar y desapilar, el primer ciclo es O(v). Luego se tiene un ciclo que se ejecuta mientras no se haya vaciado la pila. En ese ciclo, si el vértice desapilado no se ha visitado (lo cual sucederá v veces en total), se lo numera y se analizan sus adyacentes (a lo sumo v-1) para apilarlos si no han sido visitados aún. De lo cual se deduce un coste temporal $O(v^2)$ para el recorrido con la implementación indicada.

Con una implementación con listas de adyacencia, el primer ciclo tiene coste O(v). Luego se tiene, para cada vértice desapilado y no visitado, el análisis de la situación de sus adyacentes, para establecer si se visitaron o no. En total esto se realiza O(a) de veces.
Entonces, el coste temporal total del recorrido en profundidad iterativo con implementación en listas de adyacencia es O(v +a).

Ejemplo: para el siguiente grafo se muestra el recorrido en profundidad:



a	b	e	d	c	f	g	h
1	2	3	4	5	6	7	8

Bosque asociado al Recorrido en Profundidad (aplicación del recorrido en profundidad).

Al ejecutar un recorrido en profundidad en un grafo, G = (V, E) se puede obtener un bosque de ejecución. La raíz de cada árbol de este bosque es cada uno de los vértices por los cuales se ha comenzado un recorrido parcial. Las aristas que aparezcan en cada árbol del bosque enlazan vértices visitados con el que se visitó a continuación.
A continuación un algoritmo, variante del recorrido en profundidad, que permite obtener el bosque asociado a un grafo dirigido.

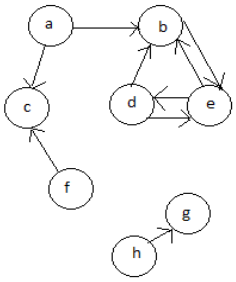
```
Bosque-DFS
{
  Para cada vértice v
        {
         Marcar v como no visitado
        }
 Inicializar el bosque B como vacío
 Para cada vértice v
       {
         Si (v no visitado)
              {
                 Inicializar el árbol T como vacío;
                 ArbolDFS( v, T);
                 Agregar T al bosque B;
                 }
        }
retornar  B
}

ArbolDFS ( u:vértice; var T:árbol)
{
        Marcar  u como visitado;
        Agregar vértice al árbol T;
```

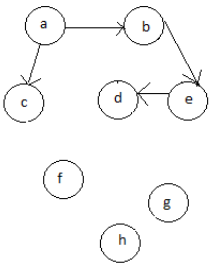
```
        Para cada w adyacente a u
          {
          Si (w no visitado)
            {
               ArbolDFS(w,T)
               Agregar arista (u, w) a T;
                }
          }
        retornar T
}
```

Análisis del coste temporal:

Si se asume un coste O(1) para las operaciones de Inicializar Árbol, inicializar Bosque, Agregar vértice al árbol y Agregar arista al Árbol, la generación del bosque asociado al recorrido en profundidad tendrá un coste análogo al del recorrido en profundidad según se haya implementado el grafo con matriz de adyacencia o con listas de adyacencia.

Ejemplo: para el grafo de la izquierda se muestra a la derecha el bosque asociado al recorrido en profundidad.





Test de aciclidad: (aplicación del recorrido en profundidad).

A menudo es necesario determinar si un grafo tiene o no ciclos (de no tenerlos se lo denomina acíclico). Para averiguarlo es posible usar el recorrido en profundidad, y detectar, si hay "aristas dde retroceso", que conecten un vértice con otro que aparece como adyacente, pero que ya ha sido visitado antes en el recorrido parcial que se está realizando.
Dado G= (V,E) grafo dirigido y TDFS árbol del recorrido en profundidad de *G*, decimos que *G* contiene un ciclo si y sólo si *G* contiene al menos una arista de retroceso.

```
TestAciclidad
{
  Inicializar b en falso; //valor lógico a retornar
```

```
Para cada vértice v
            {
               Marcar v como no visitado;
               v.encaminoactual = falso ;
             }

 Para cada vértice v
              {
                   Si (b== falso) y (v no visitado)
                     {
                        AnalizarRecParcial( v,  b);
                     }
                  Retornar b;
               }

AnalizarRecParcial ( u :vértice; var b: bool )
 {
   Marcar u como visitado;
   u.encaminoactual = verdadero;
   Para cada w adyacente a u
         {
          Si (b==falso)
              {
                Si    (w visitado) y (w.encaminoactual ==verdadero)
                  {b = verdadero;}
                Si  (w no visitado )
                   {AnalizarRecParcial ( w,b ); }
               }
          }
     u.encaminoactual = falso;
 }
```

Análisis del coste temporal:

Los costes son los mismos que se han discutido para el recorrido en profundidad, según la implementación que se haya hecho para el grafo.

Obtención de los puntos de articulación de un grafo: (aplicación del recorrido en profundidad).

Un *punto de articulación o vértice de corte* de un grafo no dirigido y conexo es un vértice tal que al ser eliminado el grafo se deja de ser conexo. Si un grafo no tiene puntos de articulación significa que

para todo par de vértices existe más de un camino que los enlaza. Entonces, si un vértice del grafo "fallara" (es decir, si hubiera que considerarlo como retirado del grafo por algún motivo) el grafo permanecería aún conexo. A los grafos sin puntos de articulación se los llama biconexos.

Definiciones:
Punto de articulación: dado G=(V,A), grafo no dirigido conexo, v∈V es un punto de articulación sii el subgrafo G' = (V-{v}, A') , no es conexo, siendo A' = A – {(s,t)/ (s,t) ∈ A y (s=v) o (u=v) }
Grafo biconexo: grafo conexo y sin puntos de articulación (aunque "falle" un vértice, se mantiene la conectividad).

Ahora bien, si consideramos un grafo no dirigido y conexo, el bosque generado en un DFS, y determinadas las aristas de retroceso,

- Si r es raíz es el árbol DFS y tiene más de un hijo en el árbol, entonces r es un punto de articulación.
- Para todo vértice u que no sea raíz en árbol DFS, u es punto de articulación si al eliminarlo del árbol resulta imposible "volver" desde alguno de sus descendientes (en el árbol) hasta alguno de los antecesores de u.

Para analizar el caso de los vértices que no son raíz, establecemos para cada uno cáal el valor más bajo (orden de un nodo en recorrido) entre el número de orden del nodo, el que al que se puede acceder desde él "bajando" arcos en el árbol generado con el recorrido, y el de aquel al que se acceda "subiendo" a través de alguna arista de retroceso

bajo(u) = mínimo número asignado en el recorrido en profundidad considerando estos valores:

- numdfs del nodo u
- numdfs de cada nodo w al que se accede desde u por una arista de retroceso (es decir (u,v) es una arista que no está en el árbol que se genera durante el recorrido).
- bajo (x) para todo x hijo de u en el árbol que se genera en el recorrido.

Es decir,
bajo(u) = Minimo(numdfs(u), numdfs(w)/ ∀*w*∈V tal que (u,w)∈A y (u,w) ∉ árbol DFS (salimos de *u* subiendo por una arista de retroceso back), bajo(x) /∀*x*∈hijos(u, T_{DFS}))

Entonces el vértice u, si no es raíz, es punto de articulación sii tiene al menos un hijo x tal que bajo(x)≥numdfs(u).

A continuación el pseudocódigo:

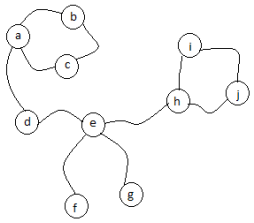
Puntos de Articulación
{// g es grafo no dirigido conexo
// se retorna lis, lista de vértices que son puntos de articulación
Para cada vértice v
{

```
          Marcar v como no visitado;
           v.ndfs := 0 ; //inicializa en 0 en la numeración a v
           v.bajo := 0; //inicializa bajo de v a 0
          }
indice := 0;
inicializar lis como vacía;
PuntoArticRec( v, indice, lis, -1 );
//el ultimo argumento es un valor que representará a
 //un vértice a analizar en función de v;
//inicialmente se pasa un valor no válido, a modo de "nulo"
 Retornar lis;
  }

PuntoArticRec
(
  v es vértice; var i : entero; var lis: lista;  p : vértice )
   i= i +1;
   v.ndfs := i;
   v.masbajo := v.ndfs;
  Marcar v como visitado;
   espuntoartic := falso;
   numerohijos:= 0; // corresponde al numero de hijos procesados
   Para cada w adyacente a v
          {
            Si  (w visitado) ∧ (w ≠ p)
                {v.masbajo := min (v.masbajo,w.ndfs);}
              Si  (w no visitado)
                 {
                  PuntoArticRec( w, i, l, v );
                  v.ma := min (v.masbajo, w.masbajo);
                 }
    Si    (p == null)
         {numerohijos := numerohijos +1;}
    Si no
         { Si (w.masbajo ≥ v.ndfs)
             {espuntoartic := verdadero;}
          }
     Si   (p == null )∧ (numerohijos > 1 )
           {lis.agregar(v);}
     Si   (p != null) ∧  (espuntoartic)
          { lis.agregar.(v);}
}
```

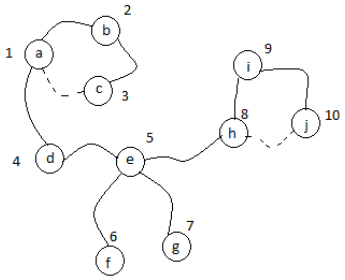
Análisis del coste temporal:

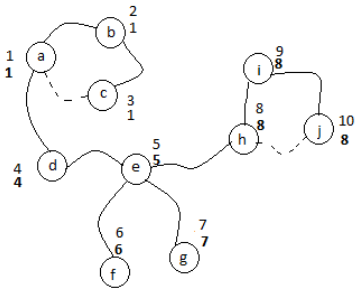
Como el algoritmo utiliza una búsqueda en profundidad a la cual incorpora sentencias y bloques O(1), el coste temporal del algoritmo es análogo al de la búsqueda en profundidad según el tipo de implementación que se haya hecho del grafo.
Ejemplo:

Obtención de los puntos de articulación del siguiente grafo:



Se realiza el recorrido, obteniéndose la numeración según el gráfico de la izquierda.
Luego se calcula bajo(v) para cada v vértice, comenzando por el de valor más alto, obteniéndose la información del gráfico de la derecha (los valores de bajo están en negrita).





Vértices padres e hijos en árboles generados durante el DFS:

Vértice	Hijos en el árbol del recorrido
a	b, d
b	c
c	----
d	e
e	f, g, h
f	---
g	---
h	i
i	j
j	---

Puntos de articulación:

a, por ser raíz y tener dos hijos en el árbol
d, porque bajo de e es 5 y numero de d es 4
e porque bajo de h es 8 y numero de e es 5,
porque bajo de g es 7 y numero de e es 5,
porque bajo de f es 6 y numero de e es 5,
(es suficiente que suceda con uno de los hijos)
h, porque bajo de i es 8 y numero de h es 8

<u>Obtención de las componentes fuertemente conexas en un grafo dirigido</u>: (aplicación del recorrido en profundidad).

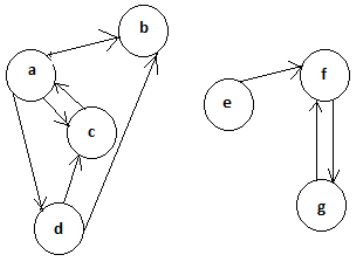
Una *componente fuertemente conexa* (CFC) de un grafo es un conjunto maximal de vértices tal que el subgrafo formado es fuertemente conexo.
Las componentes conexas para un grafo G(V,A)pueden obtenerse de la siguiente forma:

1. Realizar un recorrido DFS(G) etiquetando los vértices con la numeración en el orden inverso del recorrido (invnum).
2. Obtener el grafo transverso G^T invirtiendo el sentido de las aristas de G.
3. Realizar un recorrido DFS(G^T) en orden decreciente de los valores invnum de los vértices. Cada árbol obtenido es una componente fuertemente conexa de G.

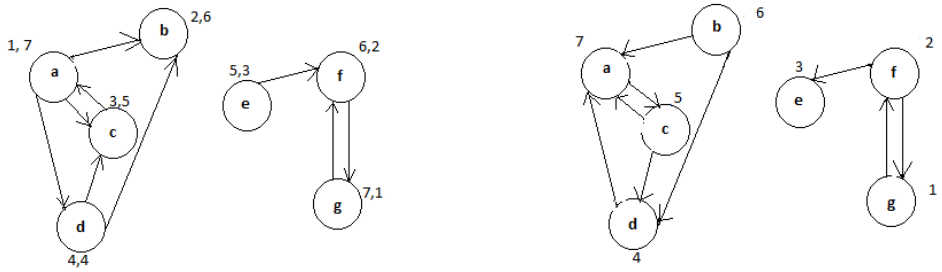
<u>Análisis del coste temporal:</u>

Se deja como ejercicio el análisis del coste.

Ejemplo:
Obtención de las componentes fuertemente conexas del siguiente grafo:



A la izquierda, el grafo recorrido. Para cada nodo se indica un par de valores: el número de recorrido en orden directo, y en el orden inverso.
A la derecha, el grafo invertido con la numeración en orden inverso.



Al recorrer el grafo invertido considerando los vértices en orden inverso, se obtienen las componentes conexas del grafo, que en este caso son:

Componente 1: a, c, d
Componente 2: b
Componente 3: e
Componente 4: f, g

Recorrido en anchura (*breadth first search, o ´búsqueda primero en anchura´*):

Este recorrido marca inicialmente todos los nodos como no visitados, y luego, para cada nodo no visitado, lo visita y marca, procediendo luego a visitar cada uno de sus adyacentes no visitados antes, luego de lo cual se continúa con los adyacentes del primer adyacente, los del segundo adyacente, etc., hasta agotar el conjunto de vértices del conjunto.
Se presenta debajo el algoritmo que recorre en anchura numerando los vértices en el orden en que se visitan.

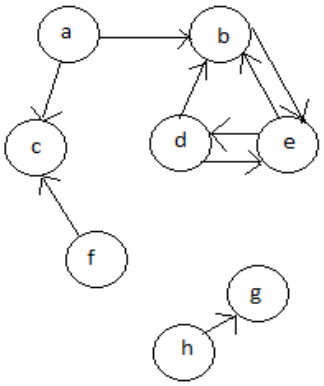
```
BFS
{
 Para todo vértice v
   { Asignar a v numero 0}
 índice=1;
 Incializar cola c;
 Mientras exista un vértice v tal que numero de v sea 0
    {
      Asignar a v numero índice;
      Incrementar índice;
      c. Acolar (v);
```

```
    Mientras cola no vacía
     {
       v= c. Desacolar();
       Para cada vértice u adyacente a v
         {
           Si (numero de u es 0)
            {
              Asignar a número de u el valor de índice
              Incrementar índice
              c.Acolar(u);
            }
         }
      }
  }
}
```

Análisis del coste temporal:

El coste depende de cómo se haya implementado le grafo. Si se lo implementó con una matriz de adyacencia, tenemos lo siguiente: el coste de marcar cada nodo como no visitado es O(v). Luego hay un ciclo de O(v) iteraciones, en cada una de las cuales se analizan los adyacentes del vértice que se desacola (como la implementación es con matriz de adyacencia, esto tiene un coste O(v)). Por lo cual el coste del recorrido en anchura con implementación de matriz de adyacencia es $O(v^2)$.
Si la implementación es con listas de adyacencia, hay que considerar el coste de marcar cada nodo como no visitado, que es O(v), y luego observar que en total tantas veces como aristas haya se va a analizar si el vértice adyacente al que se ha desacolado se ha visitado, para acolarlo si aún no se lo visitó. Entonces, el coste es O(v) +O(a), indicado como O(v+a).

Ejemplo de recorrido en anchura: se muestra el recorrido en anchura del siguiente grafo.



a	b	c	e	d	f	g	h
1	2	3	4	5	6	7	8

Recorridos topológicos:

Estos recorridos solo se aplican a grafos dirigidos acíclicos, y permiten linealizar un grafo, es decir, recorrer los vértices del mismo respetando las precedencias. Pueden plantearse recorridos topológicos como variantes del recorrido en profundidad y del recorrido en anchura.

Recorrido topológico en profundidad

```
//devuelve una lista en la cual quedan insertados los nodos según fueron visitados en el recorrido)
TopolDFS
{
Inicializar lista de salida;  //lista vacía
Para cada vértice v
   {marcarlo como no visitado }
Para cada vértice v
   { si (v no visitado) entonces
         {Rec (v, lista)
retornar lista
}

Rec ( v: vertice , var lis: lista)
{
Marcar v como visitado
Para cada vértice w adyacente a v
     { Si   (w no visitado) entonces
           {Rec(w, lista)}
  Insertar v al frente en lis
}
```

Análisis de coste temporal:

Análogo al del recorrido en profundidad habitual, según la implementación usada.

Recorrido topológico en anchura

Se comienza calculando , para cada vértice, el grado de entrada del mismo (número de aristas incidentes en el vértice). Los vértices de grado de entrada 0 se acolan (tiene que haber vértices con grado de entrada 0; de no ser así, el grafo tendría ciclos).
Luego se procesa la cola del siguiente modo: mientras no esté vacía se desacola y visita un vértice (y se dirigirá a la salida, si queremos el listado respetando precedencias) y para cada vértice adyacente al mismo se decrementa en 1 el grado de entrada del mismo (como si se retirara el vértice desacolado del grafo). Cada vértice que llegue a 0 por este decremento, se acola.

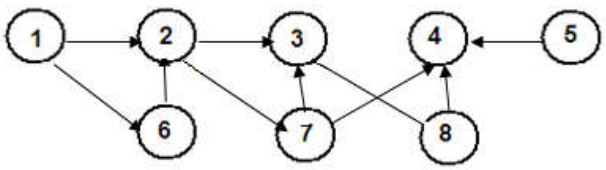
```
TopolBFS     //genera lista de salida, con todos los vértices del grafo
{
 Inicializar T;   //tabla de grados de entrada, de N posiciones, siendo n los vértices.
 Inicializar cola c; // cola para vértices del grafo; se inicializa vacía.
 Inicializar lis, lista de salida;  //se inicializa vacía.
 Para cada vértice v
   {
     T[v] = grado de entrada de v;
      //se registra el grado de entrada de cada vértice del grafo.
      //Siempre hay al menos un vértice de grado 0; de otro modo, el grafo no seria acíclico
     Si (grado de entrada de v ==0)
        {  c. acolar (v);}
     }
  Mientras  (cola c no vacía)
    {
      x=c.Desacolar();
      lis. Insertar al final (x);
      Para cada vértice w adyacente a v
         {
           T[w] = T[w] -1;
            si (T[w] == 0)
                { c. Acolar (w);}
          }
     }
 retornar lis;
}
```

Análisis de coste temporal:

Se deja como ejercicio el análisis del coste para diversas implementaciones.

Ejemplo:

Consideremos el siguiente grafo dirigido y acíclico:



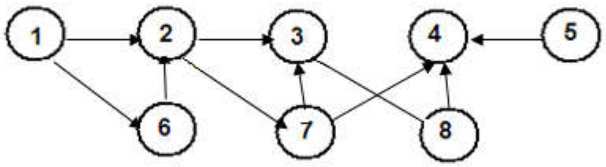
Según el algoritmo anterior,

```
{
rec(1)
Adyacentes(1):2, 6
  {
   rec(2)
   Adyacentes(2):3, 7
    {
     rec(3)
     Adyacentes(3):8
       {
       rec(8)
       Adyacentes(8):4
          {
            rec(4)
            Adyacentes(4):-
            Insertar 4 al frente en la lista de salida
          }
       Insertar 8 al frente en la lista de salida
           }
    Insertar 3 al frente en la lista de salida
      }
      {
     rec(7)
     Adyacentes(7): 3, 4 (ambos ya visitados)
     Insertar 7 al frente en la lista de salida
     }
    Insertar 2 al frente en la lista de salida
   }
   {
  rec(6)
  Adyacentes(6): 2 (ya visitado)
  Insertar 6 al frente en la lista de salida
   }
 Insertar 1 al frente en la lista de salida
 }
 {
 rec(5)
 Adyacentes(5): 4 (ya visitado)
 Insertar 5 al frente en la lista de salida
 }
```

De este modo, la lista generada que respeta las precedencias entre los nodos con un recorrido en profundidad es:

5, 1, 6, 2, 7, 3, 8, 4

Ejemplo de recorrido topológico en anchura:
Para el grafo anterior,



La tabla de grados de entrada es:

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	2	2	3	0	1	1	1

Se acolan el 1 y el 5
Cola : 1 – 5
Se desacola y procesa el 1
Se decrementan los adyacentes a 1, que son 2 y 6.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	1	2	3	0	0	1	1

6 ahora vale 0 y se acola.
Cola: 5 – 6
Se desacola y procesa 5.
Se decrementan los adyacentes a 5, que es 4.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	1	2	2	0	0	1	1

Cola: 6
Se desacola y procesa 6.
Se decrementa el adyacente a 6, que es 2.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	2	2	0	0	1	1

2 vale ahora 0 y se acola.
Cola: 2
Se desacola y procesa 2.
Se decrementan los adyacentes a 2, que son 3 y 7.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	1	2	0	0	0	1

7 llego a 0, se acola.
Cola: 7
Se desacola y procesa 7
Se decrementan los adyacentes a 7, que son 3 y 4

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	0	1	0	0	0	1

3 llego a 0, se acola
Cola: 3
Desacolar 3
Decrementar el adyacente a 3, que es 8

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	0	1	0	0	0	0

8 vale 0, se acola
Cola: 8
Desacolar 8
Decrementar el adyacente a 8, que es 4

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	0	0	0	0	0	0

4 se acola porque llego a 0.
Cola: 4
Desacolar y procesar 4, que no tiene adyacentes.

La linealización del grafo da por resultado esta lista (se han indicado las precedencias con arcos)

1-5- 6- 2- 7- 3- 8- 4

Ejercicios propuestos:

1. Determinar las componentes conexas del siguiente grafo.
2. Determinar los puntos de articulación del siguiente grafo
3. Desarrollar un algorimo que, para un par de vértices (x, y), determine la mínima cantidad de aristas del camino que enlace x con y (si existe).

4. Diseñar un algoritmo que permita determinar, para un grafo (dirigido o no) y con ciclos,la cantidad de vértices del camino más largo contenido en el grafo.

Problemas sobre caminos en un grafo:

Hay un gran número de problemas sobre caminos en grafos dirigidos y en grafos no dirigidos. Algunos plantean determinar si dado un par de vértices, o todos los pares de vértices, hay o no camino entre ellos.

Otros trabajan sobre grafos con aristas o con vértices ponderados. Una ponderación es un valor asociado a una arista o a un vértice, o a ambos.

En esta sección trataremos algunos de los problemas más comunes sobre caminos en grafos con aristas ponderadas.

Problema de los caminos más cortos con un solo origen

Dado un grafo dirigido g = (v, a), en el cual cada arco tiene asociado un costo no negativo, y donde un vértice se considera como origen, el problema de los "caminos más cortos con un solo origen" consiste en determinar el costo del camino más corto desde el vértice considerado origen a todos los otros vértices de v. Este es uno de los problemas más comunes que se plantean para los grafos dirigidos con aristas ponderadas (es decir con peso en las aristas). la 'longitud' o 'costo' de un camino es la sumatoria de los pesos de las aristas que lo conforman.

El modelo de grafo dirigido con aristas ponderadas no negativas puede, por ejemplo, corresponder a un mapa de vuelos en el cual cada vértice represente una ciudad, y cada arista (v,w) una ruta aérea de la ciudad v a la ciudad w.

Algoritmo de Dijkstra para el cálculo de los caminos mínimos:

Desarrollado por Dijkstra en 1959 (acá la primera publicación sobre el tema http://wwwm3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf)
Este algoritmo se caracteriza por usar una estrategia *greedy*, voraz (o ávida). Para aplicar este algoritmo el peso de las aristas debe no negativo.
En esta estrategia se trata de optimizar (alcanzar el máximo o el mínimo) de una función objetivo, la cual depende de ciertas variables, cada una de las cuales tiene un determinado dominio y está sujeta a restricciones.En cada etapa se toma una decisión que no tiene vuelta atrás, y que involucra elegir el elemento más promisorio (es decir, el que parece ofrecer más posibilidades de mejorar la función objetivo) de un conjunto y se analizan ciertas restricciones asociadas a cada variable de la función objetivo para ver si se verifican mejoras en ella.
La solución puede expresarse como una sucesión de decisiones, y en cada etapa de la estrategia se elegirá la mejor opción de las disponibles (óptimo local), analizándose luego si esta elección verifica las restricciones, constituyendo una solución factible.
La estrategia greedy, muy interesante en sí, no sirve para cualquier problema: se debe tratar de un

problema de optimización; pero aún en este caso, no asegura que se llegue al optimo global (podría llevarnos a un óptimo local) ni tampoco, en algunos casos, llegar a una solución factible: el problema en cuestión debe verificar ciertas condiciones para que quede garantizado que la estrategia greedy funcione (estas condiciones se verifican para el problemas de los caminos mínimos con origen dado).
Dado un grafo dirigido G = (V, A), con aristas ponderadas, y considerando un vértice como origen, determinar los costos del camino mínimos desde el vértice considerado origen a todos los otros vértices de V.

Resolución de Dijkstra utilizando una matriz de pesos y una tabla de vértices visitados:

Se tiene un grafo dirigido G = (V, A) con aristas no negativas
Consideraremos los vértices etiquetados 1..n
El vértice de partida será 1.
S será el conjunto de vértices visitados a lo largo del procesamiento.

```
CamMinDijk
{
// D es un vector de n-1 elementos para calcular los valores de los respectivos caminos mínimos
Inicializar S = { 1 }
Para cada vértice i desde 2 hasta N hacer
{ D[i] = coste inicial de alcanzar i desde el vértice 1
//corresponde al peso de la arista (1, i ), si existe, o bien un peso máximo
}
Mientras (conjunto de vértices no visitados V-S no esté vacío)
{
Elegir vértice w perteneciente a V – S tal que D[w] sea mínimo .
S= S U {w}
{
D[v]=mín ( D[v], D[w] + M[w,v])
}
}
}
```

Consideraciones sobre el coste temporal:

Si se utiliza una matriz de adyacencia y una tabla para almacenar los vértices y sus marcas de visitados, resulta:

O(v) la carga inicial de D y la marca de no visitado para los n-1 vértices candidatos.

El ciclo que se ejecuta mientras haya candidatos, es obviamente O(v).

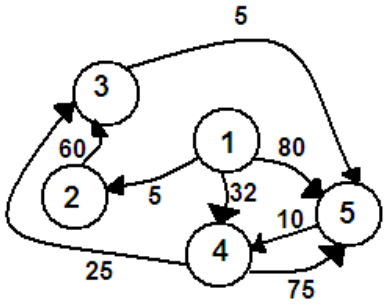
La función de elección de un vértice de coste mínimo no visitado implica recorrer tanto la tabla de visitados como el vector D, es O(v).

El análisis de cada adyacente al nodo elegido, por estar utilizándose una matriz de pesos, implica recorrer la fila correspondiente al vértice elegido, es decir O(v).

Aplicando las conocidas reglas de la suma y del producto, se llega a que el algoritmo es $O(v^2)$.

Ejemplo:

Consideraremos un grafo y los valores asociados a los arcos los presentaremos en una tabla; pero hay que recordar que la implementación del grafo podría tener estos datos en una matriz o en listas de adyacencia. Usaremos la primera versión del algoritmo (sin cola de prioridad para los vértices no visitados).



Si usamos una matriz de pesos M, es:

	1	2	3	4	5
1	0	5	∞	32	80
2	∞	0	60	∞	∞
3	∞	∞	0	∞	5
4	∞	∞	25	0	75
5	∞	∞	∞	10	0

Se ha colocado el símbolo ∞ en aquellas celdas que corresponden a celdas para las cuales no hay aristas asociadas.

Consideraremos además el vector de vértices visitados, en el cual ya hemos marcado el vértice 1 por ser de salida.

1	2	3	4	5
+	-	-	-	-

Además, tenemos el vector D en el cual vamos a trabajar para obtener los tiempos mínimos.

Inicialmente el vector D es:

2	3	4	5
5	∞	32	80

Etapa 1:

Se selecciona, de los vértices no visitados, aquel cuyo tiempo es el menor, evaluando los tiempos de D. El vértice con menor tiempo es 2.

Se marca entonces 2 como visitado en el vector de marcas.

Ahora se analizara si se mejoran los tiempos pasando por 2 como vértice intermedio.

Sólo el vértice 3 es alcanzable desde 2.

Comparamos el valor de D[3], que es ∞, con D[2] + M[2][3], que es 5+60=65

D[3] > D[2] + M[2][3] por lo tanto, reemplazamos en D[3] , ∞ por 65.

Los vertices 4 y 5 no son alcanzables desde 2, sus valores quedan iguales por ahora.

Entonces, finalizada esta etapa, se tiene:

Vector de vértices visitados:

1	2	3	4	5
+	+	-	-	-

Vector D:

2	3	4	5
5	65	32	80

Etapa 2:

Seleccionamos el vértice no visitado de menor tiempo. Es el 4, con un tiempo de 32.

3 y 5 son alcanzables desde 4.

D[3] = 63 y D[4] + M[4][3] =32+25=57, mejora el vértice 3

D[5] = 80 y D[4] + M[4][5] =32+75=107, no mejora el vértice 5

Finalizada esta etapa, se tiene:

Vector de vértices visitados:

1	2	3	4	5
+	+	-	+	-

Vector D:

2	3	4	5
5	57	32	80

Etapa 3 :

De los vértices no visitados, el de menor tiempo es 3

Desde 3 el único vértice alcanzable es 5.

D[5]=80 y D[3]+M[3][5]=57+5=62, proporcionando una mejora para 5

Vector de vértices visitados:

1	2	3	4	5
+	+	+	+	-

Vector D:

2	3	4	5
5	57	32	62

Etapa 4:

Queda solo el vértice 5 por analizar; hechos los cálculos se observa que solo el vértice 4 es alcanzable desde ahí, sin apreciarse mejora alguna.

Entonces, los valores de los caminos mínimos son:

2	3	4	5
5	57	32	62

Variante más eficiente del algoritmo de Dijkstra, con listas de adyacencia y cola con prioridades:

Si se usa una lista de adyacencia para la implementación del grafo, y una cola con prioridades para almacenar los nodos aún no elegidos con el coste del camino desde el origen hasta él (esta sería la prioridad), puede mejorarse el coste temporal.

Inicialmente se construye el heap de vértices con sus costes iniciales y se inicializa el vector D.

Mientras haya candidatos, se elige al mejor (el primero de la cola con prioridades) y se restaura el heap. Para cada adyacente al elegido (lo cual se realiza usando la correspondiente lista de adyacentes) se analiza si hay una mejora, de haberla se registrar y en ese caso, si el vértice mejorado está en el heap, se modifica su prioridad en el heap y se restaura el mismo.

Entonces, quedaría

```
CamMinDijks
{
  Inicializar heap h con todos los vértices del grafo excepto el origen; la prioridad es el peso de la
arista (1, i), si ésta     existe, o bien un peso máximo.

 Para cada vértice  i desde 2 hasta N hacer
                { D[i]  = coste inicial de alcanzar i desde el vértice 1
                  //corresponde al peso de la arista (1, i ), si existe, o bien un peso máximo
                 }
  Mientras (heap  h no vacío)
     {
       x= h. Remover_raiz();
      Para cada vértice w adyacente a x
         {
            D[v]=mín ( D[v], D[w] + M[w,v])
               Si (w fue mejorado) y  (w pertenece al heap)
                  {  h.ActualizarValor(w);}
          }
      }
```

Análisis del coste temporal:

El armado inicial del heap se puede hacer cargando los costos de los vértices (2..n) en un array y construyendo un heap sobre él, lo cual puede lograrse en O(v).
Mientras el heap no se haya vaciado, se elimina la raíz y se restaura el heap. En total esto se hará v veces, a un costo total O(v * log v).
Para cada adyacente al elegido puede tener que modificarse, en el peor caso, las prioridades de todos los vértices que estén en el heap, el cual deberá ser restaurado. Esto tiene un coste de O(a * log v).

Ahora bien, hay que tener en cuenta que la operación de determinar si el vértice cuyo costo se modifica está en el heap y en qué posición del heap se encuentra, tiene un coste asociado. Si este coste fuera O(1), entonces, el costo total es: O(v) + O(v * log v) +O(a * log v), es decir O((a+v) log v).

Problema de los caminos más cortos entre todos los pares de vértices.

Se trata de determinar los caminos mínimos que unen cada vértice del grafo con todos los otros. Si bien lo más común es que se aplique a grafos que no tengan aristas negativas, la restricción necesaria es que no existan ciclos con costos negativos.

Este problema no verifica las condiciones que aseguran que un algoritmo "greedy" funciones, por lo cual no puede aplicarse esa estrategia. Sin embargo, verifica las condiciones que permiten aplicar un algoritmo de los llamados de "programación dinámica"

La programación dinámica suele aplicarse a problemas de optimización. También considera que a la solución se llega a través de una secuencia de decisiones, las cuales deben verificarla condición de que "en una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima" (lo cual, dicho sea de paso, se verificaba también en la estrategia "greeddy"). En estos problemas se realiza una división de problemas en otros problemas menores, pero, a diferencia de lo que ocurría en "Divide y Vencerás", estos problemas resultado de la división no son independientes entre si, sino que tienen subproblemas en común, hay un 'solapamiento' de problemas. La técnica consiste en resolver y almacenar las soluciones de las zonas solapadas para no volver a realizar los mismos cálculos. En general en esta estrategia se llega a la solución realizando comparaciones y actualizaciones en datos que han sido tabulados (las soluciones de los subproblemas).

Algoritmo de Floyd

Este algoritmo desarrollado en 1962 (https://dl.acm.org/citation.cfm?doid=367766.368168) se basa en una técnica de programación dinámica que almacena en cada iteración el mejor camino entre el que pasa por el nodo intermedio k y el que va directamente del nodo i al nodo j.

Para determinar cuál es el mejor camino considera mínimo (D[v] , D[u] + M(u,v)).

Inicialmente se carga una matriz de NxN con los pesos correspondientes a las aristas, asignándose 0 a M[i][i] para todo i .

El método lleva a cabo N iteraciones del proceso en cada una de las cuales se evalúa la eventual mejora de los tiempos por la consideración del vértice i (que variara entre 1 y N a lo largo del proceso) como intermedio.

Algoritmo de Floyd

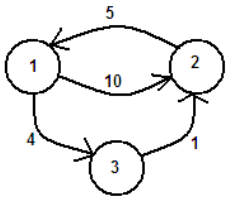
//expresado de manera informal; N es el número de vértices

{
Inicializar A con ceros en la diagonal principal y los pesos indicados por el grafo para cada uno de los restantes elementos. Si una arista no existe, se coloca peso infinito.

```
Para k desde 1 hasta N
   Para i desde 1 hasta N
      Para j desde 1hasta N
        A[i][j] = mínimo (A[i][j] , A[i][j] + A[k][j])
}
```

El coste temporal de este algoritmo es, obviamente $O(N^3)$.

Ejemplo: Consideremos el siguiente grafo para aplicar el algoritmo de Floyd



Matriz A

	1	2	3
1	0	10	4
2	5	0	∞
3	∞	1	0

Pasando por el vértice 1, A queda así

	1	2	3
1	0	10	4
2	5	0	9
3	∞	1	0

Pasando por el vértice 2, A queda así

	1	2	3
1	0	10	4
2	5	0	9
3	6	1	0

Pasando por el vértice 3, A queda así

	1	2	3
1	0	5	4
2	5	0	9
3	6	1	0

Cerradura transitiva:

Conjunto de pares de vértices de un grafo orientado cuyas componentes son vértices ligados por algún camino (no importa la longitud del mismo). Algoritmo de Warshall (1962). (http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=CC6AAB27A661D231154DC7C8D2C3D9FD?doi=10.1.1.89.7172&rep=rep1&type=pdf).

Algoritmo de Warshall:

```
{
  Inicializar la matriz de adyacencia del grafo colocando 1 en la diagonal principal
  Para k desde 1 hasta n
   {
     Para i desde 1 hasta n
      {
        Para j desde 1 hasta n
          { A[i][j] = Maximo ( A[i][j] , A[i][k] * A[k][j]);}
      }
    }
 }
```

Ejercicios propuestos:

1. Dado un grafo dirigido y acíclico con pesos no negativos en las aristas, se quiere determinar la distancia máxima desde un vértice origen a cada uno de los otros. ¿Se puede diseñar un algoritmo similar al de Dijkstra eligiendo al candidato de coste mayor en cada etapa?

2. Modificar el algoritmo de Floyd para calcular el número de caminos con distancia mínima que hay entre cada par de nodos. ¿Cómo queda el coste del algoritmo?

3. Mostrar mediante un contraejemplo que el algoritmo de Dijkstra no sirve si el grafo tiene alguna arista negativa.

Algunos problemas sobre Grafos no dirigidos.

Como ya se ha definido, un árbol libre es un grafo no dirigido conexo sin ciclos.
Se verifica que en todo árbol libre con N vértices (N>1), el árbol contiene N-1 aristas.
Si se agrega una arista a un árbol libre, aparece un ciclo.

Árbol libre, árbol abarcador, árbol abarcador de coste mínimo

Arbol libre:

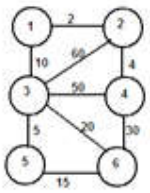
Grafo no dirigido conexo sin ciclos. Se verifica que en todo árbol libre con N vértices (N>1), el árbol contiene N-1 aristas.

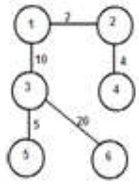
Árbol abarcador: Dado G = (V, A) grafo no dirigido conexo, un árbol abarcador para G, es un árbol libre que conecta todos los vértices de V.

Arbol Abarcador de Coste Mínimo:

Dado un grafo G=(V,A) no dirigido con aristas ponderadas(es decir, que cada arista (v, w) de A tiene un costo asociado, C(v, w)), se llama árbol abarcador de coste mínimo al árbol abarcador para G que verifica que la sumatoria de los pesos de las aristas es mínima. (El árbol abarcador de costo mínimo, puede no ser único).

Ejemplo: para el grafo de la izquierda, un árbol abarcador de coste mínimo es el de la derecha.





Coste: 2+10+4+5+20=41

Algoritmos que permiten obtener el arbol de expansion de coste minimo para un grafo dado

Existen varios algoritmos que permiten obtener el árbol de expansión de coste mínimo.

Se detallaran dos de ellos: el algoritmo de Prim y el de Kruskal.

En ambos casos se aplica la estrategia 'greedy' o voraz para alcanzar el óptimo.

Algoritmo de Prim

Desarrollado por Prim en 1957 sobre un algoritmo previo de Jarnik de 1930.
(https://archive.org/details/bstj36-6-1389)

Usa una estrategia greedy.
Las aristas deben ser no negativas.
Conjunto de candidatos: conjunto de vértices del grafo aún no incorporados al árbol (cualquiera d e ellos se elige como inicial)
Función de selección: elección del vértice que aún no está en el árbol para el cual existe alguna arista de peso mínimo que lo enlaza con algún vértice del árbol.
Función a optimizar: sumatoria de pesos de aristas del árbol
Criterio de finalización: todos los vértices están en el árbol.

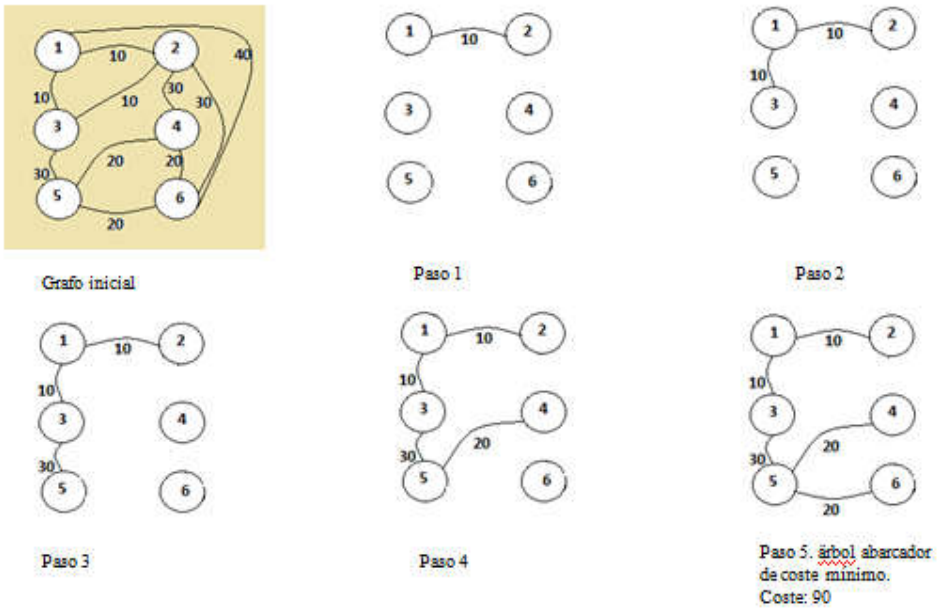
Esquema de Prim:

Llamando G al grafo y A al conjunto de aristas del árbol abarcador de coste mínimo, es

```
{
  A se inicializa en vacío
  U = {1}                          //se almacena vértice inicial
  Mientras U distinto de  V hacer:
   {
```

Determinar la arista de costo mínimo, (u,v) tal que u ϵ U y v ϵ (V-U)
Agregar a A la arista (u, v)
Agregar a U el vértice v
}
}

Algoritmo de Kruskal



Desarrollado por Kruskal en 1956.

(https://pdfs.semanticscholar.org/4f08/a18f205e41c786fc80160ce8e133d50832dc.pdf)

Usa una estrategia greedy. Las aristas deben ser no negativas.
Conjunto de candidatos: conjunto de aristas del grafo.
Función de selección: elección de la arista de coste mínimo.
Restricción: la arista no debe formar ciclo.
Función a optimizar: sumatoria de pesos de aristas del árbol
Criterio de finalización: el árbol tiene n-1 aristas.

Esquema del algoritmo:

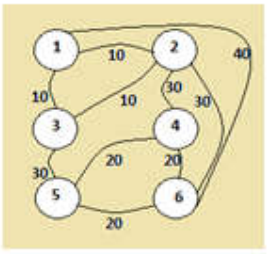
//G.(V, A) grafo no dirigido con aristas no negativas

//Consideraremos los vértices etiquetados como 1..n. Se comenzará incorporando al vértice 1 (arbitrariamente)

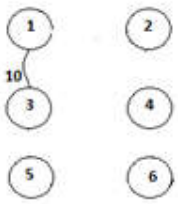
//E: conjunto de aristas del árbol abarcador; U:conjunto de vértices del árbol

```
{
 Ordenar aristas de forma creciente
  Inicializar E en vacío
  Inicializar  U en vacío
  Mientras #E < n-1
    {
      Elegir arista (u,v) de peso mínimo que no forme ciclo
      E = E U {(u,v)}
      Incorporar al conjunto U el vértice u , o el V, según corresponda
    }
}
```

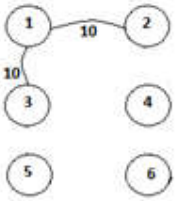
Ejemplo: para el grafo ya considerado,



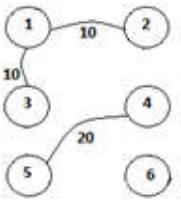
Grafo inicial



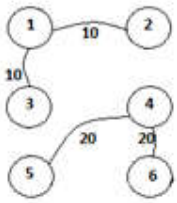
Paso 1



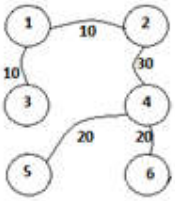
Paso 2



Paso3
La arista de coste 10 entre 2 y 3 forma ciclo, se descarta



Paso 4
La arista de coste 20 entre 5 y 6 se descarta



Paso 5. árbol sbarcador de coste mínimo.
Coste: 90

¿Cómo chequear si cada nueva arista considerada (u,v) forma ciclo?

u y **v** en la misma componente => incorporar (u, v) genera ciclo.
Se comienza con una componente para cada vértice. Las componentes se unen al añadir una arista.
Función que retorna componente de un vértice v: Find(v)
Función que une la componente de u y la componente de v: Union(u , v).
(Se demuestra que ejecutar k uniones con n elementos tiene un coste temporal k log(n), ya que en k uniones hay 2*k elementos involucrados)

Kruskal //versión con Union y Find para un grafo de n vértices
Inicializar A conjunto de aristas del árbol abarcador en vacío

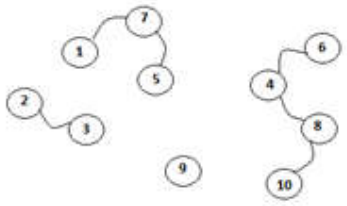
Ordenar S conjunto de aristas del grafo de forma creciente

```
Para cada vértice v
   {MakeSet (v);}
Mientras (#A < n-1)
   {
     Elegir arista (u, v) de peso mínimo y retirarla de S
     si (Find (u) != Find (v))
         {
          Union (Set(u) , Set(v));
           A = A U {(u,v)};
         }
   }
```

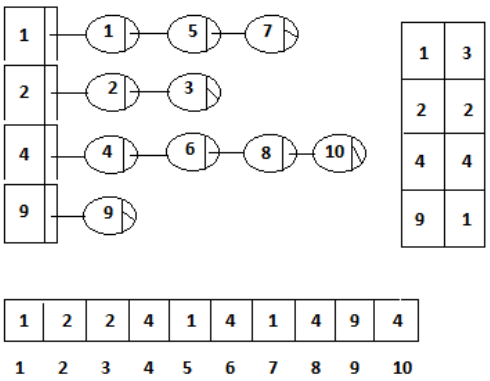
Posibles implementaciones para Union y Find:

Las diferentes formas de implementar las funciones Union y Find influyen en el costo del algoritmo de Kruskal.

Ejemplo, para el siguiente grafo,



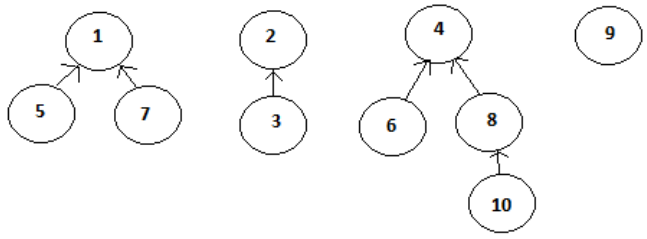
Puede plantearse una implementación de este tipo,



En esta implementación, para las componentes hay un array de listas. El menor valor es representante de la lista. Para unir se elige la lista menor que se incorpora a la otra. Para determinar los tamaños, hay una tabla con los mismos (a la derecha)

El array inferior permite implementar Find con O(1).

Otra posibilidad es



En esta implementación, para las componentes hay estructuras arborescentes "invertidas". En cada una, cada nodo, excepto uno, apunta a su padre. El que está en el lugarde la raíz es el representante de la componente.

Ejercicios propuestos:

1. Analizar el coste del algoritmo de Kruskal con las estructuras sugeridas para implementar Union y Find.

2. ¿Cuál de los dos algoritmos (Prim y Kruskal) conviene más si la densidad de aristas es alta?

3. Analizar la validez, y en ese caso el coste del siguiente algoritmo para determinar las componentes conexas de un grafo no dirigido:

```
    Para cada vértice v del grafo     {MakeSet (v);}
  Para cada arista de (x,y) perteneciente a A hacer
   {
     Si (Find(x) == Find (y))
```

```
        {   Union (x, y);   }
    }
```

¿Cuántas veces se ejecuta Union?
¿Y Find?

4. Se tienen los registros de todos los tramos posibles de viajes que realiza una agencia de turismo. Cada tramo especifica la ciudades de origen y de destino, y para todo tramo (x, y) existe el tramo (y,x). Se necesita resolver lo siguiente: dados los pares (origen x, destino y), ¿cuál es el mínimo número de transbordos que debe realizarse? Diseñar un algoritmo apropiado y evaluar su coste.