UT7 - Grafos Dirigidos

Un grafo dirigido G consiste en un conjunto de vértices V y un conjunto de arcos A. Los vértices se denominan también nodos o puntos, y los arcos arcos dirigidos o líneas dirigidas. Un arco es un par ordenado de vértices (v, w), v es la cola y w la cabeza del arco. Esto se expresa somo v → x y se representa:

se dice que w es adyacente a v.

La longitud de un camino es el número de arcos en el mismo. Un vértice sencillo (v) tiene un camino de longitud cero (de v a v).

Un camino es **simple** si todos sus vértices, excepto tal vez el primero y el último, son distintos.

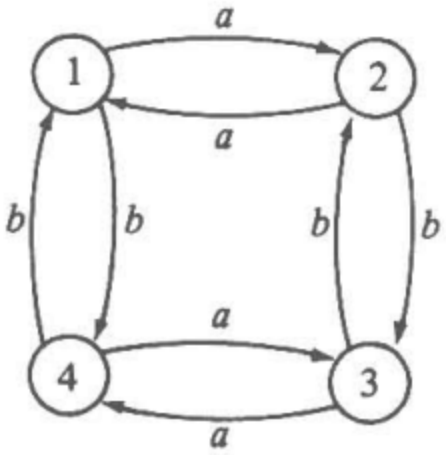
Un **ciclo** simple es un camino simple de longitud por lo menos 1, que empieza y termina en el mismo vértice.

Un grafo dirigido etiquetado cumple la condición de que cada arco, vértice o ambos pueden tener una etiqueta asociada (puede ser nombre, costo, cualquier dato).

Representaciones de grafos dirigidos

Se pueden emplear varias estructuras de datos, dependiendo de las operaciones que se aplicarán a los vértices y a los arcos. Una representación común es la **matriz de adyacencia**.

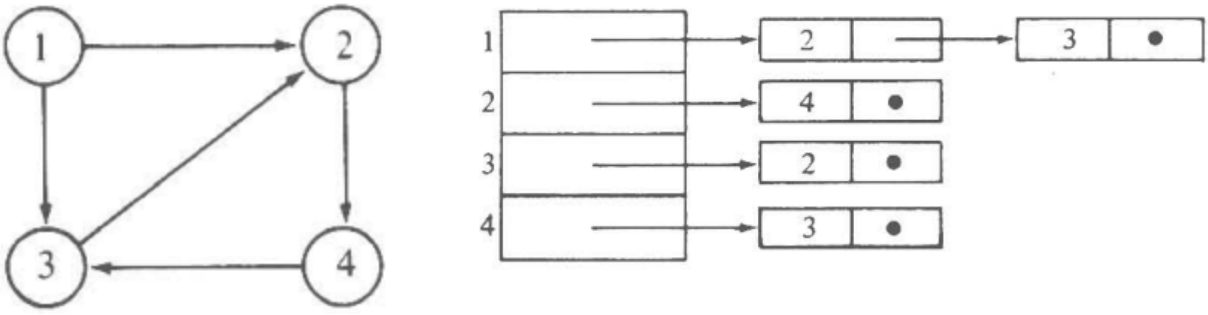
La matriz de adyacencia de un grafo dirigido toma dos vértices del mismo y tiene valor verdadero o falso si existe o no un arco entre ellos. Y la matriz de adyacencia etiquetada es básicamente lo mismo pero en vez de un valor booleano se representa el arco con su etiqueta.



	1	2	3	4
1		a		b
2	a		b	
3		b		a
4	b		a	

La principal desventaja de estas matrices es que requieren un espacio de n**2. Sólo leer o examinar la matriz puede llevar un tiempo O(n**2).

Para evitar esta desventaja se puede usar una **lista de adyacencia,** la misma se obtiene para un determinado vértice v, y consiste en una lista, en cualquier orden, de todos los vértices adyacentes a v. Esta lista requiere un espacio proporcional a la suma del numero de vértices más el número de arcos. Una gran desventaja es que puede llevar un tiempo O(N) determinar si existe un arco de un cierto vértice a otro cierto vértice, ya que pueden haber N vértices en la lista de adyacencia para ese vértice.



Problema de los caminos más cortos con un sólo origen

Existe un grafo dirigido G = (V, A) en el cuál cada arco tiene una etiqueta y cada vértice se especifica como origen. El problema es determinar el corto del camino más corto del

origen a todos los demás vértices de V, donde la **longitud de un camino** es la suma de los costos de los arcos del camino.

Por ejemplo: G es un mapa de vuelos, donde cada vértice es una ciudad y cada arco una ruta aérea de una ciudad a la otra. La etiqueta del arco v → w es el tiempo que se demora en volar de v a w. La solución del problema anterior determina el tiempo de viaje mínimo para ir de cierta ciudad a las demás en el mapa.

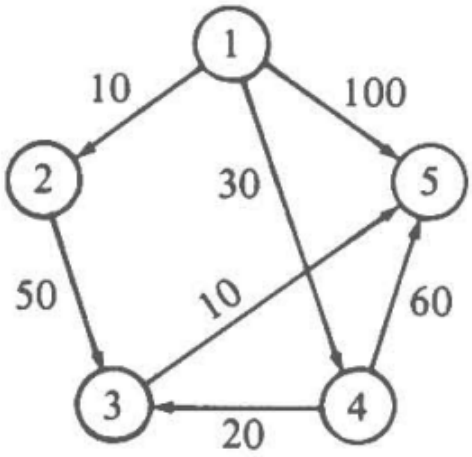
Para resolver el problema se utilizará una **técnica ávida** llamada **algoritmo de Dijkstra**, que opera a partir de un conjunto S de vértices cuya distancia más corta desde el origen se conoce. En principio, S contiene sólo el vértice de origen, en cada paso se agrega algún vértice restante, cuya distancia desde el origen es la más corta posible. En cada paso del algoritmo se utiliza un arreglo D para registrar la longitud del camino especial (que pasa por S) más corto a cada vértice. Una vez que S incluye todos los vértices, D contendrá la distancia más corta del origen a cada vértice.

```
        procedure Dijkstra;
            { Dijkstra calcula el costo de los caminos más cortos entre el vértice 1
                y todos los demás de un grafo dirigido }
        begin
(1)         S := { 1 };
(2)         for i := 2 to n do

(3)             D[i] := C[1, i]; { asigna valor inicial a D }
(4)         for i := 1 to n–1 do begin
(5)             elige un vértice w en V–S tal que D[w] sea un mínimo;
(6)             agrega w a S;
(7)             for cada vértice v en V–S do
(8)                 D[v] := mín(D[v],D[w] + C[w, v])
            end
        end; { Dijkstra }
```

Ejemplo:

P[2] = 1, P[3] = 4, P[4] = 1 y P[5] = 3 (Te fijas en el grafo de donde vienen las flechas para cada vértice, si le llega más de una, entonces te quedas con la de menor valor). Para encontrar el vértice más corto de 1 a 5 se siguen los predecesores (los determinados al



principio del ejemplo) en orden inverso, arrancando por 5. El predecesor de 5 es 3, el de 3 es 4 y el de 4 es 1, como llegamos hasta el 1 (el origen en este ejemplo) paramos. Entonces el camino más corto entre los vértices 1 y 5 es 1, 4, 3, 5. Esto se comprueba al mirar el grafo y contar los valores de los arcos, este camino es el de menor valor.

Cálculos de Dijkstra para el grafo anterior:

Iteración	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
inicial	{1}	—	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Tiempo de ejecución de Dijkstra: Para un grafo dirigido con n vértices y a aristas, si se emplea una matriz de adyacencia el tiempo de ejecución es de O(n**2). Si a es mucho menor que n**2, es mejor utilizar una lista de adyacencia, dejando el tiempo de ejecución del algoritmo en O(a log n).

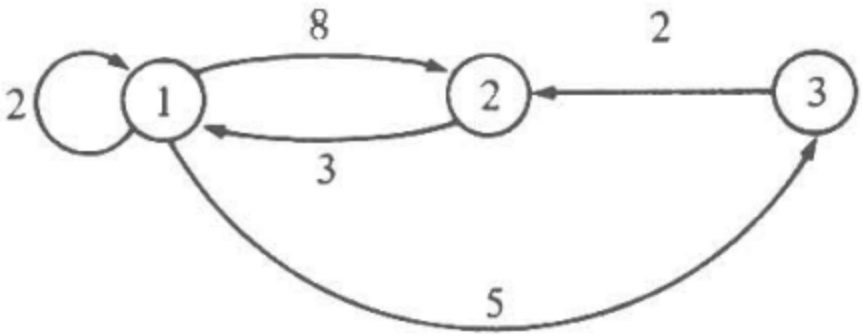
Problema de los caminos más cortos entre los pares (CMCP)

Suponga que se tiene un grafo dirigido etiquetado que da el tiempo de vuelo para ciertas rutas entre ciudades y se desea construir una tabla con el menor tiempo de vuelo entre dos ciudades cualesquiera.

Se toma un grafo G = (V, A), el problema es encontrar el camino de longitud más corta entre v y w para cada par ordenado de vértices (v, w).

Este problema podría resolverse con Dijkstra, tomando por turno cada vértice como vértice de origen, pero una forma mas directa es mediante el **algoritmo de Floyd**. El mismo utiliza una matriz A de nxn en la que se calculan las longitudes de los caminos más cortos. Inicialmente de hace A[i, j] = C[i, j] para toda i ≠ j. Si no existe un arco que vaya de i a j se supone que C[i, j] = infinito. Cada elemento de la diagonal se hace 0. Después se hacen n iteraciones en la matriz, en cada una la longitud del camino va disminuyendo (si es posible, sino queda igual), así al final de la k-esima iteración A[i, j] tendrá por valor la menor longitud de cualquier camino entre dos vértices.

Floyd tiene un tiempo de ejecución de O(n**3).



	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_0[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_1[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_2[i, j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

$A_3[i, j]$

```
procedure Floyd ( var A: array[1..n, 1..n] of real;
        C: array[1..n, 1..n] of real );
    { Floyd calcula la matriz A de caminos más cortos dada la matriz de costos
      de arcos C }
    var
        i, j, k: integer;
    begin
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := C[i, j];
        for i := 1 to n do
            A[i, i] := 0;
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    if (A[i, k] + A[k, j]) < A[i, j] then
                        A[i, j] := A[i, k] + A[k, j]
    end; { Floyd }
```

Recuperación de los caminos

```
procedure más_corto ( var A: array[1..n, 1..n] of real;
    C: array[1..n, 1..n] of real; P: array[1..n, 1..n] of integer );
    { más_corto toma una matriz de costos de arcos C de n×n y produce una matriz A
        de n × n de longitudes de caminos más cortos y una matriz P de n × n
        que da un punto en la «mitad» de cada camino más corto }
    var
        i, j, k: integer;
    begin
        for i := 1 to n do
            for j := 1 to n do begin
                A[i, j] := C[i, j];
                P[i, j] := 0
            end;
        for i := 1 to n do
            A[i, i] := 0;
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    if A[i, k] + A[k, j] < A[i, j] then begin
                        A[i, j] := A[i, k] + A[k, j];
                        P[i, j] := k
                    end
    end; { más_corto }
```

Para imprimir los vértices intermedios del camino más corto entre dos vértices, se invoca el procedimiento camino()

```
procedure camino ( i, j: integer );
    var
        k: integer;
    begin
        k := P[i, j];
        if k = 0 then
            return;
        camino(i, k);
        writeln(k);
        camino(k, j)
    end; { camino }
```

Cerradura transitiva

En algunos casos podría ser interesante saber si existe un camino de longitud igual o mayor a uno que vaya de v a w. Para esto se puede especializar el algoritmo de Floyd, obteniendo el **algoritmo de Warshall**.

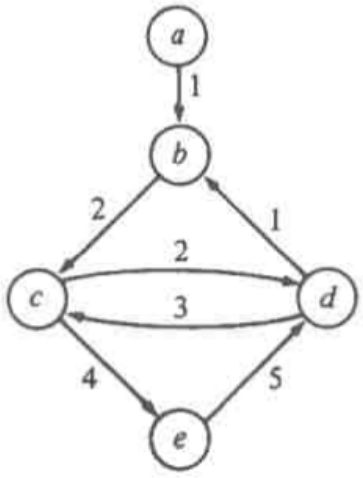
Supongamos que existe una matriz C=[i,j], donde C=1 si hay un arco entre i y j, y C=0 sino. Se quiere obtener la matriz A tal que A[i,j] = 1 si hay un camino de longitud igual o mayor que uno de i a j, y 0 sino. A sería la cerradura transitiva de la matriz de adyacencia.

Esta es la cerradura transitiva para el grafo planteado más arriba:

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

```
procedure Warshall ( var A: array[1..n, 1..n] of boolean;
        C: array[1..n, 1..n] of boolean );
    { Warshall convierte a A en la cerradura transitiva de C }
    var
        i, j, k: integer;
    begin
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := C[i, j];
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    if A[i, j] = false then
                        A[i, j] := A[i, k] and A[k, j]
    end; { Warshall }
```

El **centro** de un grafo es el vértice con menor excentricidad. La excentricidad es, del mínimo camino que cada vértice tiene hacia el vértice al que se le está calculando la excentricidad, el mayor (la distancia al vértice más alejado).



vértice	excentricidad
a	∞
b	6
c	8
d	5
e	7

El centro es el vértice d.

Para encontrar el centro de un grafo dirigido G, donde C es la matriz de costos:

1. Se aplica Floyd a C para obtener la matriz A de los caminos más cortos entre todos los pares.
2. Se encuentra el costo mínimo de cada columna i, esto da la excentricidad del vértice i.
3. Se encuentra el vértice con excentricidad mínima, es decir el centro de G.

El tiempo de ejecución de este proceso sería O(n**3) ya que el paso 1 tiene ese orden y es el mayor.

Recorridos en grafos dirigidos

Búsqueda en profundidad (bpf)

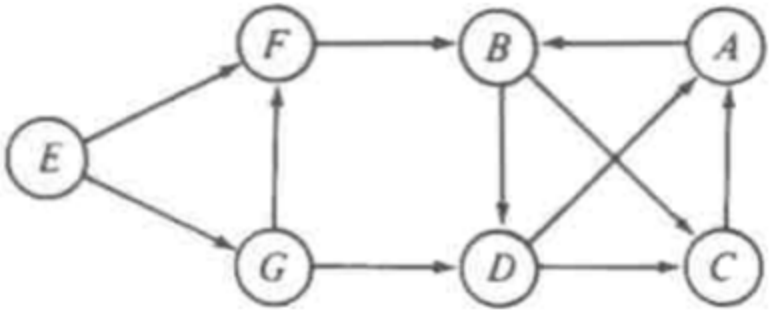
Es una generalización del recorrido en preOrden de un árbol.

Se tiene un grafo dirigido G en el cual todos los vértices están marcados en principio como *no visitados*, esta búsqueda selecciona un vértice de partida y lo marca como visitado, luego recorre cada vértice adyacente a v no visitado recursivamente. Una vez que se visitan todos los vértices adyacentes a v, la búsqueda de v está completa, luego se chequea si algún vértice sigue *no visitado* y se lo toma como vértice de partida.

Todas las llamadas bpf en la búsqueda en profundidad de un grafo con a arcos y n ≤ a vértices tiene un tiempo O(a), esto es porque en ningún vértice se llama a bpf más de una vez (ya que a visitarlo se marca como *visitado*).

```
                procedure bpf ( v: vértice );
                    var
                        w: vértice;
                    begin
(1)                     marca[v] := visitado;
(2)                     for cada vértice w en L[v] do
(3)                         if marca[w] = no_visitado then
(4)                             bpf(w)
                    end; { bpf }
```

Ejemplo:



El algoritmo marca A como visitado y selecciona el vértice B de la lista de adyacencia de A, como B no ha sido visitado, la búsqueda llama bpf(B). El algoritmo marca B como visitado y ahora selecciona el primer vértice en la lista de adyacencia de B (puede ser C o D, depende de cómo esté en la lista), supongamos que es C, se invoca bpf(C), A se encuentra en la lista de adyacencia de C, pero ya fue visitado. Luego se invoca bpf(D) y en la lista de adyacencia de D está C, pero también fue visitado ya. En este punto la llamada recursiva bpf(A) ya terminó, pero hay vértices del grafo que no fueron visitados (E,F,G), entonces para completar la búsqueda se llama bpf(E).

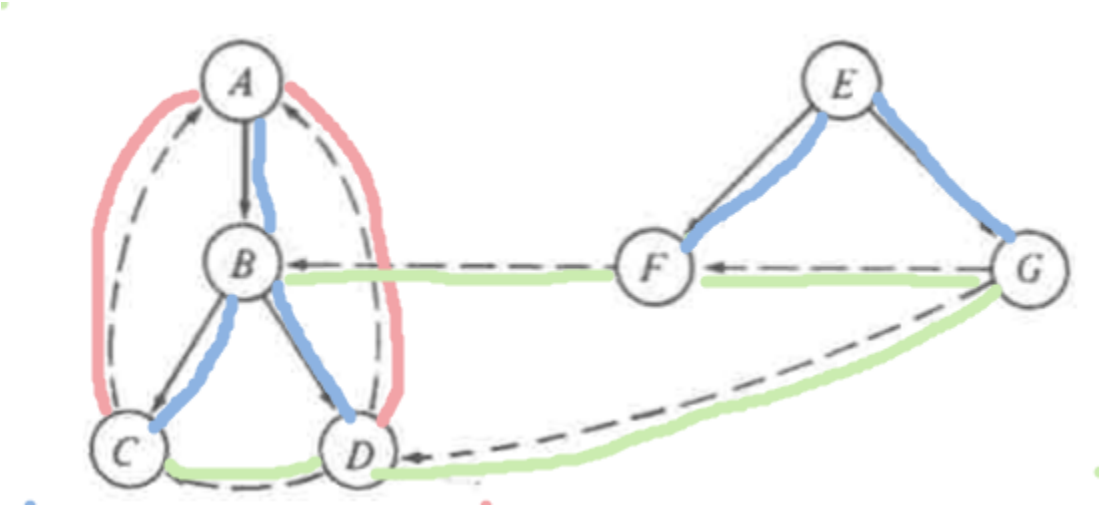
Bosque abarcador en profundidad

Durante un recorrido en profundidad de un grafo dirigido, cuando se recorren ciertos arcos, llevan a vértices sin visitar. Los arcos que llevan a nuevos vértices se llaman **arcos de árbol** y forman un **bosque abarcador en profundidad** para el grafo dirigido.

Además existen los **arcos de retroceso** y son aquellos que van de un vértice a uno de sus antecesores en el bosque (uno que ya fue visitado). Un arco que va de un vértice a sí mismo es un arco de retroceso. (C → A)

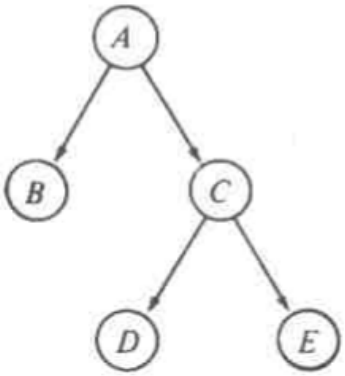
Un arco **no** abarcador que va de un vértice a un descendiente propio se llama **arco de avance**.

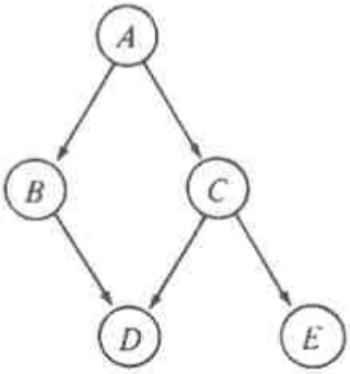
Un arco que va de un vértice a otro que **no** es antecesor ni descendiente se llama **arco cruzado** (D → C, G → D). Todos los arcos cruzados de la imagen van de derecha a izquierda, si se agregan hijos al árbol en el orden en que fueron visitados sería de izquierda a derecha, y si se agregan árboles nuevos al bosque tmb sería de izquierda a derecha.

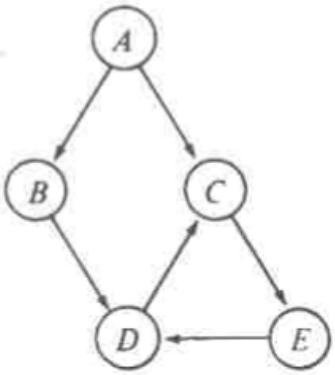


Grafos dirigidos acíclicos (gda)

Es un grafo dirigido sin ciclos (pretty obvious huh). Los gda son más generales que los árboles, pero menos que los grafos dirigidos. A continuación se deja una imagen con un árbol, un gda y un grafo dirigido con un ciclo. Los tres son grafos dirigidos:







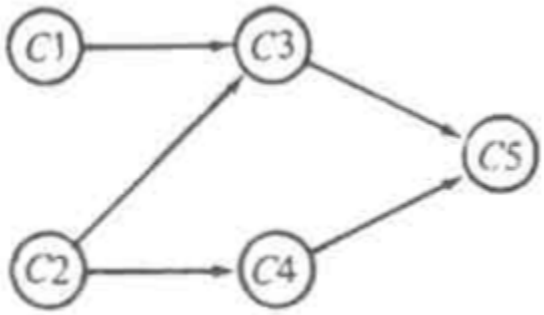
Prueba de aciclicidad

Para el grafo G = (V, A) se quiere determinar si es acíclico. Para esto se puede usar la búsqueda en profundidad, si se encuentra un arco de retroceso el grafo tiene un ciclo.

Clasificación topológica

Un proyecto grande suele dividirse en varias tareas más pequeñas, algunas de las cuales deben realizarse en cierto orden específico, para que pueda concluir el proyecto. Los gda pueden emplearse para modelar de manera natural estas situaciones. Por ejemplo se pone un arco de C a D si C fuera un prerrequisito de D.

Ejemplo:



Este grafo G representa una estructura de prerrequisitos de cinco cursos, donde para cursar C3 es necesario aprobar antes C1 y C2.

La **clasificación topológica** es un proceso de asignación de un orden lineal a los vértices del gda tal que si existe un arco del vértice i a j, i aparece antes que j en el ordenamiento lineal. C1, C2, C3, C4, C5 es una clasificación topológica de G.

```
procedure clasificación_topológica(v: vértice);
    {imprime los vértices accesibles desde v en orden topológico invertido}
    var
        w: vértice;
    begin
        marca[v] := visitado;
        for cada vértice w en L[v] do
            if marca[w] = no_visitado then
                clasificación_topológica(w);
        writeln(v)
     end; {clasificación_topológica}
```

Cuando el método termina de buscar todos los vértices adyacentes de un vértice x, imprime x. El fin de llamar a este método es imprimir en orden topológico inverso todos los vértices de un gda accesibles desde v por un camino en el gda. Esto funciona porque **no** existen arcos de retroceso