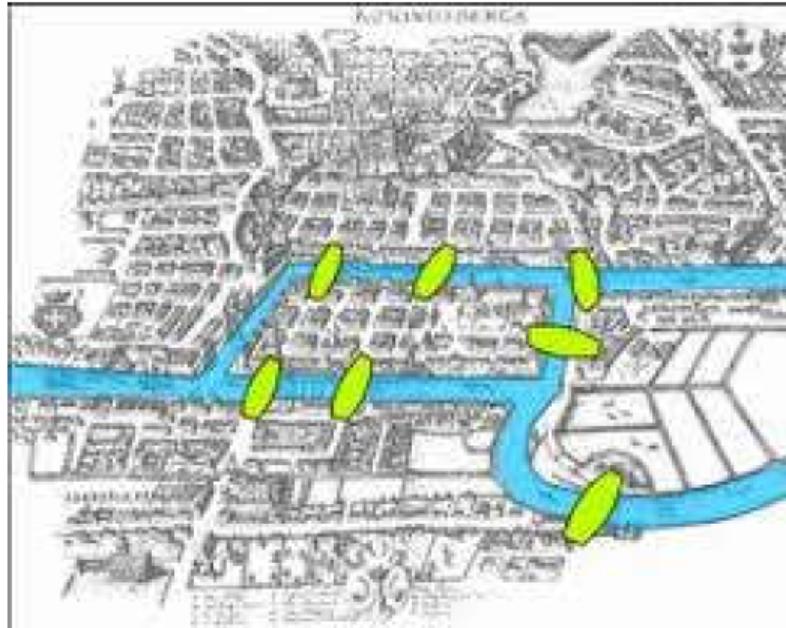


Grafos

- ▶ Materia Algoritmos y Estructuras de Datos - CB100
- ▶ Cátedra Schmidt
- ▶ Grafos

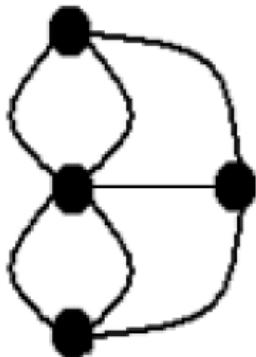
Grafos

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 denominó ‘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.

Para relatar de forma somera la resolución de Euler, podemos decir que se puede considerar que en un grafico de este tipo hay 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 por eso son intermedios). Se demuestra que, para que sea posible realizar el dibujo en las condiciones indicadas, si el inicio y la finalización se hace en nodos distintos, estos nodos, y solamente estos deben tener un número impar de aristas incidentes; el resto de los nodos debe tener “grado” par. Si lo que se quiere es comenzar y terminar en el mismo vértice, entonces todos los nodos deben tener un número par de aristas incidentes en ellos.

Analizando la situación de los puentes, se concluye que el problema de los puentes de Königsberg no tenía solución.

De este modo comenzó la teoría de Grafos.

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

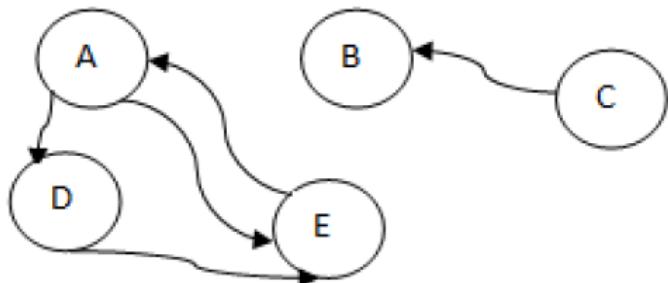
Los grafos son orientados (o dirigidos o digrafos) si las aristas (o arcos) que conectan sus vértices (también llamados nodos) están orientadas.

Los grafos son no orientados (o no dirigidos) si las aristas que conectan sus vértices no están orientadas.

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 pertenecientes a $V \times V$.

En símbolos el grafo dirigido G es

$G = (V, A)$ donde A es un subconjunto de $V \times V$ Ejemplo:



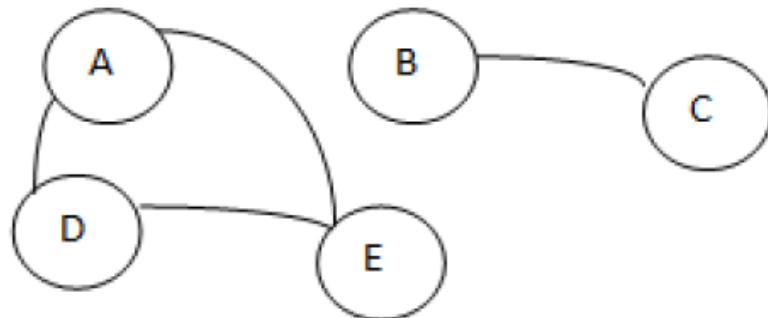
$$V = \{A, B, C, D, E\}$$

$$A = \{(A, D), (A, E), (E, A), (D, E), (C, B)\}$$

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 $V \times V$.
En símbolos el grafo dirigido G es

$G = (V, A)$ donde A es un conjunto de pares no ordenados de $V \times V$

Ejemplo:



$$V = \{A, B, C, D, E\}$$

$$A = \{(A, D), (A, E), (D, E), (C, B)\} \text{ (pares no ordenados)}$$

En ambos casos, si (v, w) pertenece a A se dice que w es adyacente a v.

Camino: secuencia de vértices v_1, v_2, \dots, v_n tales que

v_2 es adyacente a v_1

v_3 es adyacente a v_2

...

v_n es adyacente a v_{n-1}

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 que conteniendo vértices distintos, excepto el primero que coincide con el último.

Subgrafo: dado el grafo $G = (V, A)$, un *subgrafo* de G es un grafo $G' = (V', A')$ que cumple V' es un subconjunto de V y A' es un subconjunto de A .

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

Grafo subyacente de un grafo dirigido: 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 si 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.

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

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).

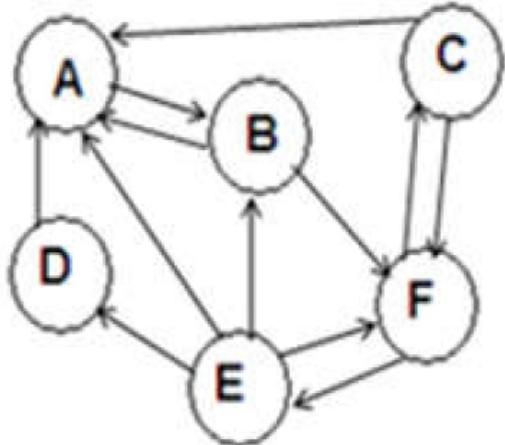
Estructuras de datos utilizadas para implementar grafos

Como sucede en la mayoría de los TDA Contenedores, las propuestas básicas de implementaciónn almacenan los datos de forma contigua (por ejemplo, usando arrays, matrices o tablas) o de forma dispersa (utilizando listas con punteros).

Entonces, la primera alternativa es 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. Adicionalmente, en alguna estructura se puede almacenar, si es necesario, información adicional sobre los vertices (identificadores o etiquetas, y quizás valores asociados)

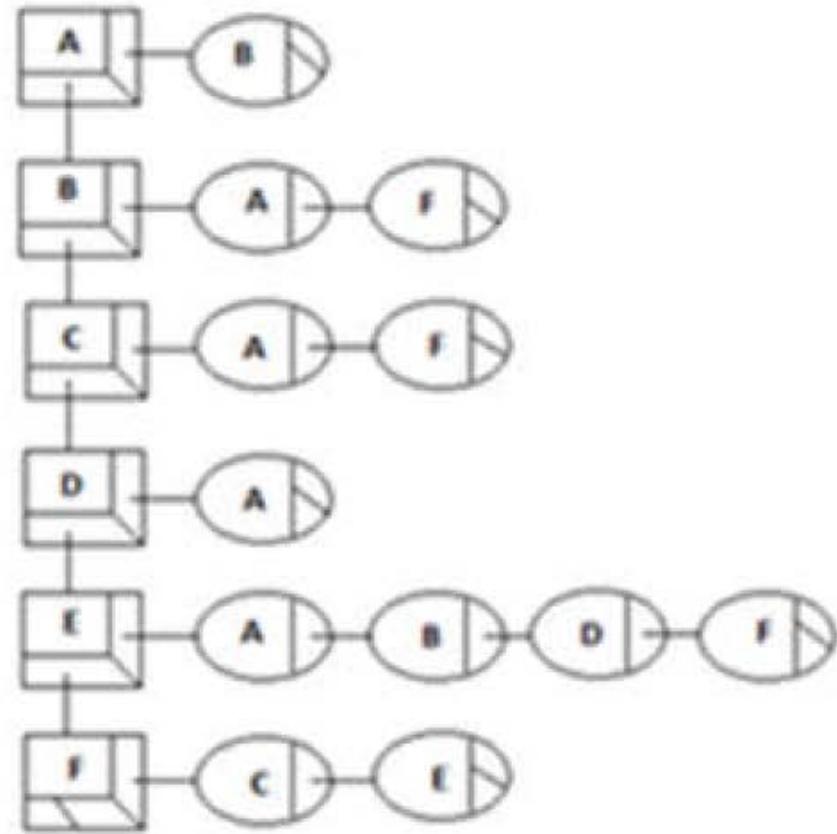
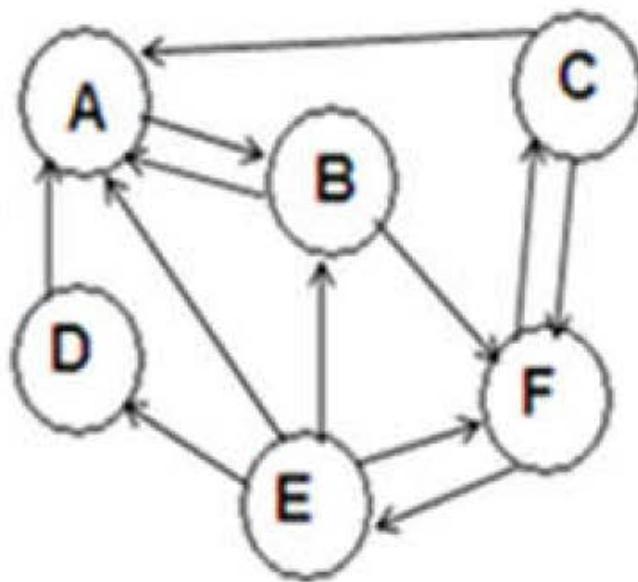
Ejemplo:



	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

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:



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.

Recorrido en profundidad (*depth first search* o ‘busqueda en profundidad’):

Consiste en hacer lo siguiente:

Inicialmente se marcan todos los nodos como “no visitados”.

Luego, mientras haya vértices no visitados, se toma el primero de ellos, y se lo visita y marca (como visitado) para luego pasar al primer adyacente de este vértice que se haya no visitado y proceder del mismo modo, pasando luego al primer adyacente del adyacente que no haya sido visitado, y así hasta llegar a un nodo que no tenga adyacentes no visitados.

El recorrido concluye con la visita de todos los nodos, no pasando por ninguno de ellos más de una vez.

Para implementar el algoritmo suele usarse una pila, o bien un algoritmo recursivo.

Puede, además, numerarse los vértices a medida que se visitan, tal como se lleva a cabo en los algoritmos que se describen a continuación.

Recorrido en Profundidad recursivo con numeración de vértices

//Se considera que cada vértice tiene un atributo número que almacena el orden en que se lo visitó.

//Inicialmente todos los atributos numero valen 0. Eso significa que no fueron visitados.

//Índice es una variable que permite realizar el conteo de los nodos visitados para asignar el valor a

// cada atributo número de cada vértice

DFSRecursivoCon Numeracion

{

Para cada vértice v del grafo G

{

Asignar 0 a número de v;

indice= 0;

}

Para cada vértice v

{

Si (numero de v es 0) entonces

{Numerar (v, indice);} //índice pasa por referencia; su valor se actualiza entre llamadas

}

}

Numerar (u:vértice; var nd: entero)

{

 nd = nd+1;

 Asignar nd al atributo número del vértice u;

 Para cada vértice w adyacente a u

 {

 Si (numero de w es 0) entonces

 {Numerar (w, nd);}

 }

}

Análisis del coste temporal:

El coste depende de las estructuras que se usen para almacenar el grafo.

Si tenemos un grafo G con una cantidad v de vértices y a aristas y tenemos una implementación de matriz de adyacencia, observamos que:

El ciclo que inicializa numero de cada vértice marcándolo como no visitado es $O(v)$.

El ciclo que revisa si cada vértice ha sido o no visitado para invocar a Num tiene $O(v)$ iteraciones.

El coste del bloque depende del coste de la function Numerar.

Numerar comienza con una secuencia de sentencias de coste $O(1)$. Luego hay un ciclo que analiza cada adyacente a u para invocar a Numerar si ese adyacente no ha sido visitado.

Para determinar los adyacentes a u si tenemos implementación en Matriz de Adyacencia hay que recorrer la fila correspondiente al vértice u en la matriz. Esto involucra $O(v)$ iteraciones, siendo v el número total de veces que se ejecuta Numerar. 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,

Iniciar numero de cada vértice en 0 es $O(v)$.

Numerar se invoca para cada vértice, tantas veces como adyacentes tenga el vértice (pero hay que considerar que los vertices procesados por ser adyacentes a un vértice no son después procesados de nuevo). Los adyacentes a un vértice se determinan recorriendo la lista de vertices adyacentes. El proceso Numerar realizará el recorrido de todas las listas de nodos ayacentes. El costo se expresa por tanto como $O(a)$.

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:

//Como en el algoritmo anterior, se considera que cada vértice tiene un atributo número

//que almacena el orden en que se lo visitó. Inicialmente todos los atributos numero valen 0,

//lo cual indica que no fueron visitados

//Índice es una variable que permite realizar el conteo de los nodos visitados para asignar el valor a

// cada atributo número de cada vértice

// Este algoritmo utiliza una pila inicialmente vacía

DFSIterativoConNumeracion

{

Iniciar pila p a vacío;

indice = 0;

Para cada vértice v

{

Asignar 0 a numero de v;

}

Apilar v en pila p;

Mientras pila p no vacía

{

Desapilar p en u; //u es una variable tipo vértice

Si (el numero de u es 0)

{

índice= índice +1;

asignar índice a numero de u;

Para cada w adyacente a u

Si (numero de w es 0)

{

Apilar w en pila p;

}

}

}

Análisis del coste temporal:

Si se trabaja con un grafo G con una cantidad v de vértices y a aristas y con una implementación de matriz de adyacencia, sucede lo siguiente:

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.

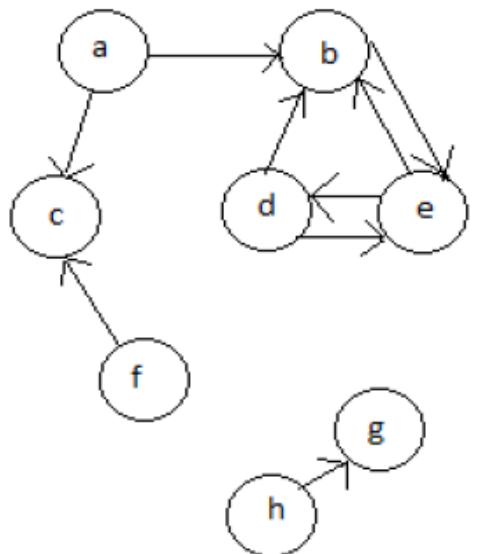
En cambio, con una implementación con listas de adyacencia,

El primer ciclo tiene coste $O(v)$.

Luego se tiene, para cada vértice desapilado y con numeración 0 (es decir, no visitado), el apilamiento de sus adyacentes. 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:

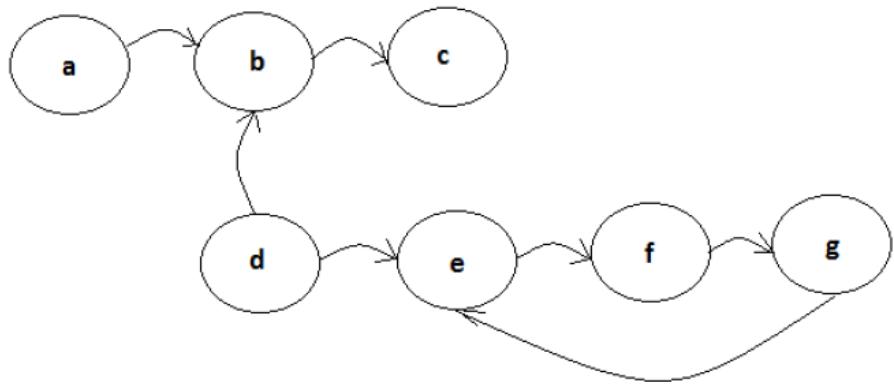


a	b	e	d	c	f	g	h
---	---	---	---	---	---	---	---

Test de aciclicidad: (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 de 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 (es decir, la cuestión no es si se nos encontramos con un vértice que ya ha sido visitado, sino si ese vértice ya visitado lo es en el recorrido parcial que se está realizando).

Ejemplo:



En la figura se observa que, partiendo del vértice a se visita a-b-c. Ese es el primer “recorrido parcial”. Luego se comienza con d un nuevo recorrido parcial, porque es el siguiente vértice no visitado. Se visita d. Los adyacentes a d son b y e. El vértice c ya ha sido visitado, pero esto no significa que exista un ciclo. De d se pasa y visita e, luego se visita f y luego g. El vértice e es adyacente a g y ya ha sido visitado, pero lo ha sido en este recorrido parcial. Eso es lo que indica que hay un ciclo. Por lo cual se deberá llevar un “doble registro” de vértices visitados: para todo el grafo y para cada recorrido parcial. La información de lo que sucede en cada recorrido parcial es la que permite detectar ciclos.

El algoritmo puede plantearse de la siguiente forma:

```
//La variable lógica b almacena el valor a retornar  
//Este algoritmo utiliza para cada vértice un atributo que indica si fue visitado alguna vez (en  
//cualquiera de los recorridos realizados, es una “marca general”)  
//Y usa otro atributo llamado en_recorrido_actual para indicar si fue visitado en el recorrido parcial  
que se esté realizando  
//Luego de terminado cada recorrido parcial, el atributo en_recorrido_actual se reinicializa
```

Test_Aciclidad

{

 Inicializar b en falso; //valor lógico a retornar

 Para cada vértice v

{

 Marcar v como no visitado;

 Asignar falso a atributo en_recorrido_actual de v;

}

Para cada vértice v

```
{  
    Si (b== falso) y (v no visitado)  
    {  
        Analizar_Recorrido_Parcial(v, b); // b pasa por referencia  
    }  
    Retornar b;  
}
```

Analizar_Recorrido_Parcial (u :vértice; var b: bool)

{

 Marcar u como visitado;

 Asignar verdadero a atributo en_recorrido_actual de u;

 Para cada w adyacente a u

 {

 Si (b==falso)

 {

 Si (w visitado) y (en_recorrido_actual de w == verdadero)

 {asignar verdadero a b;}

 Si (w no visitado)

 { Analizar_Recorrido_Parcial (w,b); }

 }

 }

 Asignar falso a atributo en_recorrido_actual de u;

}

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.

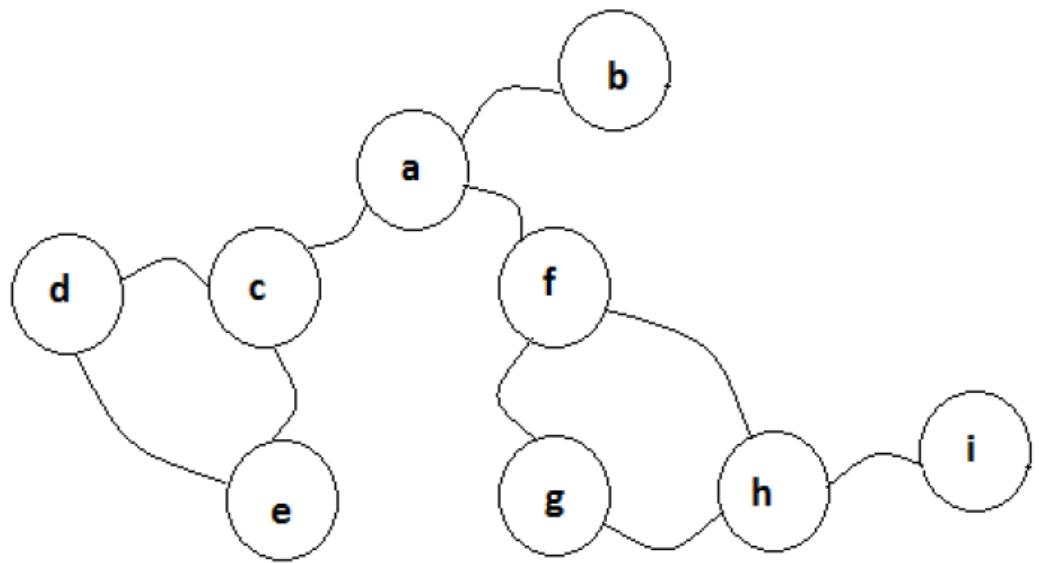
Puntos de articulación de un grafo: (aplicación del recorrido en profundidad).

Otra información que puede obtenerse de un grafo a través del recorrido en profundidad, es el conjunto de puntos de articulación.

Un punto de articulación (o vértice de corte) de un grafo no dirigido y conexo es un vértice que verifica que al ser eliminado del grafo éste deja de ser conexo.

Ejemplo:

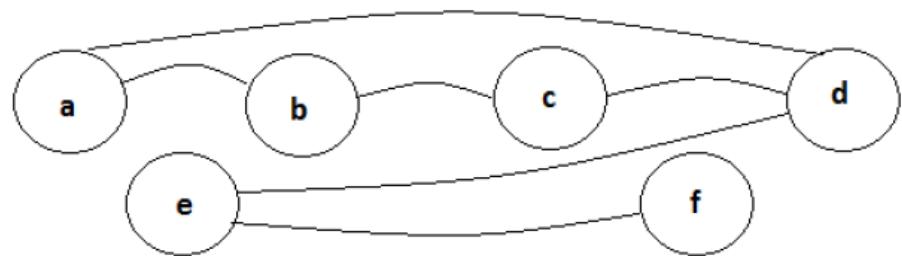
En la siguiente figura se puede observar que si se elimina alguno de estos vértices: a, c, f, h, el grafo deja de ser conexo. Esos vértices son puntos de articulación o vértices de corte.



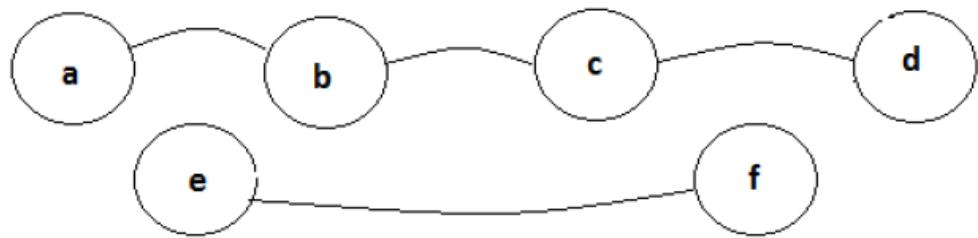
Si un grafo no tiene puntos de articulación significa que para todo par de vértices existe más de un camino que los enlaza. Entonces, si un vértice del grafo “se cayera” (es decir, si hubiera que eliminarlo del grafo) el grafo permanecería aún conexo. A los grafos que no tienen puntos de articulación se los llama biconexos.

Vamos a obtener los puntos de articulación de un grafo no dirigido aplicando un recorrido en profundidad. Para esto consideremos que cuando realizamos un recorrido de ese tipo, cada recorrido parcial (es decir el que comenzamos con un vértice en particular) es un “árbol libre” tal como lo hemos definido al principio. El conjunto de estos árboles obtenidos cada uno por un recorrido parcial se denomina “bosque”.

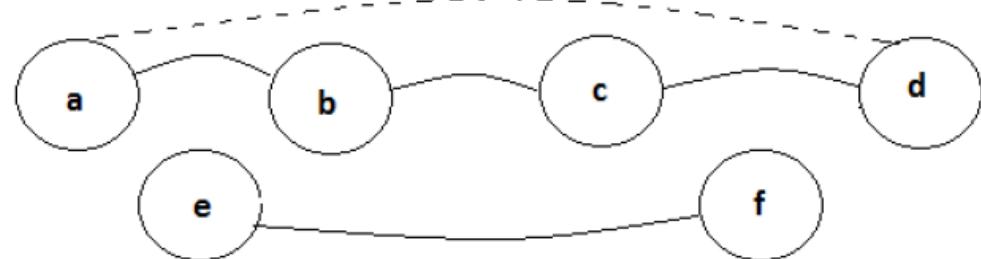
En el transcurso del recorrido, algunas aristas del grafo no aparecerán en el árbol de un recorrido parcial por ser “aristas de retroceso”. Ejemplo: en el grafo de la siguiente figura,



Este es el bosque del recorrido en profundidad. Tiene dos árboles.



Se ha marcado en línea punteada la arista de retroceso.



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, queremos saber para cada uno cuan atrás podemos llegar desde él. Consideraremos la función bajo, que tiene el menor número entre: el número que le asignamos al realizar el recorrido en profundidad, el número de algún vértice alcanzado desde él mediante una arista de retroceso, y el valor bajo alcanzado por alguno de sus hijos.

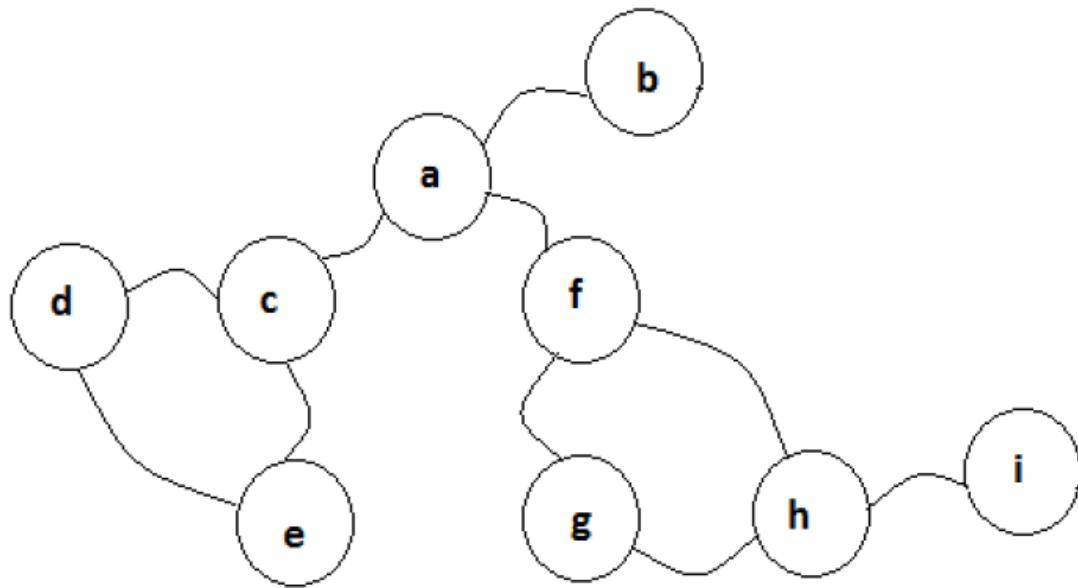
$bajo(u)$ = mínimo número asignado en el recorrido en profundidad considerando estos valores:

- Número asignado al vértice u en el recorrido en profundidad
- Número asignado en el recorrido en profundidad a cada nodo w al que se accede desde u por una arista de retroceso
- $bajo(x)$ para todo x hijo de u en el árbol que se genera en el recorrido.

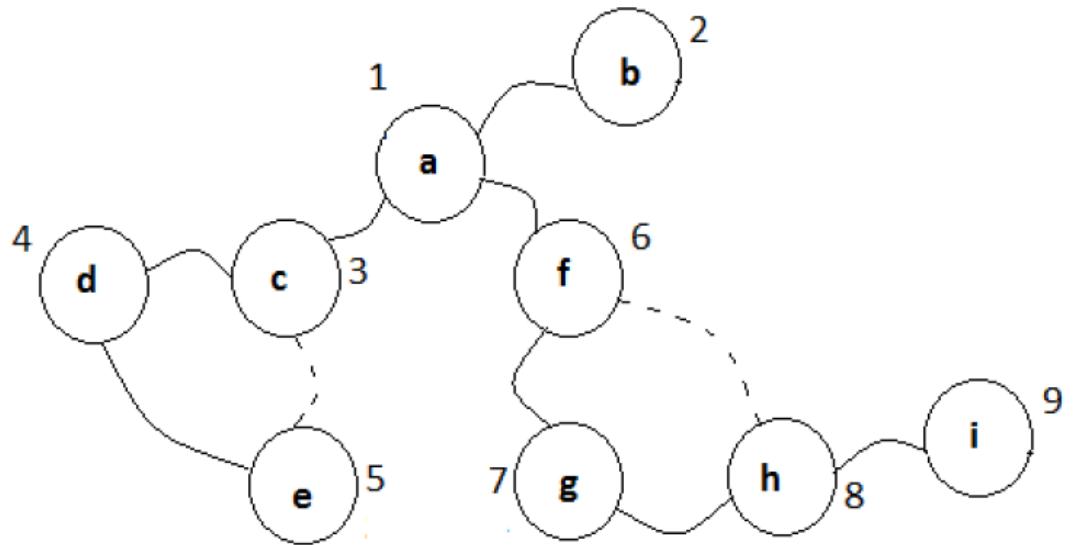
Entonces el vértice u , si no es raíz, es punto de articulación si y sólo si tiene al menos un hijo x tal que $bajo(x) \geq$ numeroasignado al vértice u en el recorrido en profundidad.

Esto es así porque significa que ese vértice x “necesita” de u para acceder a otros de numeración más baja. Si u se eliminara del grafo, dejaría de ser conexo.

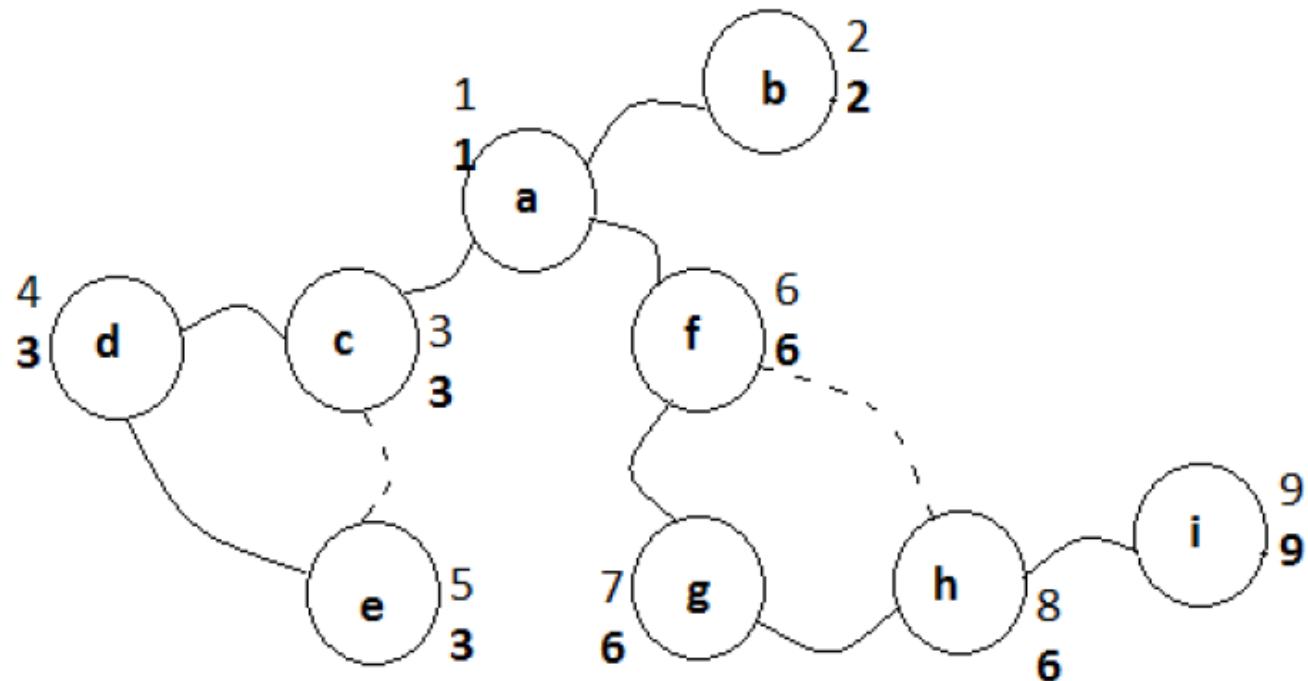
A continuación se muestra el proceso de obtención de los puntos de articulación:
Partimos de



Numeramos cada vértice realizando el recorrido en profundidad y graficamos el árbol obtenido con el recorrido. En líneas punteadas las aristas de retroceso.



En la siguiente figura se ha colocado el valor de bajo(v) junto a cada vértice, debajo del número asignado en el recorrido.



Analizamos lo que sucede con cada vértice:

El vértice a es punto de articulación por ser raíz del árbol de recorrido.

Para los vértices no raíz consideraremos lo que sucede con la siguiente tabla:

Analizamos lo que sucede con cada vértice:

El vértice **a** es punto de articulación por ser raíz del árbol de recorrido.

Para los vértices no raíz consideremos lo que sucede con la siguiente tabla:

Vértice	Hijos
b (numeración: 2)	----
c (numeración: 3)	d (bajo: 3)
d (numeración: 4)	e (bajo: 3)
e (numeración: 5)	---
f (numeración: 6)	g (bajo: 6)
g (numeración: 7)	h (bajo: 6)
h (numeración: 8)	i (bajo: 9)
i (numeración: 9)	---

Observamos que los vértices **c**, **f** y **h** tienen hijos cuyo valor de bajo es mayor o igual que el número del vértice padre. Son puntos de articulación.

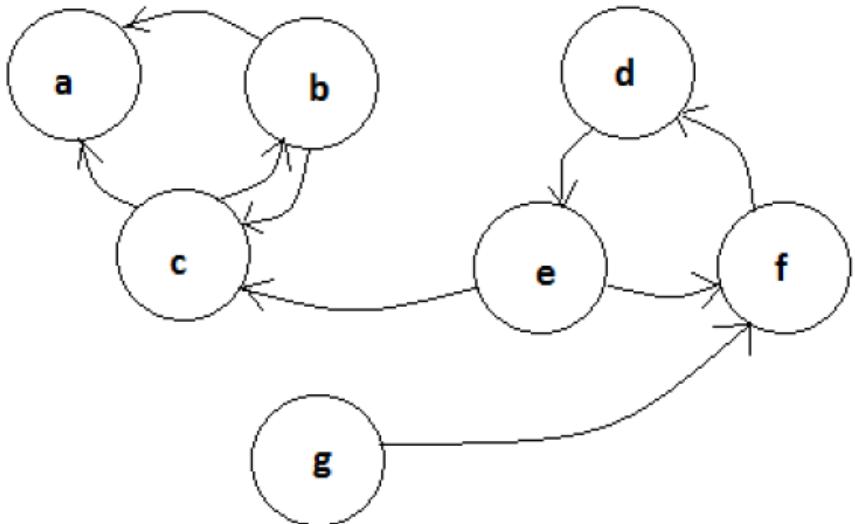
Entonces, tal como habíamos observado en el gráfico al principio, los puntos de articulación o vértices de corte de este grafo son: a, c, f, h.

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

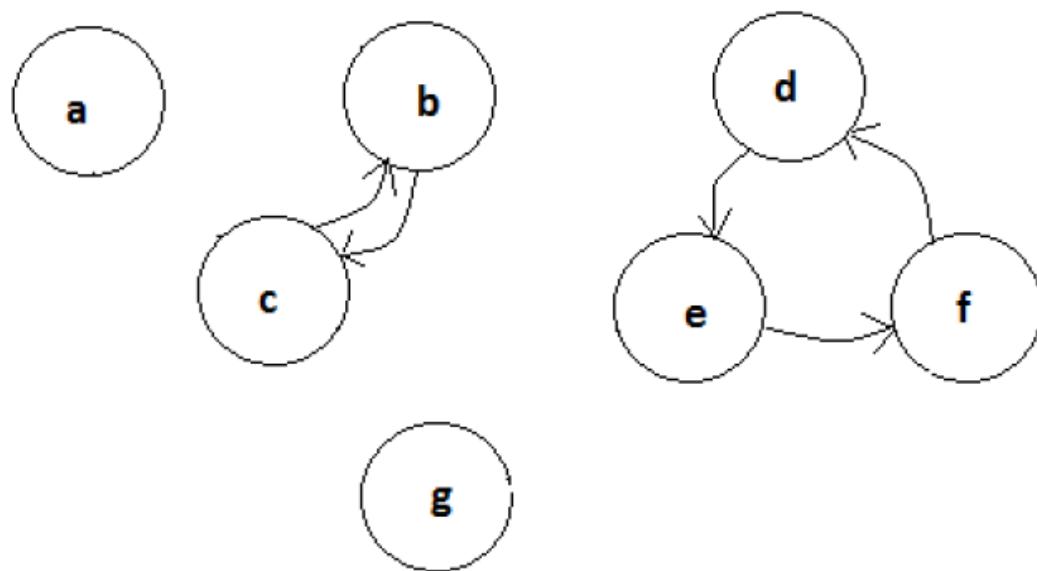
La búsqueda en profundidad también puede usarse para determinar con eficiencia las componentes fuertemente conexas de un conjunto.

Una componente fuertemente conexo de un grafo dirigido es un conjunto maximal de vértices en el que existe camino entre cualquier par de vértices.

Ejemplo: en la siguiente figura



Se muestran las componentes conexas; se han mantenido los enlaces entre vértices de cada componente:

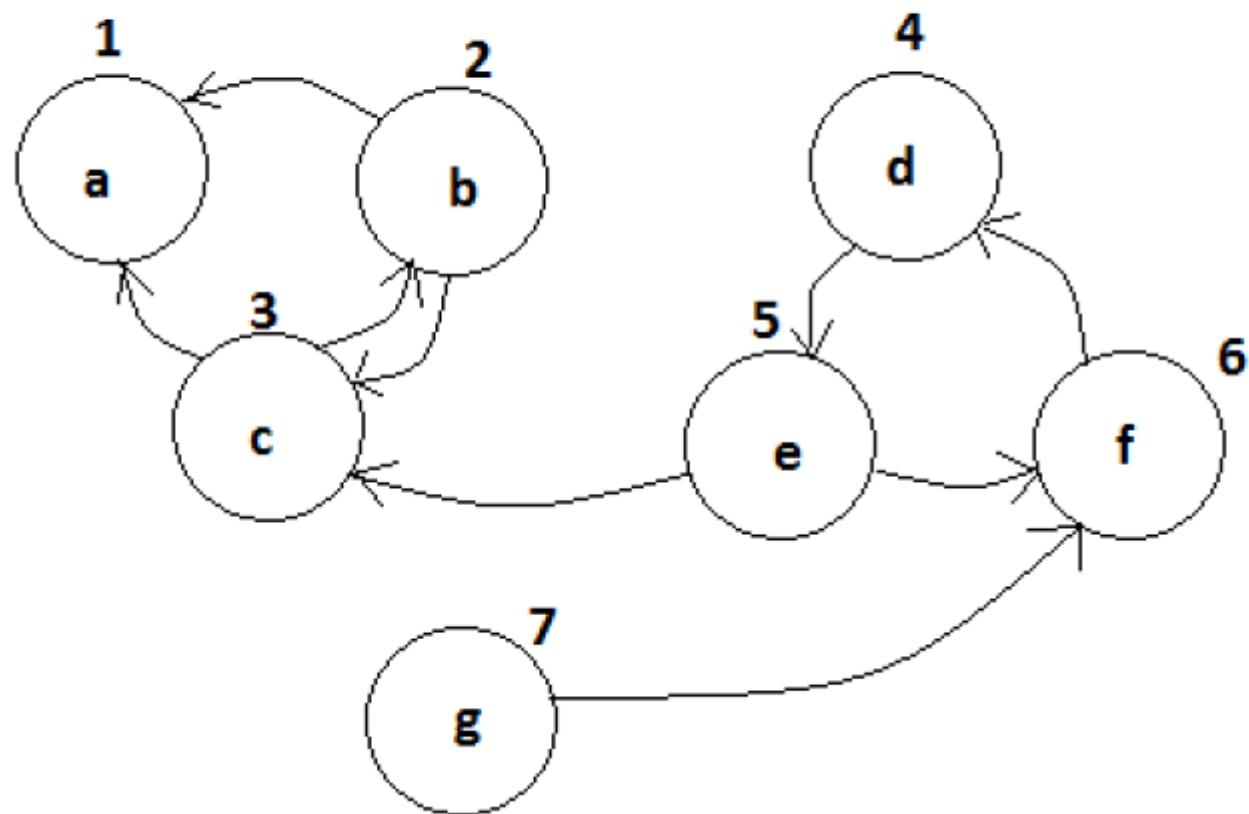


El algoritmo para hallar las componentes conexas de un grafo dirigido **G** es el siguiente:

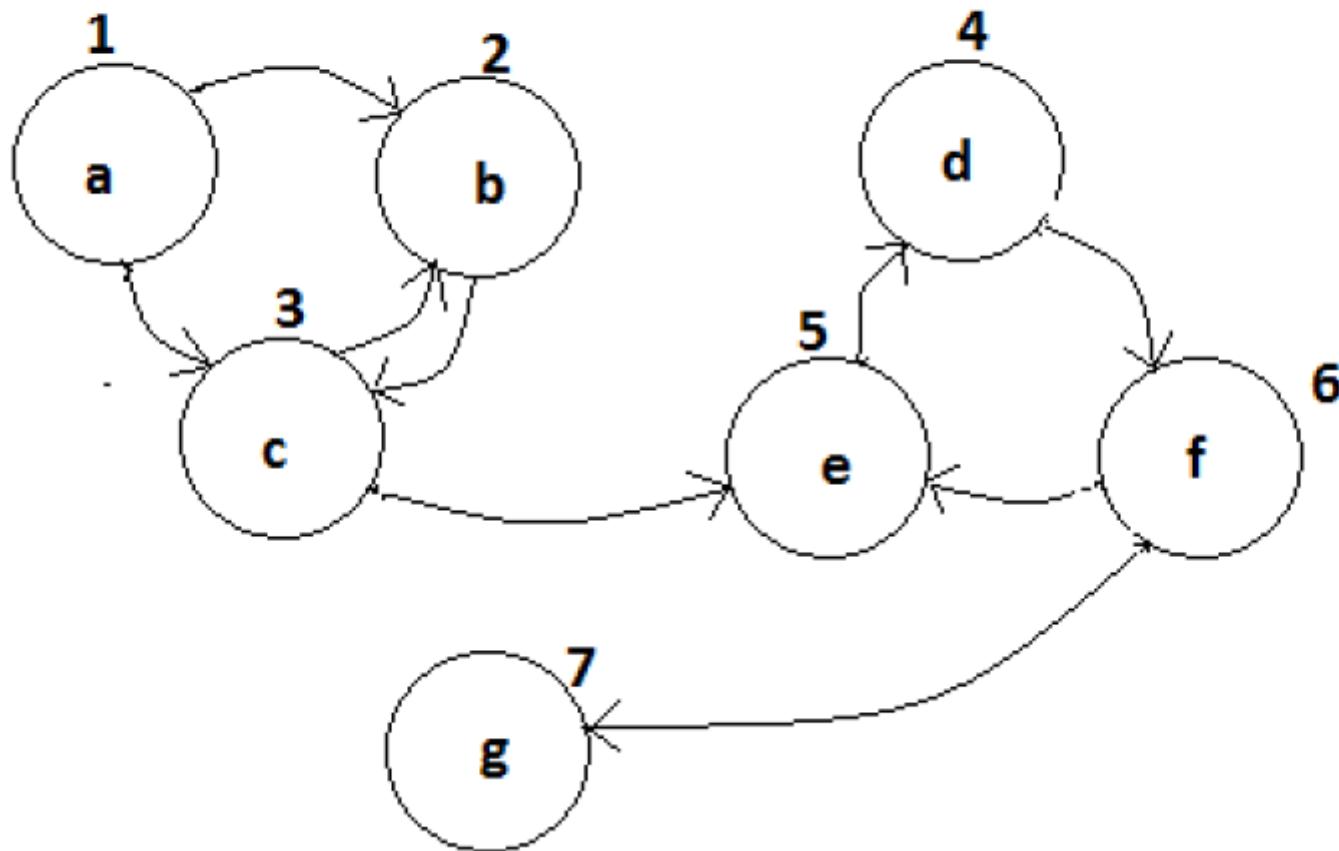
- Realizar un recorrido en profundidad con numeración de vértices en **G**
- Construir un grafo dirigido nuevo **G'**, invirtiendo las direcciones de todas las aristas de **G**.
- Realizar un recorrido en profundidad en **G'**, partiendo del vértice con numeración más alta, de acuerdo con la numeración asignada en primer paso 1. Cuando termina un recorrido parcial tomando los vértices de modo decreciente, se continua con el siguiente vértice con numeración más alta.

Al finalizar el recorrido, cada árbol correspondiente a un recorrido parcial es una componente fuertemente conexa del grafo.

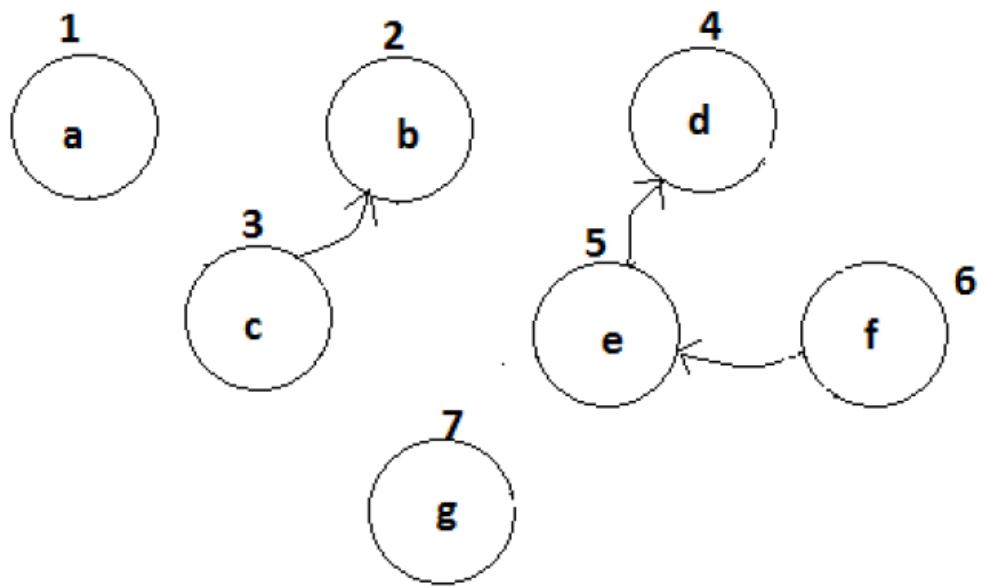
Primero, asignamos la numeración con el recorrido en profundidad en G.



Luego invertimos el sentido de las aristas y obtenemos G' .



Recorremos G' en profundidad, comenzando por el vértice 7 y terminando en el 1. Obtenemos:



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 a continuación un algoritmo que recorre en anchura numerando los vértices en el orden en que se visitan.

```
//Se considera que cada vértice tiene un atributo número //que almacena el orden en que se lo visitó.  
//Inicialmente todos los atributos numero valen 0, lo cual indica que no fueron visitados  
//Índice es una variable que permite realizar el conteo de los nodos visitados para asignar el valor a  
// cada atributo número de cada vértice  
// Este algoritmo utiliza una cola inicialmente vacía
```

BFSIterativoConNumeracion

{

Incializar cola c a vacío;

índice=0;

Para cada vértice v

{

Asignar 0 a numero de v

}

Mientras exista un vértice v tal que numero de v sea 0

{

 indice=índice+1;

 asignar índice a numero de v;

 Acolar v en cola c;

 Mientras cola no vacía

{

 Desacolar en v;

 Para cada vértice u adyacente a v

{

 Si (numero de u es 0)

{

 indice=índice+1;

 asignar índice a número de u;

 Acolar u en cola c;

}

}

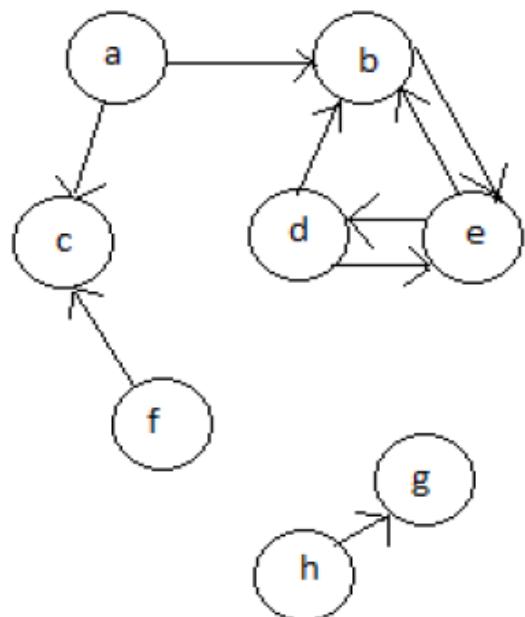
}

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 acollarlo 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:

Los recorridos topológicos son proceso de asignación de un orden lineal a los vértices de un grafo dirigido acíclico, de modo que se respeten las precedencias indicadas por las relaciones de adyacencia.

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 lis en la cual quedan insertados los nodos según fueron visitados en el recorrido)
TopologicoDFS
{
    inicializar lista de salida lis a vacío; //lista vacía
    para cada vértice v
    {
        marcarlo como no visitado
    }
    para cada vértice v
    {
        si (v no visitado) entonces
        {
            Rec (v, lis) //lis pasa por referencia
        }
        retornar lis;
    }
}
```

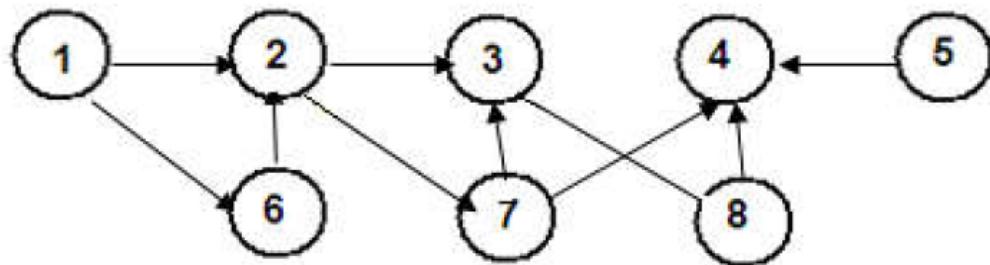
```
Rec ( v: vertice , var lis: lista)
{
    marcar v como visitado
    para cada vértice w adyacente a v
    {
        si (w no visitado) entonces
        {
            Rec(w, lis)
        }
        insertar v al frente en lista lis
    }
}
```

Análisis de coste temporal:

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

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

Recorrido topológico en anchura

Se comienza calculando, para cada vértice, el grado de entrada del mismo (número de aristas que finalizan 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.

// Este algoritmo genera una lista de salida lis, con todos los vértices del grafo

// La tabla T almacena para cada vértice su grado de entrada. Se va actualizando durante el proceso.

```

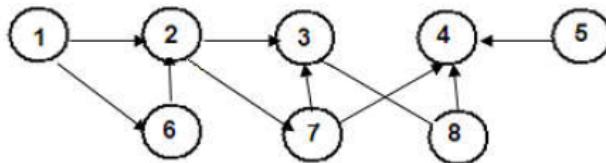
TopologicoBFS
{
    inicializar table T;
    Inicializar cola c a vacío; // cola para vértices del grafo; se inicializa vacía.
    Inicializar lista lis a vacío, lista de salida;
    Para cada vértice v
    {
        Almacenar el grado de entrada de v en la table T;
        Si (grado de entrada de v es 0)
        {
            Acolar v en cola c;
        }
    }
    Mientras (cola c no vacía)
    {
        Desacolar en x;
        Insertar x al final en lista;
        Para cada vértice w adyacente a v
        {
            Decrementar en 1 el grado de w registrado en la tabla T;
            si (grado de w registrado en tabla T es 0)
            {
                Acolar w en cola c;
            }
        }
    }
    Retornar lista lis;
}

```

Análisis de coste temporal: es análogo al de los algoritmos que aplican recorridos y que hemos considerado antes.

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

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 llamada *greedy*, voraz, o ávida. Se exige, para aplicar esta estrategia en este problema que el peso de las aristas sea no negativo.

La estrategia greedy trata de optimizar (alcanzar el máximo o el mínimo) una función objetivo, la cual depende de ciertas variables. Cada una de estas variables tiene un determinado dominio y está sujeta a restricciones. La solución está asociada a una sucesión de decisiones. En cada etapa se toma una decisión que no tiene vuelta atrás, y que involucra elegir el elemento más promisorio (la mejor opción de las disponibles, el candidato que parece ofrecer más posibilidades para 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 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 óptimo 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).

El problema a tratar tiene, entonces este enunciado:

Dado un grafo dirigido $G = (V, A)$, con aristas ponderadas, y considerando un vértice como origen, determinar los costos de los caminos 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:

Consideramos el grafo orientado G con aristas ponderadas no negativas. Vamos a considerarlos vértices etiquetados con valores $1..n$ y también que el vértice de partida es el 1.

La función a minimizar es la de costes desde el origen a cada uno de los restantes vértices.

En un vector D vamos a almacenar los costes actualizados a medida que avanza la ejecución del algoritmo.

Nuestro conjunto de candidatos es el de los vértices del grafo no visitados. Desde el comienzo en ese conjunto no estará el vértice de inicio, ya que de ahí partimos.

En cada etapa del algoritmo seleccionamos un vértice cuyo coste desde el origen sea mínimo. Este coste inicialmente es el que indican las aristas que parten del vértice origen, pero se va actualizando porque podemos bajarlo “pasando” por otro vértice de coste menor antes.

El algoritmo termina cuando el conjunto de candidatos queda sin elementos.

```
// G = (V, A) es el grafo dirigido con aristas no negativas
```

```
//M es la matriz de pesos
```

```
//S es el conjunto de vértices visitados a lo largo del procesamiento.
```

```
// D es un vector de n-1 elementos para calcular los valores de los respectivos caminos mínimos
```

```

Caminos_Minimos_Dijkstra
{
// comenzamos con el vértice origen en el conjunto de vértices procesados
 inicializar el conjunto S en { 1 }
Para cada vértice i desde 2 hasta N hacer
{
    //si hay arista (1, i ) almacenamos su peso en la posición correspondiente de D
    // o bien un peso máximo si no hay arista
    D[i] = M[1][i]
}
//mientras haya candidatos que considerar, elegimos el más promisorio y analizamos condiciones
Mientras (conjunto de vértices no visitados V-S no esté vacío)
{
    Elegir vértice w perteneciente a V – S tal que D[w] sea mínimo
    //Agregamos w al conjunto de vértices que fueron visitados
    S= S U {w}
    para todo vértice v adyacente a w
    {
        //Se registran las eventuales mejoras en aquellos v
        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 $O(v)$.

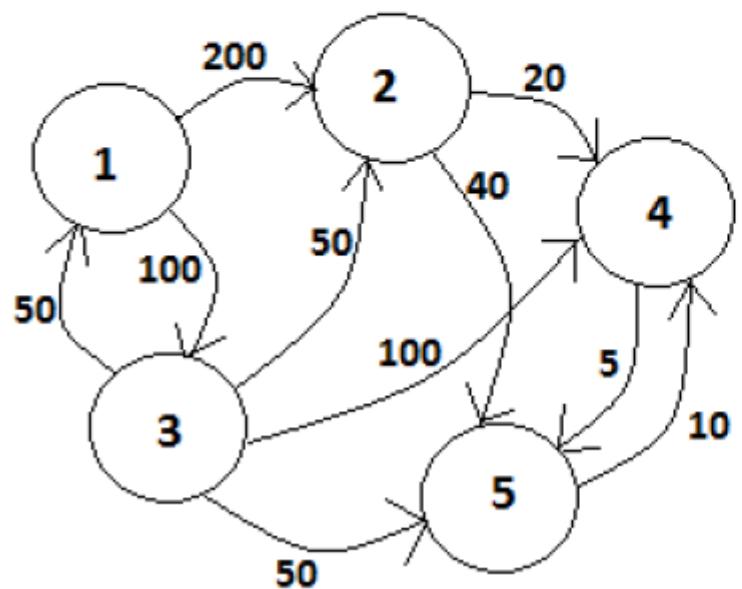
La función de elección de un vértice de coste mínimo no visitado implica recorrer tanto la tabla de visitados como el vector D, es $O(v)$.

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 versión del algoritmo desarrollada antes. Por ejemplo:



La tabla o matriz de pesos M es:

	1	2	3	4	5
1	0	200	100	∞	∞
2	∞	0	∞	20	40
3	50	50	0	100	50
4	∞	∞	∞	0	5
5	∞	∞	∞	10	0

El vector D donde registramos los sucesivos advances en el algoritmo, inicialmente es:

2	3	4	5
200	100	∞	∞

Hemos indicado en una fila las etiquetas de los nodos y debajo, sus costos. Estos valores iniciales los tomamos de la fila correspondiente al origen, en la table T. Los valores ∞ indican que no hay arista de enlace entre esos vértices.

El conjunto de candidatos V-S es {2, 3, 4, 5}. El vértice “más promisorio” de los que no fueron visitados es el de coste más bajo, en este caso el 3.

Seleccionamos 3, lo sacamos del conjunto de candidatos, y analizamos cuales son los ayacentes a 3 para ver si alguno puede tener mejora.

3 tiene como adyacentes a 1, 2, 4 y 5.

El coste de alcanzar 2 desde 3 es 50, por tanto, podemos llegar desde 1 hasta 2 pasando por 3 a coste 150 (100 es el coste de la arista (1,3) y 50 el coste de la arista (3, 2)). Este valor mejora el 200 de la table, por lo que lo mejoramos.

El coste de alcanzar 4 desde 3 es 100, entonces podemos llegar desde 1 hasta 4 a coste 20 (100 coste de arista (1, 3) más 100 de coste de arista (3,4)). Antes el coste era ∞ porque no había camino, por tanto mejoramos la situación.

El coste de alcanzar 5 desde 3 es 50, así que podemos llegar desde 1 hasta 5 pasando por 3 a coste 150 (100 es el coste de la arista (1,3) y 50 el coste de la arista (3, 5)). Esto mejora el ∞ anterior, mejoramos el valor.

Entonces el vector de costes luego de esta primera etapa queda:

2	3	4	5
150	100	200	150

De los vértices no visitados, (el conjunto es ahora $\{2, 4, 5\}$) hay dos de coste mínimo: 2 y 5. Podemos elegir cualquiera de ellos. Por ejemplo, el 2, que sacamos del conjunto de candidatos.

Con el 2, resulta lo siguiente: los vértices adyacentes son 4 y 5.

El costo de alcanzar 4 desde 2 es 20. Por lo que se puede alcanzar 4 desde 1 a coste 170, ya que 150 es el costo $(1, 2)$ y 20 es el costo $(2, 4)$.

El costo de alcanzar 5 desde 2 es 40. Así que se alcanzaría 5 desde 1 pasando por 2 a coste 190, porque es 150 el costo de $(1, 2)$ más 40 de costo de $(2, 5)$. Pero el costo previo era 150, el nuevo valor (190) no lo mejora, por lo tanto no lo registramos, no es una mejora.

Entonces el vector de costes queda:

2	3	4	5
150	100	200	150

Ahora, de los vértices no visitados ($\{4, 5\}$) el más promisorio es el 5.

5 tiene como adyacente a 4. El coste de la arista $(5, 4)$ es 10. Entonces puede alcanzarse el vértice 4 desde 1 a un coste más bajo que el que teníamos. Ahora, tenemos 150 para llegar a 5 desde 1 más 10 para alcanzar 4 desde 5.

Registraremos la mejora

2	3	4	5
150	100	200	150

Nos ha quedado solo el 4 en el conjunto de candidatos. Lo retiramos y hacemos los análisis correspondientes, pero observamos que no se produce ninguna mejora.

Los costos mínimos hasta 2, 3, 4 y 5 son 150, 100, 200 y 150 respectivamente.

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

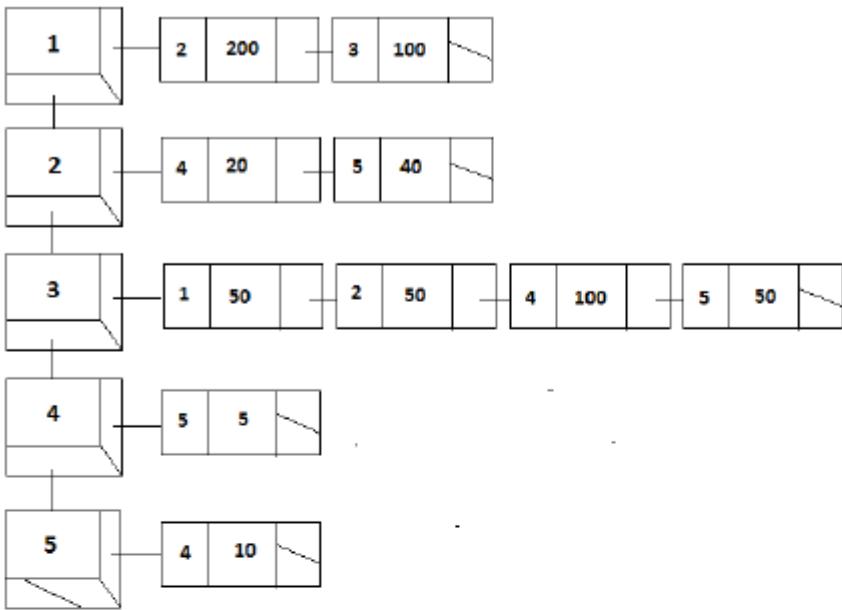
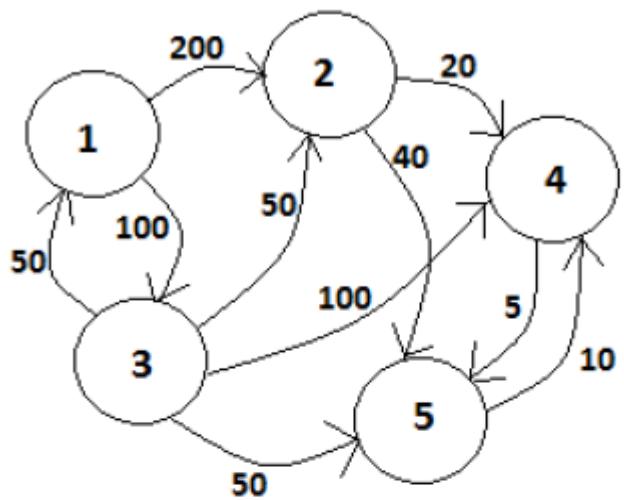
En el algoritmo que hemos descripto hay dos circunstancias que influyen mucho en la eficiencia: uno de ellos es la función de elección del más promisorio. Si tenemos los vértices almacenados en un array “común” con sus costes, hay que recorrer el array en cada etapa para establecer cual es el no visitado de coste mínimo. Eso implica un coste $O(n)$ en cada etapa. Puede disminuirse usando por ejemplo una cola con prioridad, que puede almacenarse en un heap de mínimo implementado en array. En esta cola la prioridad está dada por el coste de alcanzar el vértice desde el origen. Esto disminuiría el coste de la elección a $O(\log n)$, puesto que la elección del más promisorio corresponde a la baja de la raíz del heap, que puede hacerse en tiempo logarítmico.

Ahora bien, hay que considerar también que cada vez que se registre una mejora en el coste de alcanzar un vértice, si el vértice en cuestión está en el heap, se debe registrar la mejora, lo cual se refleja en un movimiento del nodo del heap, que se va mover en dirección a la raíz (lo reubicaremos comparándolo con el padre e intercambiando si es necesario, y luego comparando con el nuevo padre y así sucesivamente, como hacemos en un alta en el heap, pero en este caso con un nodo que ya estaba).

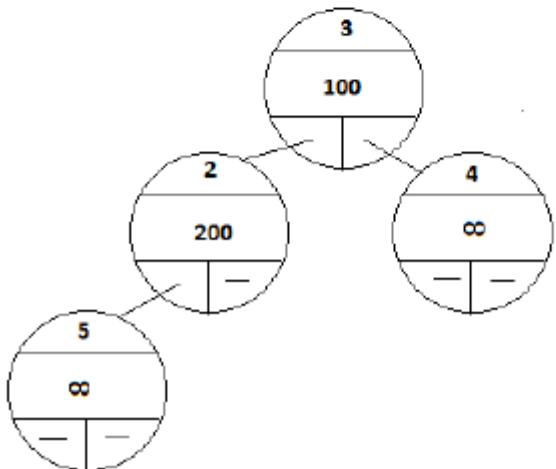
Además podemos implementar el grafo en listas de adyacencia, de modo que cuando determinemos quienes son los adyacentes de un vértice, en lugar de revisar $n-1$ celdas (que pueden no ser de adyacentes), sólo recorramos la lista de adyacentes.

Inicialmente se construye el heap de vértices con sus costes iniciales (esto, como hemos visto antes, puede hacerse a coste $O(n)$) y se inicializa el vector de costes con los valores de la lista de adyacentes del vértice origen.

Mientras el heap con candidatos no se vacíe, se elige al mejor (el de la raíz) 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 registra y si el vértice mejorado está en el heap, se modifica su prioridad en el heap y se restaura nuevamente la estructura.



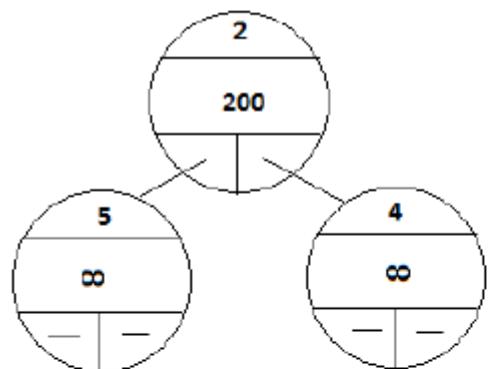
Por otro lado, el heap con los candidatos y sus costos inicialmente es:



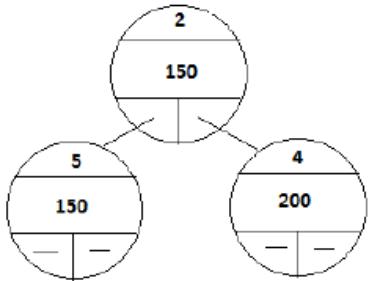
La tabla de costes inicial es

2	3	4	5
200	100	∞	∞

Eliminamos la raíz del heap y lo recomponemos; quedará así:



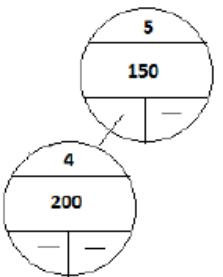
El 3, valor elegido, permite mejorar el valor de 2, a 150, luego 4 cambia a 200 y 5 a 150. Estas modificaciones se reflejan en el heap, que quedará así:



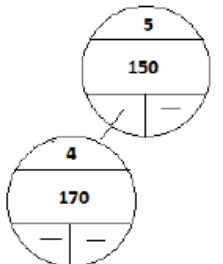
Y el vector de costos quedará:

2	3	4	5
150	100	200	150

Nuevamente eliminamos la raíz del heap y lo recomponemos; queda:



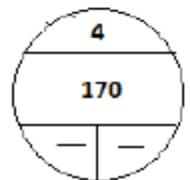
Recorremos la lista de adyacentes a 2, se mejora 4 y esto se refleja en el heap, que queda:



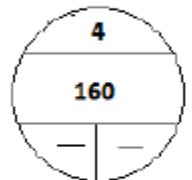
El array de costos quedará así:

2	3	4	5
150	100	200	150

Eliminamos nuevamente la raíz del heap, queda



Con 5 el coste de llegar a 4 baja, quedando el heap así:



Finalmente eliminamos le ultimo nodo del heap, pero no se obtiene ninguna mejora.

El array, como habíamos visto queda finalmente así:

2	3	4	5
150	100	200	150

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 suele resolverse aplicando la estrategia de “Programación Dinámica”.

La programación dinámica suele aplicarse también a problemas de optimización. 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).

Programación Dinámica

La programación dinámica (DP, por sus siglas en inglés) es una poderosa técnica de resolución de problemas en computación. Se utiliza cuando un problema puede descomponerse en subproblemas más pequeños y solapados, y sus soluciones parciales pueden reutilizarse para resolver el problema más grande.

Definición Formal

La programación dinámica es un método para resolver problemas de optimización al dividirlos en subproblemas más pequeños, resolviendo cada uno una sola vez, y almacenando sus resultados para evitar cálculos repetidos.

Se basa en dos principios fundamentales:

1. Subestructura Óptima:

- La solución óptima de un problema se compone de las soluciones óptimas de sus subproblemas.

2. Superposición de Subproblemas:

- El problema general puede dividirse en subproblemas que se resuelven muchas veces (es decir, los subproblemas se solapan).

💡 ¿Cómo Funciona?

1. **Dividir:** Se descompone el problema principal en subproblemas más pequeños.
2. **Resolver:** Se resuelven los subproblemas, comenzando por los más simples.
3. **Almacenar:** Se guardan las soluciones de los subproblemas (en tablas o estructuras similares) para evitar cálculos redundantes.
4. **Combinar:** Se combinan las soluciones de los subproblemas para formar la solución del problema completo.

🛠️ Técnicas Principales

Existen dos formas de aplicar la programación dinámica:

1. **Top-Down (Memorización):**
 - Resolver el problema principal de forma recursiva.
 - Almacenar las soluciones de los subproblemas en una tabla (normalmente un array o un diccionario).
 - Si un subproblema ya fue resuelto, simplemente se reutiliza la solución almacenada.
2. **Bottom-Up (Tabulación):**
 - Resolver primero los subproblemas más pequeños.
 - Construir la solución del problema principal iterativamente usando una tabla.
 - Es más eficiente en cuanto al uso de memoria.

Aplicaciones Clásicas de la Programación Dinámica

1. Problemas de Secuencias:

- Longest Common Subsequence (LCS)
- Longest Increasing Subsequence (LIS)

2. Optimización:

- Knapsack Problem (Mochila)
- Minimum Coin Change

3. Grafos:

- Floyd-Warshall (como vimos antes).
- Caminos más cortos en grafos.

4. Cadenas y Textos:

- Edit Distance (mínimos cambios para convertir una cadena en otra).
- Palíndromos más largos.

5. Problemas Combinatorios:

- Caminos en una cuadrícula (Grid Paths).
- Número de formas de subir una escalera.

Ventajas:

- Resuelve problemas que serían costosos con métodos puramente recursivos.
- Reduce la redundancia de cálculos al reutilizar resultados.

Desventajas:

- Puede requerir más memoria para almacenar las soluciones de los subproblemas.
- Difícil de aplicar si no se identifica una subestructura óptima.

El algoritmo de **Floyd-Warshall** es un método clásico en la programación dinámica utilizado para resolver el problema de las distancias más cortas entre todos los pares de nodos en un grafo ponderado. Vamos a desglosarlo paso a paso:

Descripción

1. Tipo de Grafo:

- Funciona con grafos dirigidos o no dirigidos.
- Los pesos de las aristas pueden ser positivos o negativos, pero no debe haber ciclos de peso negativo (es decir, que sumen un peso total negativo al recorrerlos).

2. Entrada del Algoritmo:

- Una matriz de adyacencia `dist` donde:
 - $\text{dist}[i][j]$ representa el peso de la arista desde el nodo i al nodo j .
 - Si no hay arista entre i y j , se usa un valor infinito (∞).

3. Salida:

- Una matriz donde $\text{dist}[i][j]$ contiene la distancia más corta desde i hasta j .

4. Idea Central:

- El algoritmo intenta mejorar gradualmente las distancias considerando nodos intermedios.
- Para cada par de nodos i y j , verifica si pasar por un nodo intermedio k reduce la distancia actual entre ellos.

 **Pasos del Algoritmo**

Supongamos que tienes n nodos numerados de 0 a $n - 1$.

1. Inicialización:

- Si $i == j$, entonces $\text{dist}[i][j] = 0$ (la distancia de un nodo a sí mismo es 0).
- Si hay una arista directa entre i y j , asigna su peso a $\text{dist}[i][j]$.
- Si no hay conexión directa, $\text{dist}[i][j] = \infty$.

2. Actualización:

- Para cada nodo intermedio k (del 0 al $n - 1$):
 - Para cada par de nodos i y j :
 - Actualiza $\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$.

3. Resultado:

- Al final del proceso, $\text{dist}[i][j]$ contendrá la distancia más corta entre i y j , o ∞ si no están conectados.

 **Ejemplo**

Supongamos un grafo con 4 nodos y la siguiente matriz inicial:

$$\text{dist} = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

1. **Inicialización:** La matriz inicial ya tiene los valores dados.
2. **Iteración con nodo intermedio $k = 0, 1, 2, 3$:**
 - Se actualiza la matriz dist comparando rutas posibles pasando por k .

Al final, la matriz de distancias más cortas sería:

$$\text{dist} = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 8 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Ventajas:

- Es simple de implementar.
- Encuentra las distancias más cortas entre todos los pares de nodos.

Desventajas:

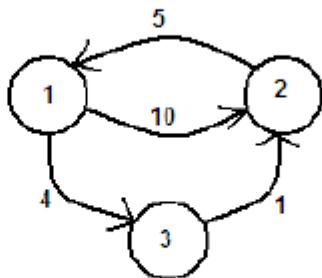
- Complejidad temporal: $O(n^3)$, lo que lo hace poco eficiente para grafos muy grandes.
- No funciona correctamente si hay ciclos de peso negativo.

```
3 public class AlgoritmoFloydMarshall {  
4     static final int INFINITO = 99999; // Representa infinito  
5  
6     public static void floydWarshall(int[][][] grafo) {  
7         int n = grafo.length;  
8         int[][] distancias = new int[n][n];  
9  
.0         // Inicializamos la matriz de distancias con los valores del grafo  
.1         for (int i = 0; i < n; i++) {  
.2             for (int j = 0; j < n; j++) {  
.3                 distancias[i][j] = grafo[i][j];  
.4             }  
.5         }  
.6  
.7         // Aplicamos el algoritmo  
.8         for (int k = 0; k < n; k++) {  
.9             for (int i = 0; i < n; i++) {  
.0                 for (int j = 0; j < n; j++) {  
.1                     // Si el camino a través de k es más corto, actualizamos  
.2                     if (distancias[i][k] + distancias[k][j] < distancias[i][j]) {  
.3                         distancias[i][j] = distancias[i][k] + distancias[k][j];  
.4                     }  
.5                 }  
.6             }  
.7         }  
.8  
.9         // Imprimimos el resultado  
.0         imprimirSoluciones(distancias);  
.1     }  
.2 }
```

```
public static void imprimirSoluciones(int[][][] dist) {
    System.out.println("Matriz de distancias más cortas:");
    for (int[] row : dist) {
        for (int val : row) {
            if (val == INFINITO) {
                System.out.print("INF ");
            } else {
                System.out.print(val + " ");
            }
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int[][][] graph = {
        {0, 3, INFINITO, 7},
        {8, 0, 2, INFINITO},
        {5, INFINITO, 0, 1},
        {2, INFINITO, INFINITO, 0}
    };
    floydWarshall(graph);
}
```

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

El **algoritmo de Warshall** es una técnica eficiente para calcular la **cerradura transitiva** de un grafo dirigido. La **cerradura transitiva** de un grafo indica si es posible alcanzar un nodo j desde un nodo i siguiendo una secuencia de aristas.

Definición y Contexto

¿Qué es la Cerradura Transitiva?

- La **cerradura transitiva** de un grafo es un grafo que tiene la misma cantidad de nodos, pero con una arista adicional (i, j) si existe un camino dirigido desde i hasta j en el grafo original.
- Esencialmente, responde la pregunta: *¿Es posible llegar del nodo i al nodo j ?*

Entrada y Salida del Algoritmo de Warshall

1. Entrada:

- Una matriz de adyacencia A de tamaño $n \times n$, donde:
 - $A[i][j] = 1$ si hay una arista directa de i a j .
 - $A[i][j] = 0$ si no hay arista.

2. Salida:

- Una matriz T , que representa la **cerradura transitiva**, donde $T[i][j] = 1$ si existe un camino de i a j (directo o indirecto).

Cómo Funciona el Algoritmo

1. Inicialización:

- Usa la matriz de adyacencia original como punto de partida.

2. Actualización:

- Para cada nodo intermedio k (del 0 al $n - 1$):

- Para cada par de nodos i y j :

- Actualiza $T[i][j] = T[i][j] \vee (T[i][k] \wedge T[k][j])$.

- Esto significa: *Hay un camino entre i y j si ya existía un camino directo o si puedo ir de i a k y luego de k a j .*

3. Resultado:

- Al final, la matriz T contiene la cerradura transitiva del grafo.

```
3 public class WarshallAlgorithm {
4
5     public static void warshall(int[][] grafo) {
6         int n = grafo.length;
7         int[][] caminos = new int[n][n];
8
9         // Inicializamos la matriz de cierre transitivo con la matriz de adyacencia
10        for (int i = 0; i < n; i++) {
11            for (int j = 0; j < n; j++) {
12                caminos[i][j] = grafo[i][j];
13            }
14        }
15
16        // Aplicamos el algoritmo
17        for (int k = 0; k < n; k++) { // Nodo intermedio
18            for (int i = 0; i < n; i++) { // Nodo de origen
19                for (int j = 0; j < n; j++) { // Nodo de destino
20                    caminos[i][j] = caminos[i][j] | (caminos[i][k] & caminos[k][j]);
21                }
22            }
23        }
24
25        // Imprimimos la matriz de cierre transitivo
26        imprimirMatriz(caminos);
27    }
28}
```

```
public static void imprimirMatriz(int[][] matrix) {
    System.out.println("Cerradura transitiva:");
    for (int[] row : matrix) {
        for (int val : row) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int[][] graph = {
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {0, 0, 0, 1},
        {0, 0, 0, 0}
    };
    warshall(graph);
}
```

```
public static void imprimirMatriz(int[][] matrix) {
    System.out.println("Cerradura transitiva:");
    for (int[] row : matrix) {
        for (int val : row) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    int[][] graph = {
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {0, 0, 0, 1},
        {0, 0, 0, 0}
    };
    warshall(graph);
}
```

Explicación del Código

1. Inicialización:

- Creamos la matriz `reach`, copiando los valores de la matriz de adyacencia `graph`.

2. Iteraciones:

- Por cada nodo intermedio k , evaluamos si hay nuevos caminos entre i y j pasando por k .

3. Operaciones Booleanas:

- `|` (OR lógico): Indica si hay camino directo o indirecto.
- `&` (AND lógico): Evalúa si ambos caminos parciales ($i \rightarrow k$ y $k \rightarrow j$) existen.

4. Salida:

- Al final, imprimimos la matriz de cierre transitivo.

Ejemplo de Entrada y Salida

Entrada:

El grafo representado por la matriz de adyacencia:

$$\text{graph} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Salida:

La matriz de cerradura transitiva es:

$$\text{reach} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esto significa que:

- Desde el nodo 0 se puede alcanzar todos los demás nodos (1, 2, y 3).
- Desde el nodo 1 se puede alcanzar 2 y 3, pero no el nodo 0.
- Desde el nodo 2 solo se puede alcanzar el nodo 3.
- Desde el nodo 3 no se puede alcanzar ningún otro nodo.

Características del Algoritmo

- **Complejidad Temporal:** $O(n^3)$, donde n es el número de nodos.
- **Complejidad Espacial:** $O(n^2)$, ya que usamos una matriz de tamaño $n \times n$.
- Muy eficiente para grafos densos o cuando necesitas determinar rápidamente la accesibilidad entre nodos.

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.
- Si u y v son dos vértices distintos de un árbol, entonces hay un solo camino que los une.

Un grafo con peso o ponderación en las aristas es un grafo $G = (A, V)$, con un costo $C(v, w)$, asociado a cada arista (v, w) perteneciente a A .

La representación por matriz de adyacencia la haremos colocando el peso de la arista en la celda de la misma.

Para la representación por lista de adyacencia, agregaremos un atributo peso a cada nodo de la lista de adyacentes.

Definimos como árbol abarcador de un grafo $G = (V, A)$ no dirigido y conexo a un árbol libre que conecta todos los vértices de V .

Árbol abarcador de coste mínimo

Un **árbol abarcador de coste mínimo** (MST, por sus siglas en inglés: Minimum Spanning Tree) es un subgrafo de un grafo no dirigido, ponderado y conectado, que conecta todos los nodos con el coste total mínimo. Es un concepto fundamental en teoría de grafos y tiene aplicaciones en redes, optimización y diseño de sistemas.

Definición y Propiedades del MST

1. Árbol Abarcador:

- Es un subgrafo que incluye todos los vértices del grafo original.
- No contiene ciclos.
- Es conexo.

2. Coste Mínimo:

- La suma de los pesos de sus aristas es la menor posible entre todos los árboles abarcadores.

3. Condiciones del Grafo:

- El grafo debe ser **conexo**.
- Los pesos de las aristas deben ser conocidos y constantes.

Algoritmos Clásicos para Construir un MST

Hay dos algoritmos principales para construir el MST:

1. **Algoritmo de Kruskal** (basado en conjuntos).
2. **Algoritmo de Prim** (basado en expansión de nodos).

1. Algoritmo de Kruskal

Descripción:

- Construye el MST seleccionando aristas en orden creciente de peso y asegurándose de no formar ciclos.

Pasos:

1. Ordenar todas las aristas por peso en orden ascendente.
2. Inicializar un conjunto para cada nodo (estructura disjunta o *Union-Find*).
3. Iterar sobre las aristas ordenadas:
 - Si la arista conecta dos conjuntos diferentes, añadirla al MST.
 - Unir los conjuntos (nodos) correspondientes.
4. Detenerse cuando el MST tiene $n - 1$ aristas (n es el número de nodos).

Complejidad:

- $O(E \log E)$, donde E es el número de aristas (ordenamiento) y $O(\alpha(V))$ por las operaciones de conjuntos disjuntos (α es la inversa de Ackermann).

```

static void kruskalMST(int[][] graph, int V) {
    List<Edge> edges = new ArrayList<Edge>();

    // Convertimos la matriz de adyacencia en una lista de aristas
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (graph[i][j] != 0) {
                edges.add(new Edge(i, j, graph[i][j]));
            }
        }
    }

    Collections.sort(edges); // Ordenar aristas por peso

    int[] parent = new int[V];
    int[] rank = new int[V];
    for (int i = 0; i < V; i++) {
        parent[i] = i; // Inicializar cada nodo como su propio conjunto
        rank[i] = 0;
    }

    List<Edge> mst = new ArrayList<>();
    for (Edge edge : edges) {
        int u = find(parent, edge.src);
        int v = find(parent, edge.dest);

        if (u != v) { // Si no forman ciclo
            mst.add(edge);
            union(parent, rank, u, v);
        }

        if (mst.size() == V - 1) break; // Detenerse cuando el MST está completo
    }

    // Imprimir el MST
    System.out.println("Árbol abarcador de coste mínimo:");
    for (Edge edge : mst) {
        System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight);
    }
}

```

Estrategia Greedy

La estrategia greedy (o estrategia voraz) es un enfoque algorítmico que resuelve problemas tomando decisiones óptimas *locales* en cada paso, con la esperanza de que estas decisiones conduzcan a una solución *globalmente óptima*.

Definición

Un algoritmo greedy toma decisiones paso a paso, eligiendo en cada momento la opción que parece ser la mejor o más prometedora según un criterio definido (generalmente el más barato, el más rápido, el de mayor beneficio, etc.).

Se basa en el principio de **optimización local**:

- "*Si siempre hacemos lo mejor en el momento, alcanzaremos lo mejor en general.*"

Elementos Clave de un Algoritmo Greedy

Para que una estrategia greedy funcione correctamente, el problema debe cumplir ciertas propiedades:

1. Subestructura Óptima:

- Una solución óptima global se puede construir a partir de soluciones óptimas a subproblemas más pequeños.
- Ejemplo: En el problema del Árbol Abarcador de Costo Mínimo (MST), las mejores decisiones para las aristas locales llevan al árbol de menor costo.

2. Propiedad Greedy (Elección Greedy):

- Una decisión tomada en el momento no afecta negativamente las decisiones futuras.
- Ejemplo: En el Problema de la Mochila Fraccional, tomar el objeto con mayor relación valor/peso en cada paso es siempre lo mejor.

Cómo Funciona una Estrategia Greedy

1. Inicializar una solución vacía.
 2. Mientras no se haya alcanzado la solución completa:
 - Elegir la mejor opción disponible (según un criterio definido).
 - Actualizar el estado del problema, reduciendo su complejidad.
 3. Devolver la solución obtenida.
-

Ventajas y Desventajas

Ventajas:

1. **Simplicidad:** Los algoritmos greedy suelen ser fáciles de implementar.
2. **Eficiencia:** Funcionan rápido, ya que no revisan todas las combinaciones posibles (a diferencia de la programación dinámica o fuerza bruta).
3. **Aplicaciones prácticas:** Muy útiles en problemas que cumplen las propiedades necesarias.

Desventajas:

1. **Subóptimos:** No siempre garantizan una solución globalmente óptima, especialmente si el problema no tiene subestructura óptima.
2. **Específicos:** Solo funcionan para problemas específicos que cumplen con las propiedades greedy.



Ejemplos Clásicos de Algoritmos Greedy

1. Árbol Abarcador de Costo Mínimo (MST):

- Algoritmos de Kruskal y Prim.
- Seleccionan las aristas de menor peso de forma local para construir un árbol globalmente óptimo.

2. Problema de la Mochila Fraccional:

- Selecciona los objetos con mayor relación valor/peso hasta llenar la capacidad de la mochila.

3. Problema del Cambio de Monedas:

- Selecciona la mayor denominación posible de monedas en cada paso para alcanzar un total (funciona si las denominaciones están diseñadas adecuadamente, como en sistemas decimales).

4. Problema del Caminante Avaro (Traveling Salesman Problem - Heurística):

- En cada paso, elige la ciudad más cercana como siguiente destino.

2. Algoritmo de Prim

Descripción:

- Construye el MST creciendo desde un nodo inicial, seleccionando iterativamente la arista de menor peso que conecta un nodo del MST a un nodo fuera del MST.

Pasos:

1. Seleccionar un nodo inicial y agregarlo al MST.
2. Mantener un registro de las aristas candidatas (mínimos pesos de nodos externos al MST).
3. Escoger la arista de menor peso y agregar el nodo al MST.
4. Repetir hasta que todos los nodos estén en el MST.

Complejidad:

- $O(V^2)$ para grafos densos usando una matriz de adyacencia.
- $O(E \log V)$ para grafos dispersos usando un *Min-Heap* (cola de prioridad).

Diferencias entre Kruskal y Prim

Criterio	Kruskal	Prim
Enfoque	Basado en aristas	Basado en nodos
Estructura	Conjuntos disjuntos (<i>Union-Find</i>)	Min-Heap
Eficiencia	Mejor para grafos dispersos	Mejor para grafos densos
Ordenamiento	Ordena aristas por peso	No requiere ordenamiento explícito

```

5 public class Prim {
6     static int findMinVertex(int[] key, boolean[] mstSet, int V) {
7         int min = Integer.MAX_VALUE, minIndex = -1;
8         for (int v = 0; v < V; v++) {
9             if (!mstSet[v] && key[v] < min) {
10                 min = key[v];
11                 minIndex = v;
12             }
13         }
14         return minIndex;
15     }
16
17     static void primMST(int[][] graph, int V) {
18         int[] parent = new int[V]; // Para almacenar el MST
19         int[] key = new int[V]; // Para rastrear los pesos mínimos
20         boolean[] mstSet = new boolean[V]; // Nodos incluidos en el MST
21
22         // Inicializar todas las claves como infinito y el MST como falso
23         Arrays.fill(key, Integer.MAX_VALUE);
24         Arrays.fill(mstSet, false);
25
26         key[0] = 0; // Comenzar con el primer nodo
27         parent[0] = -1; // El nodo raíz no tiene parent
28
29         for (int count = 0; count < V - 1; count++) {
30             int u = findMinVertex(key, mstSet, V); // Obtener el nodo con la clave mínima
31             mstSet[u] = true; // Incluir este nodo en el MST
32
33             // Actualizar las claves y padres de los nodos adyacentes
34             for (int v = 0; v < V; v++) {
35                 if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {
36                     parent[v] = u;
37                     key[v] = graph[u][v];
38                 }
39             }
40         }
41
42         // Imprimir el MST
43         printMST(parent, graph, V);
44     }

```

```
static void printMST(int[] parent, int[][] graph, int V) {
    System.out.println("Árbol abarcador de coste mínimo (Prim):");
    System.out.println("Arista\tPeso");
    for (int i = 1; i < V; i++) {
        System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]));
    }
}

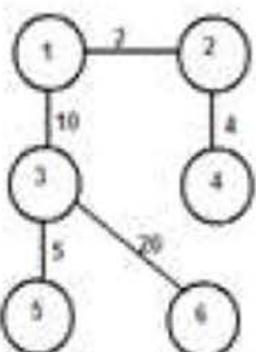
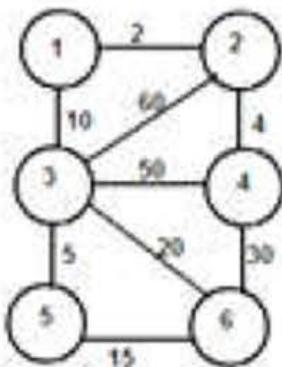
public static void main(String[] args) {
    int[][] graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    int V = graph.length;
    primMST(graph, V);
}
```

Ejemplo grafico

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.



$$\text{Coste: } 2+10+4+5+20=41$$

Mientras **U** distinto de **V** hacer:

{

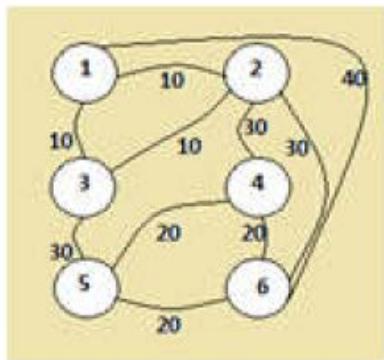
Determinar la arista de costo mínimo, (u, v) tal que $u \in U$ y $v \in (V - U)$

Agregar a **A** la arista (u, v)

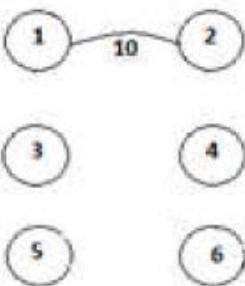
Agregar a **U** el vértice **v**

}

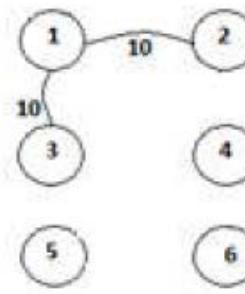
}



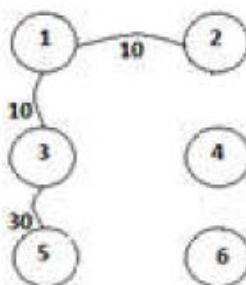
Grafo inicial



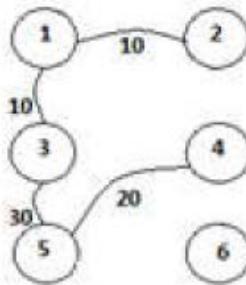
Paso 1



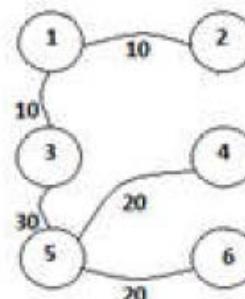
Paso 2



Paso 3

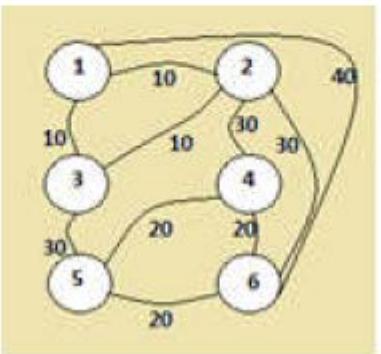


Paso 4

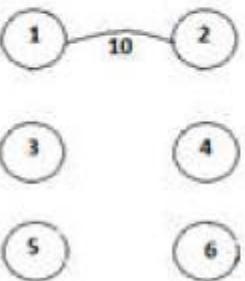


Paso 5. **árbol abarcador de coste mínimo.**
Costo: 90

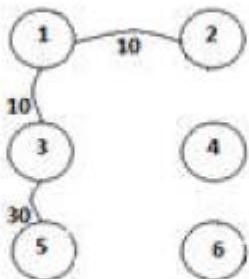
Algoritmo de Kruskal



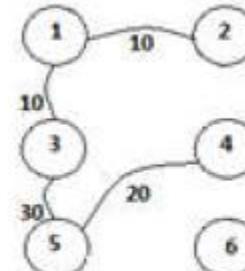
Grafo inicial



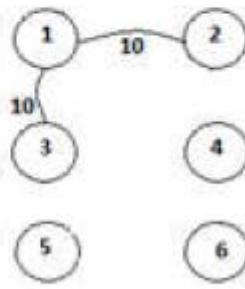
Paso 1



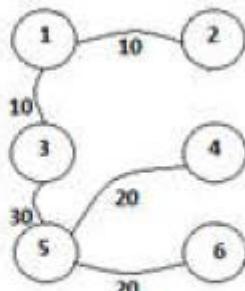
Paso 3



Paso 4

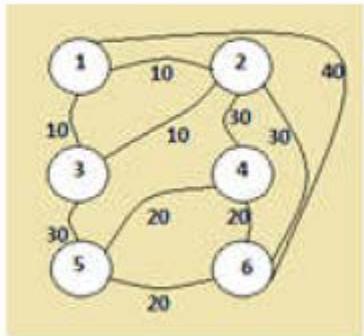


Paso 2

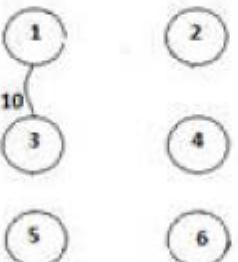


Paso 5. árbol abarcador
de coste mínimo.
Costa: 90

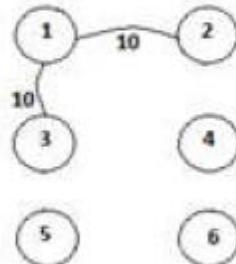
Ejemplo: para el grafo ya considerado,



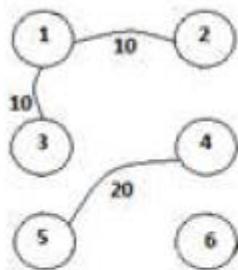
Grafo inicial



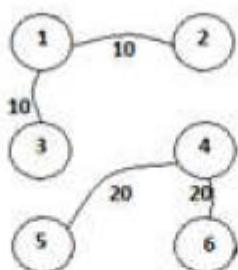
Paso 1



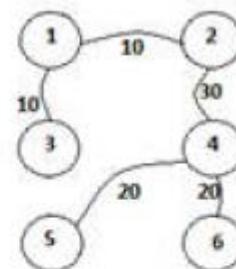
Paso 2



Paso 3
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

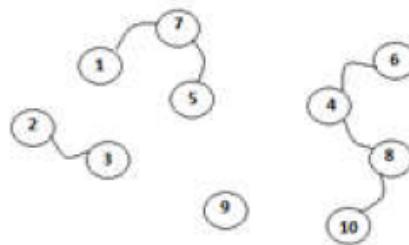


Paso 5. Árbol abarcador
de coste mínimo.
Coste: 90

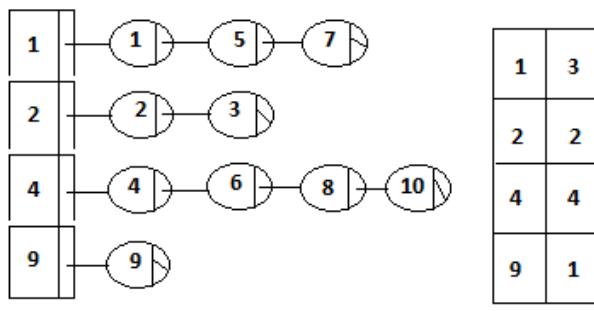
Posibles implementaciones para Union y Find:

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,

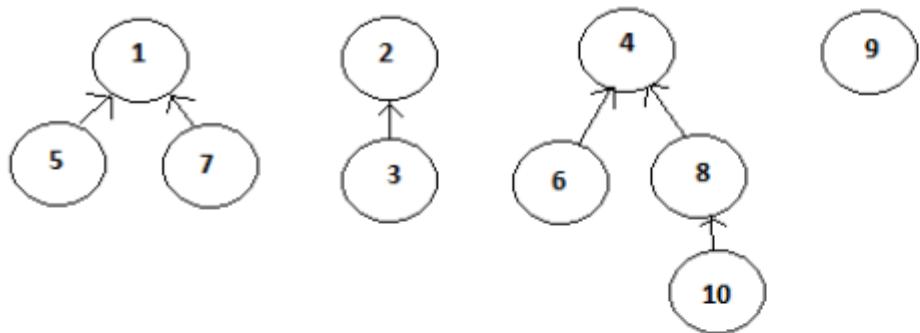


1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

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 ($\text{Find}(x) == \text{Find}(y)$)

{ Union (x, y); }

}

¿Cuántas veces se ejecuta Union?

¿Y Find?

Ford-Fulkerson

El algoritmo de **Ford-Fulkerson** es un método utilizado para encontrar el **flujo máximo** en una red de flujo. Una red de flujo está compuesta por:

1. Un **grafo dirigido**.
2. Un nodo fuente (s) desde donde inicia el flujo.
3. Un nodo sumidero (t) donde termina el flujo.
4. Capacidades en las aristas que limitan cuánto flujo puede pasar por ellas.

Conceptos básicos

Antes de entender el algoritmo, es importante manejar algunos conceptos:

1. Flujo (f):

- Cantidad de flujo que pasa por una arista.
- Debe cumplir:
 - $f(u, v) \leq c(u, v)$, donde $c(u, v)$ es la capacidad de la arista de u a v .
 - Conservación del flujo: la suma del flujo que entra a un nodo (excepto s y t) debe ser igual a la suma del flujo que sale.

2. Capacidad residual:

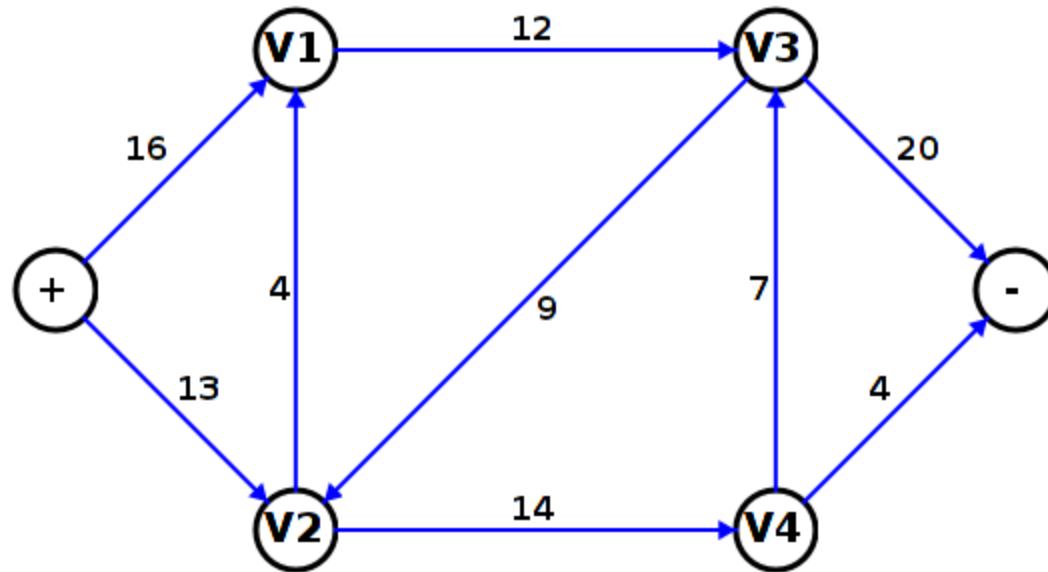
- Es la cantidad de flujo adicional que puede pasar por una arista.
- Si $f(u, v)$ es el flujo actual, la capacidad residual $r(u, v)$ se calcula como:

$$r(u, v) = c(u, v) - f(u, v)$$

3. Camino aumentante:

- Un camino desde s a t en el grafo residual donde todas las aristas tienen capacidad residual positiva.

Ejemplo de flujo del Algoritmo de Ford Fulkerson



Pasos del algoritmo de Ford-Fulkerson

El algoritmo se basa en encontrar caminos aumentantes y aumentar el flujo a través de ellos hasta que no haya más caminos disponibles.

1. Inicialización:

- Comienza con un flujo inicial $f = 0$ en todas las aristas.

2. Construcción del grafo residual:

- Se calcula el grafo residual a partir de las capacidades residuales $r(u, v)$.

3. Búsqueda de un camino aumentante:

- Encuentra un camino desde s hasta t en el grafo residual. Esto se puede hacer mediante una búsqueda en profundidad (DFS) o en amplitud (BFS).

4. Aumentar el flujo:

- Calcula el flujo máximo que puede pasar por ese camino, es decir, el mínimo de las capacidades residuales en el camino.
- Aumenta el flujo en ese camino en el grafo original y ajusta las capacidades residuales.

5. Repetir:

- Repite los pasos 3 y 4 hasta que no haya más caminos aumentantes en el grafo residual.

6. Resultado:

- La suma del flujo que sale del nodo fuente (s) es el flujo máximo.

Ventajas y limitaciones

- **Ventaja:** Es simple y efectivo para grafos pequeños.
- **Limitación:** Si los valores de capacidad son grandes, puede no terminar debido a que usa números fraccionarios (aunque esto se resuelve usando el método Edmonds-Karp, que es una variante de Ford-Fulkerson con BFS).

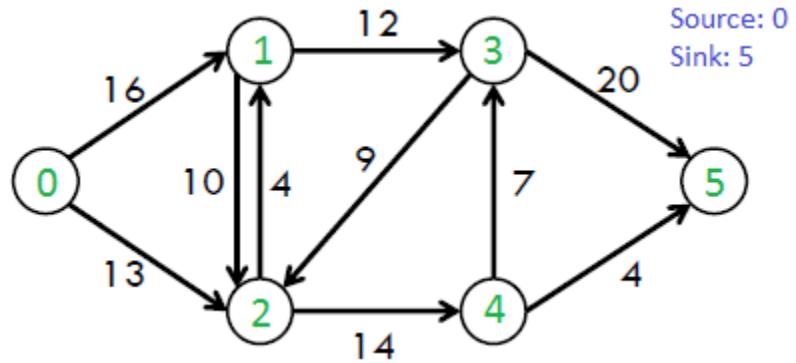
Se denomina flujo al envío o circulación de unidades homogéneas de algún producto.

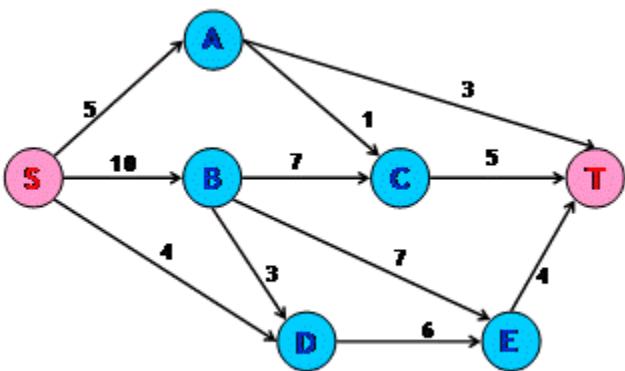
En un grafo que representa un flujo, hay un vértice fuente u origen y otro sumidero o vertedero.

Cada arco tiene una ponderación que corresponde a la capacidad máxima de flujo.

Se quiere enviar desde el origen al sumidero la mayor cantidad posible de flujo de modo que se verifique:

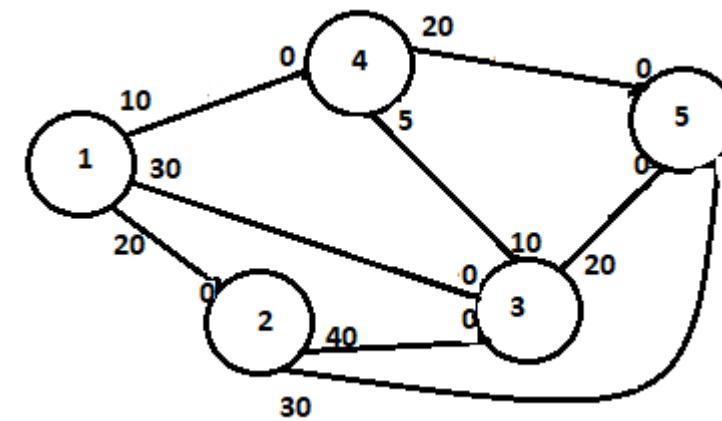
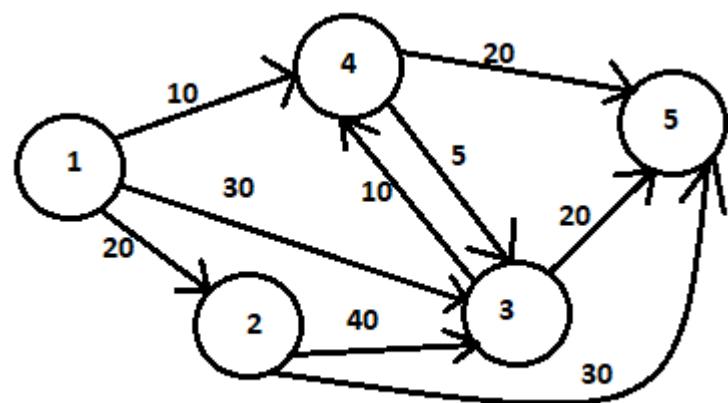
- El flujo siempre es positivo y en unidades enteras
- El flujo a través de un arco siempre es menor o igual que la capacidad del mismo
- El flujo que entra en un vértice es igual al que sale



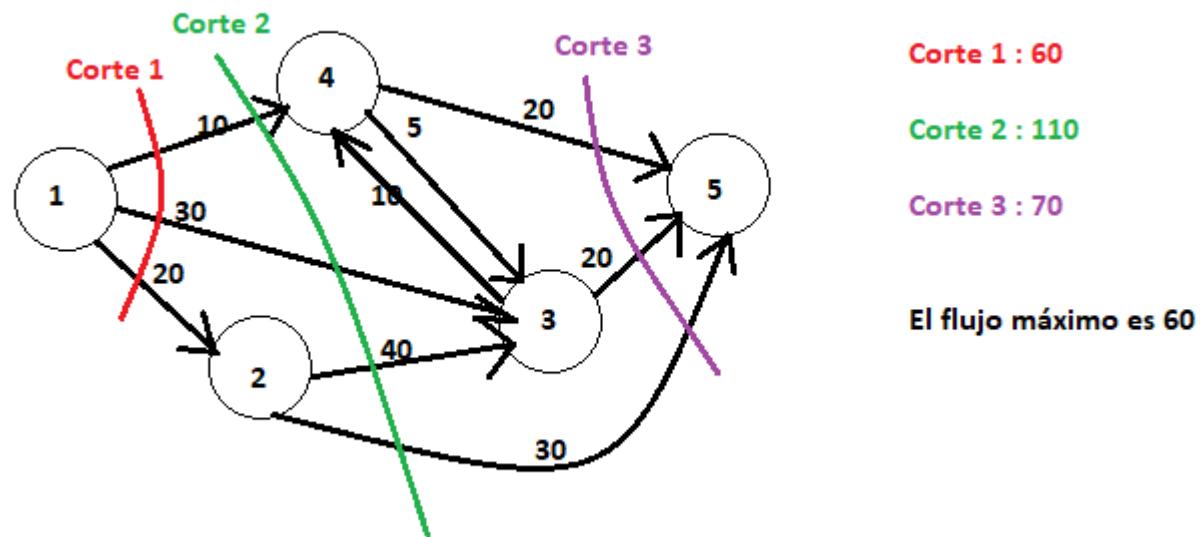


Se denomina corte a una serie de arcos cuya supresión causa una interrupción completa del flujo entre origen y destino.

La capacidad del corte es la sumatoria de las capacidades de los arcos asociados al mismo.



Se demuestra que el flujo máximo de una red es igual a la capacidad del corte de menor capacidad



El algoritmo de Ford Fulkerson propone buscar caminos en los cuales se puede aumentar el flujo hasta que se alcance un máximo.

Para esto, busca una ruta que permita penetrar con flujo positivo neto, que una origen y destino.

Se consideran capacidades iniciales de cada arco que une el vértice i con el vértice j como C_{ij} y C_{ji} respectivamente.

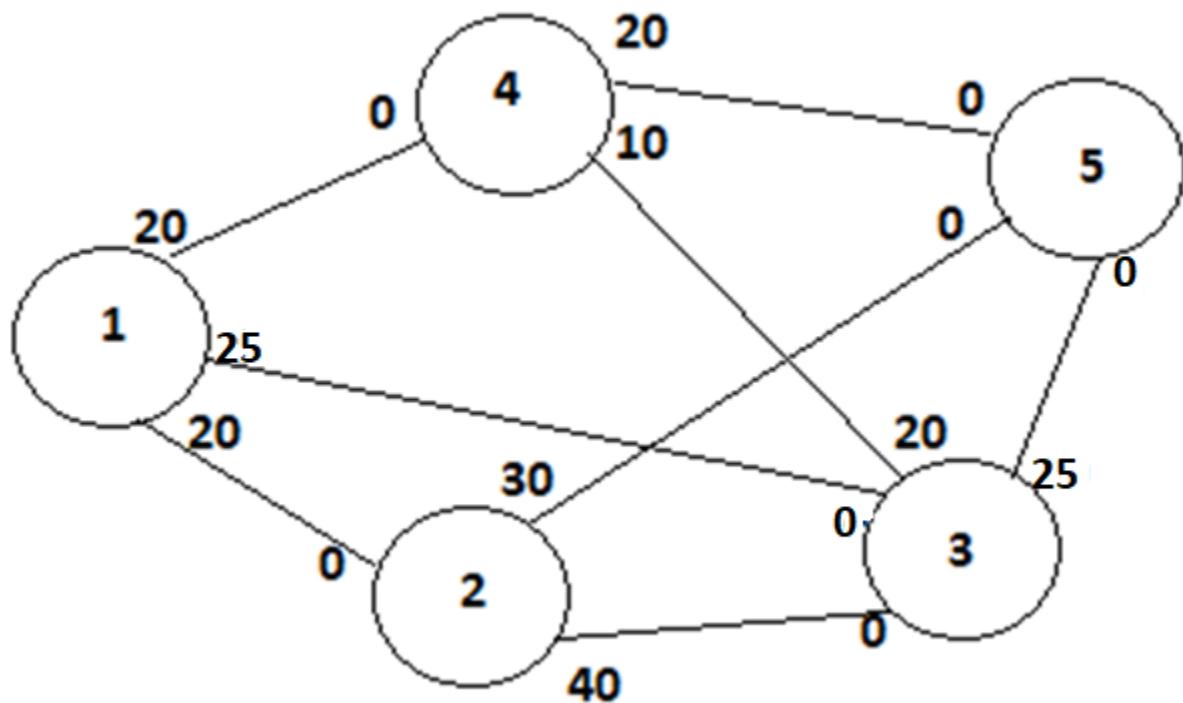
Estas capacidades van modificándose en las sucesivas etapas.

Las capacidades residuales son las capacidades restantes del arco una vez que pasa un flujo por él.

Se denominan c_{ij} y c_{ji} respectivamente.

Para un vértice j que recibe flujo de i se define una clasificación $[a_j, i]$, donde a_j es el flujo desde el vértice i al vértice j .

Ejemplo :



```
6 public class FordFulkerson {  
7     // Método para realizar una búsqueda en profundidad (DFS) para encontrar un camino aumentante  
8     private static boolean dfs(int[][][] residualGraph, int source, int sink, int[] parent) {  
9         int numVertices = residualGraph.length;  
10        boolean[] visited = new boolean[numVertices];  
11        Arrays.fill(visited, false);  
12  
13        // Utilizamos una pila para realizar el DFS  
14        Stack<Integer> stack = new Stack<>();  
15        stack.push(source);  
16        visited[source] = true;  
17  
18        while (!stack.isEmpty()) {  
19            int u = stack.pop();  
20  
21            // Recorremos todos los nodos adyacentes  
22            for (int v = 0; v < numVertices; v++) {  
23                // Si no está visitado y hay capacidad residual  
24                if (!visited[v] && residualGraph[u][v] > 0) {  
25                    parent[v] = u; // Guardamos el camino  
26                    if (v == sink) {  
27                        return true; // Llegamos al sumidero  
28                    }  
29                    stack.push(v);  
30                    visited[v] = true;  
31                }  
32            }  
33        }  
34        return false;  
35    }  
36}
```

```

37 // Implementación del algoritmo de Ford-Fulkerson
38 public static int fordFulkerson(int[][] graph, int source, int sink) {
39     int numVertices = graph.length;
40
41     // Crear el grafo residual
42     int[][] residualGraph = new int[numVertices][numVertices];
43     for (int u = 0; u < numVertices; u++) {
44         for (int v = 0; v < numVertices; v++) {
45             residualGraph[u][v] = graph[u][v];
46         }
47     }
48
49     // Array para guardar el camino aumentante
50     int[] parent = new int[numVertices];
51
52     int maxFlow = 0; // Inicializamos el flujo máximo en 0
53
54     // Mientras exista un camino aumentante
55     while (dfs(residualGraph, source, sink, parent)) {
56         // Determinar la capacidad mínima en el camino aumentante
57         int pathFlow = Integer.MAX_VALUE;
58         for (int v = sink; v != source; v = parent[v]) {
59             int u = parent[v];
60             pathFlow = Math.min(pathFlow, residualGraph[u][v]);
61         }
62
63         // Actualizar las capacidades residuales
64         for (int v = sink; v != source; v = parent[v]) {
65             int u = parent[v];
66             residualGraph[u][v] -= pathFlow; // Reducimos en la dirección original
67             residualGraph[v][u] += pathFlow; // Aumentamos en la dirección opuesta
68         }
69
70         // Añadir el flujo de este camino al flujo máximo
71         maxFlow += pathFlow;
72     }
73
74     return maxFlow;
75 }
76

```

```
77 public static void main(String[] args) {
78     // Ejemplo de grafo representado como matriz de adyacencia
79     int[][] graph = {
80         {0, 16, 13, 0, 0, 0},
81         {0, 0, 10, 12, 0, 0},
82         {0, 4, 0, 0, 14, 0},
83         {0, 0, 9, 0, 0, 20},
84         {0, 0, 0, 7, 0, 4},
85         {0, 0, 0, 0, 0, 0}
86     };
87
88     int source = 0; // Nodo fuente
89     int sink = 5;   // Nodo sumidero
90
91     System.out.println("El flujo máximo es: " + fordFulkerson(graph, source, sink));
92 }
93 }
94 }
```

Explicación del código

1. Representación del grafo:

- El grafo se representa como una matriz de adyacencia, donde $graph[u][v]$ indica la capacidad entre los nodos u y v .

2. Búsqueda de caminos aumentantes:

- Utilizamos una función DFS que encuentra un camino aumentante desde la fuente (*source*) al sumidero (*sink*).
- El arreglo `parent` almacena el camino encontrado.

3. Actualización del grafo residual:

- Se reduce la capacidad en las aristas originales y se incrementa en las aristas inversas para mantener el grafo residual.

4. Cálculo del flujo máximo:

- Sumamos el flujo de cada camino aumentante al flujo máximo total.

Entrada y salida

En el ejemplo proporcionado, la salida será:

yaml

 Copiar código

```
El flujo máximo es: 23
```

Esto corresponde al flujo máximo en el grafo de ejemplo. Si quieras, puedo explicarte en detalle alguna parte del código o ayudarte a adaptarlo a otro caso. 😊

Ejemplo

Primera etapa:

$$S_1 = \{2, 3, 4\}$$

$$S_3 = \{4, 5\}$$

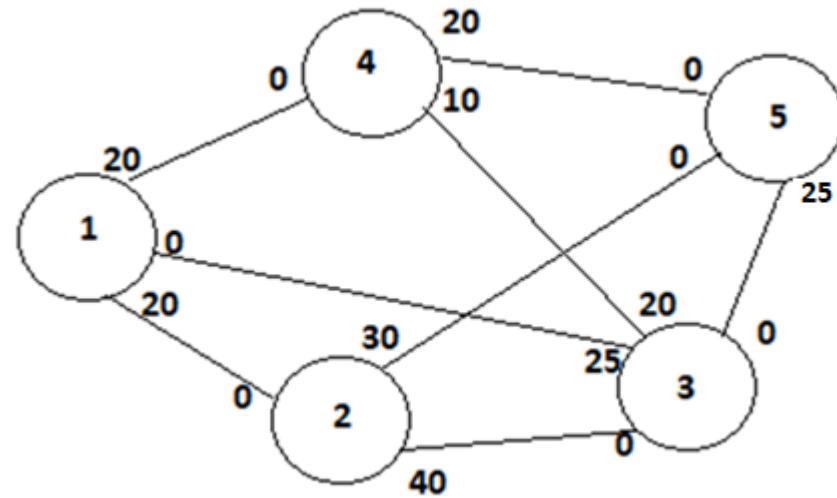
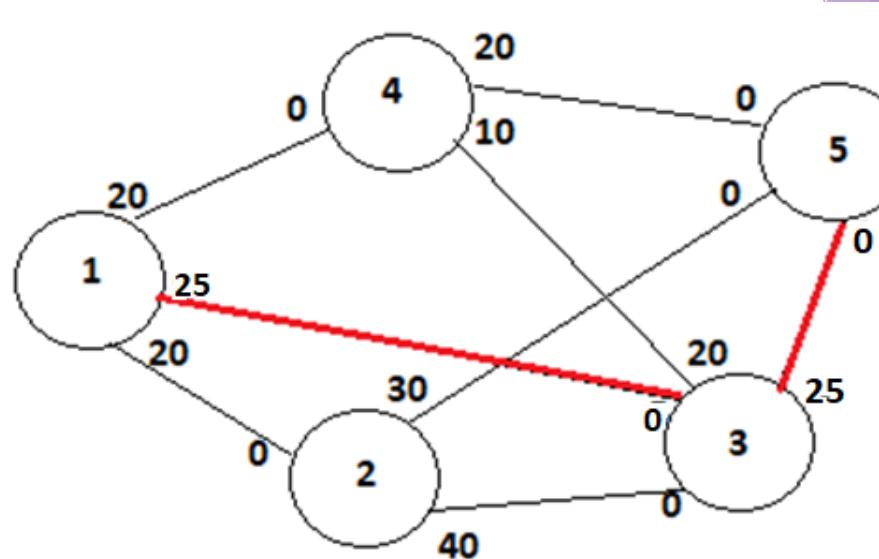
Ruta: 1 - 3 - 5

$$C_{13,31} = (25-25, 0+25) = (0, 25)$$

$$C_{35,53} = (25-25, 0+25) = (0, 25)$$

$$k = \text{mínimo } (\infty, 25, 25) = 25$$

$$f_1 = 25$$



Grafo con los valores de los vértices actualizados luego de analizada la ruta

Segunda etapa:

$$S_1 = \{2, 3, 4\}$$

$$S_2 = \{3, 5\} // 1 \text{ no puede estar}$$

$$S_3 = \{1, 4, 5\}$$

$$S_4 = \{3, 5\}$$

Ruta: 1 - 2 - 3 - 4 - 5

$$C_{12,21} = (20-20, 0+20) = (0, 20)$$

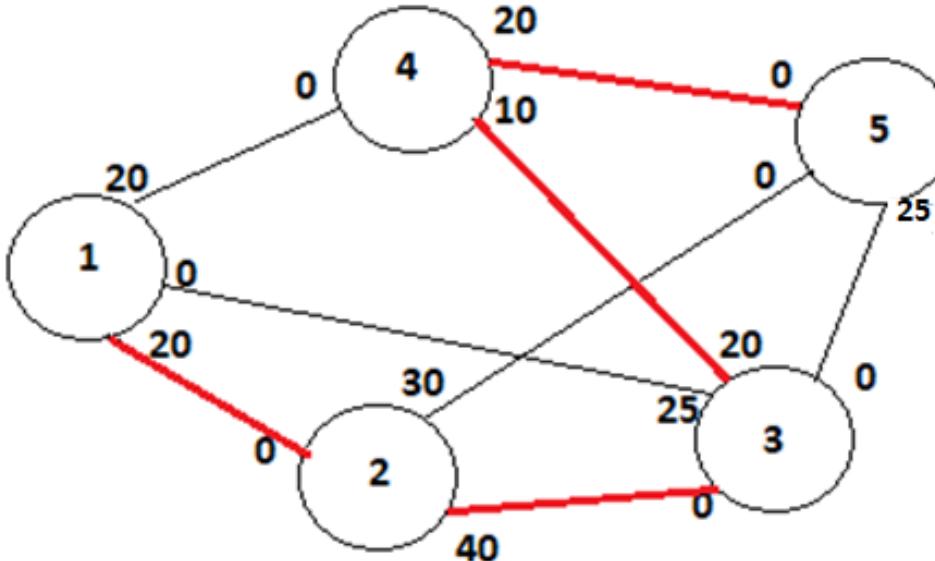
$$C_{23,32} = (40 - 20, 0 + 20) = (20, 20)$$

$$C_{34,43} = (20-20, 10+20) = (0, 30)$$

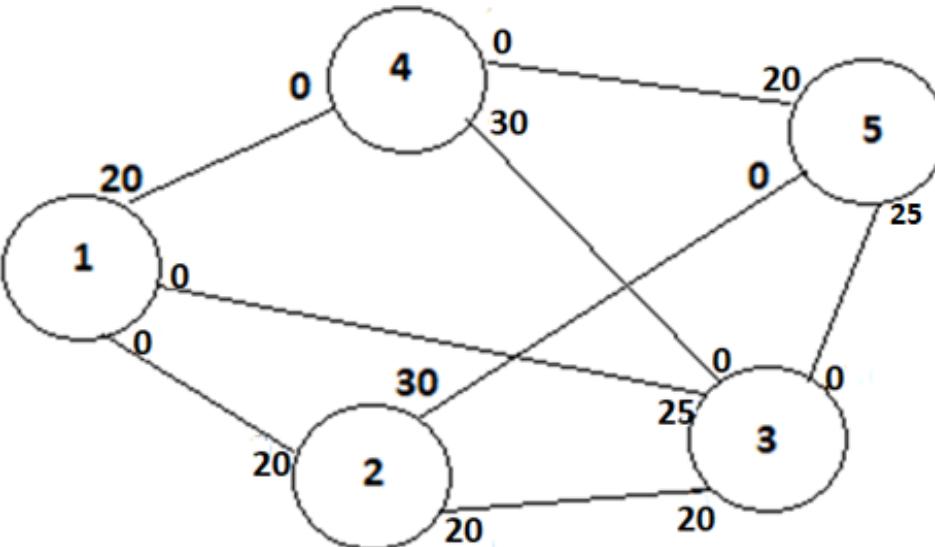
$$C_{45,54} = (20-20, 0+20) = (0, 20)$$

$$k = \min \{\infty, 20, 40, 20, 20\} = 20$$

$$f_2 = 20$$



Se ha indicado en rojo una ruta de penetración



Grafo con los valores de los vértices actualizados luego de analizada la ruta

Tercera etapa:

$$S1 = \{2, 3, 4\}$$

$$S2 = \{3, 5\}$$

$$S3 = \{2, 4, 5\} \quad // 1 \text{ no puede estar}$$

$$S4 = \{3, 5\}$$

Ruta: 1 - 4 - 3 - 2 - 5

$$C_{14,41} = (20-20, 0+20) = (0, 20)$$

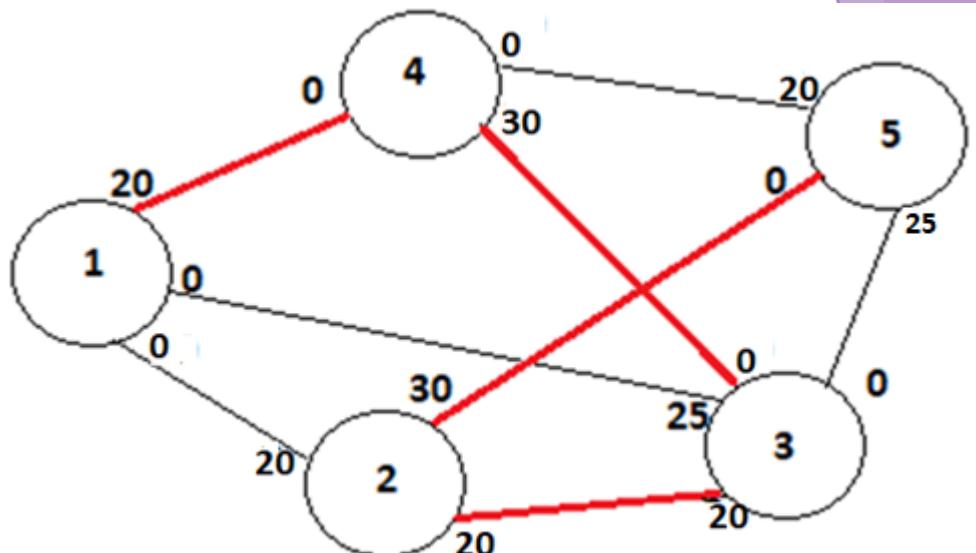
$$C_{43,34} = (30-20, 0+20) = (10, 20)$$

$$C_{32,23} = (20-20, 20+20) = (0, 40)$$

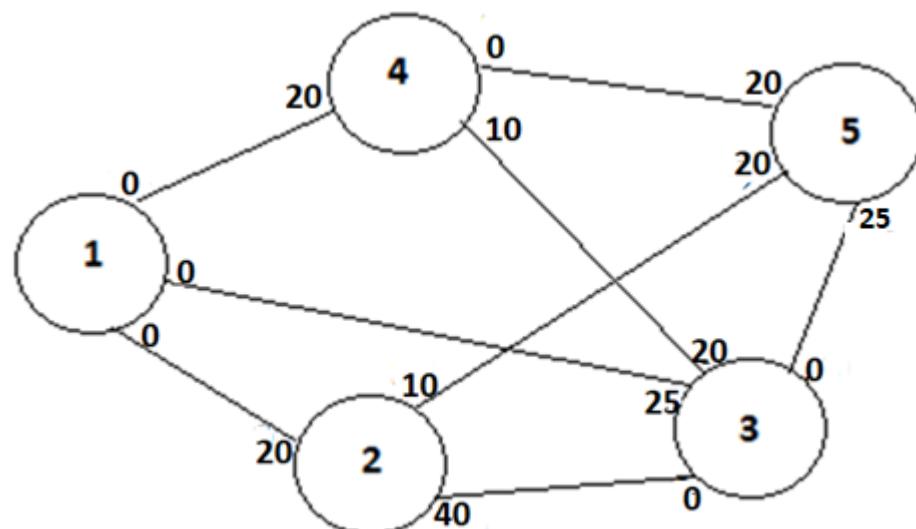
$$C_{23,32} = (30-20, 0+20) = (10, 20)$$

$$k = \text{mínimo } (\infty, 20, 30, 20, 30) = 20$$

$$f_3 = 20$$



Se ha indicado en rojo una ruta de penetración



Grafo con los valores de los vértices actualizados luego de analizada la ruta

$$\text{Flujo máximo} = f_1 + f_2 + f_3 = 25 + 20 + 20 = 65$$

Backtracking

La estrategia de **backtracking** (o retroceso) es un método general para encontrar soluciones a problemas que pueden ser expresados en términos de elección de decisiones secuenciales. Es una técnica que explora todas las posibles configuraciones de una solución de manera sistemática y eficiente, retrocediendo cuando se determina que una elección particular no conduce a una solución viable. Aquí te explico cómo funciona y en qué consiste:

Funcionamiento de la Estrategia de Backtracking

1. **Elección:** En cada paso, se elige una opción entre las disponibles.
2. **Exploración:** Se avanza con la opción elegida y se sigue tomando decisiones hasta que se llega a una solución completa.
3. **Validación:** Cada vez que se toma una decisión, se verifica si la decisión es válida. Si no lo es, se descarta.
4. **Retroceso:** Si una decisión lleva a un callejón sin salida (es decir, no conduce a una solución completa y válida), se retrocede al paso anterior y se intenta con la siguiente opción disponible.

Características del Backtracking

- **Recursivo:** El backtracking generalmente se implementa mediante una función recursiva que prueba todas las posibilidades.
- **Exploración en profundidad:** Explora cada rama de decisiones completamente antes de retroceder y probar la siguiente.
- **Pruning (Poda):** Puede incluir técnicas para descartar ramas de decisiones que no pueden llevar a una solución válida, lo que mejora la eficiencia.

Aplicaciones Comunes

- **Resolución de laberintos:** Encontrar el camino correcto en un laberinto.
- **Problemas de asignación:** Como el problema de las N reinas, donde se debe colocar N reinas en un tablero de ajedrez sin que se ataquen entre sí.
- **Sudoku:** Completar el tablero respetando las reglas del juego.
- **Problemas de combinación:** Como encontrar todas las combinaciones posibles de un conjunto dado.

Estrategia “Vuelta atrás”

Conceptos básicos

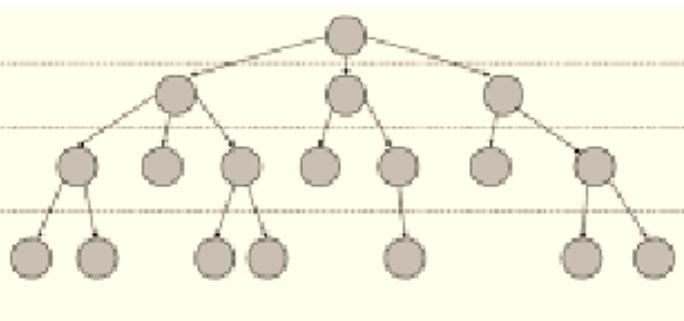
Fuerza Bruta Vs. Vuelta Atrás

La estrategia de Backtracking o Vuelta Atrás proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.

Se asemeja a un recorrido en profundidad dentro de un árbol conceptual cuya existencia sólo es implícita, y que denominaremos árbol de expansión.

Cada nodo de nivel k representa una parte de la solución y está formado por k etapas que se suponen ya realizadas. Los nodos hijo son los siguientes posibles pasos en la etapa siguiente.

Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.



Esta técnica se utiliza para resolver problemas en los que la solución es compuesta y donde cada solución es el resultado de una secuencia de decisiones.

Los problemas a resolver por vuelta atrás son los siguientes:

- Problemas de decisión: Buscan la solución o soluciones que satisfacen ciertas restricciones.
- Problemas de optimización: Buscan la solución óptima en base a una función objetivo.

Conceptos básicos

Al desarrollar el método, puede darse una de dos situaciones:

- ▶ Que se tenga éxito y se llega a una solución (una hoja del árbol).
 - ▶ Si buscábamos una solución al problema => el algoritmo finaliza acá.
 - ▶ Si buscábamos todas las soluciones o la mejor de entre todas ellas => el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.
- ▶ Si el recorrido no tiene éxito (=si en alguna etapa la solución parcial construida hasta el momento no se puede completar) se llegó a un nodo fracaso.
=> el algoritmo vuelve atrás en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo. Si existe uno o más caminos aún no explorados que puedan conducir a solución, el recorrido del árbol continúa por ellos

¿Cómo evitar el crecimiento exponencial?

Considerar el menor conjunto de nodos que puedan llegar a ser soluciones => hacer una poda

Ejemplos

Un problema puede resolverse con un algoritmo Vuelta Atrás cuando

la solución se puede expresar como una n-tupla $[x_1, x_2, \dots, x_n]$ en la cual cada una de las componentes x_i del este vector es elegida en cada etapa de entre un conjunto finito de valores. Cada etapa representará un nivel en el árbol de expansión

Algunos ejemplos:

- ▶ El problema de las n reinas
- ▶ El problema de los saltos del caballo de ajedrez
- ▶ El problema de la mochila (0,1)
- ▶ El recorrido del rey de ajedrez
- ▶ La salida del laberinto
- ▶ El problema de las parejas estables
- ▶ El armado de horarios docentes
- ▶ Los subconjuntos de suma dada
- ▶ El problema del viajante de comercio (ciclo hamiltoniano)
- ▶ El problema de los horarios de los trenes
- ▶ El armado de un sudoku
- ▶ El armado de cuadrados mágicos

Esquema general para la búsqueda de una solución

```
PROCEDURE VueltaAtras(etapa);
BEGIN
    IniciarOpciones;
    REPEAT
        SeleccionarNuevaOpcion;
        IF Aceptable THEN
            AnotarOpcion;
            IF SolucionIncompleta THEN
                VueltaAtras(etapa_siguiente);
                IF NOT exito THEN
                    CancelarAnotacion
                END
            ELSE (* solucion completa *)
                exito:=TRUE
            END
        END
    UNTIL (exito) OR (UltimaOpcion)
END VueltaAtras;
```

Esquema general para la obtención de todas las soluciones

```
PROCEDURE VueltaAtrasTodasSoluciones(etapa);  
BEGIN  
    IniciarOpciones;  
    REPEAT  
        SeleccionarNuevaOpcion;  
        IF Aceptable THEN  
            AnotarOpcion;  
            IF SolucionIncompleta THEN  
                VueltaAtrasTodasSoluciones(etapa_siguiente);  
            ELSE  
                ComunicarSolucion  
            END;  
            CancelarAnotacion  
        END  
    UNTIL (UltimaOpcion);  
END VueltaAtrasTodasSoluciones;
```

Tomado de “Diseño de Algoritmos”, de Guerequeta y Vallecillo

Problema de las “cuatro” reinas

Planteamos el problema de las 8 reinas pero de forma reducida: ubicar en un tablero de 4 por 4, 4 reinas de modo que no se amenacen entre ellas. No puede haber dos reinas en la misma fila, columna o diagonal.

Supongamos que en cada etapa resolvemos una fila. Para cada fila, hay 4 posiciones (columnas) posibles: C0, C1, C2 y C3.

Entonces, inicialmente:

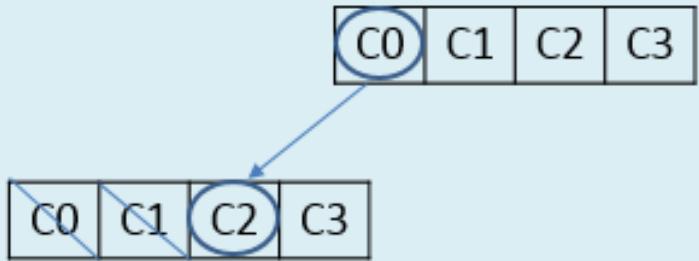
C0	C1	C2	C3
----	----	----	----

Elegimos C0, y es:

O			

Ahora pasamos a la fila 1. ‘No se verifican las restricciones para C0 ni para C1, entonces

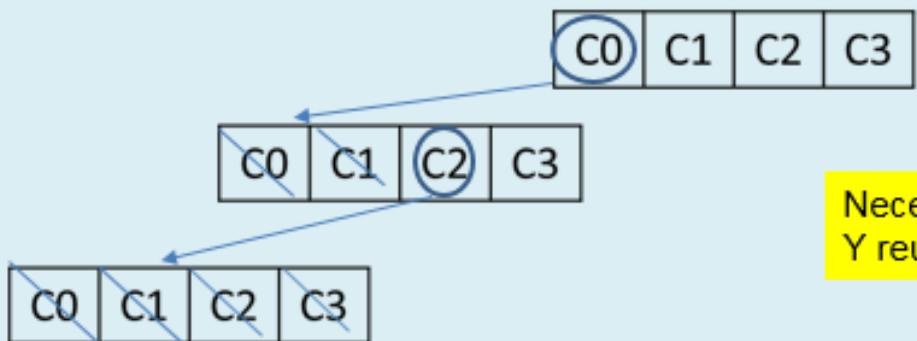
Elegimos C2



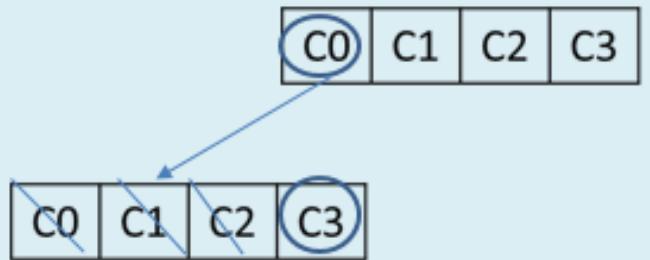
Y queda

o			
			o

Para la fila 3, no podemos elegir C0 ni C1 ni C2 ni C3, es decir:



Necesitamos hacer una "vuelta atrás",
Y reubicar la segunda reina



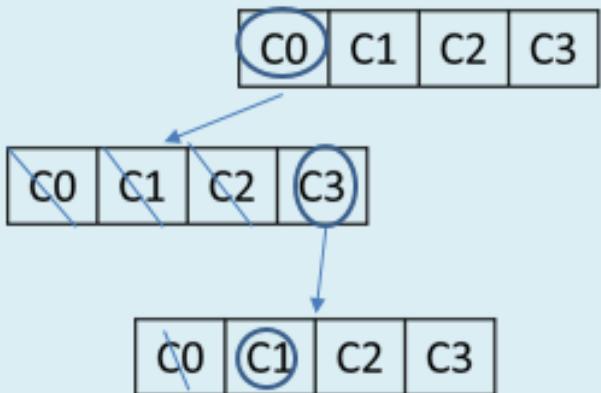
Y nos queda

o			

Para la tercera fila, C0 no sirve, pero sí C1. La elegimos y queda:

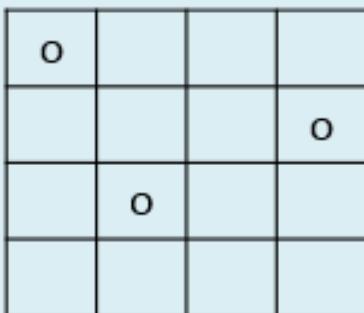
o			
	o		

El árbol de acuerdo a la última elección será:



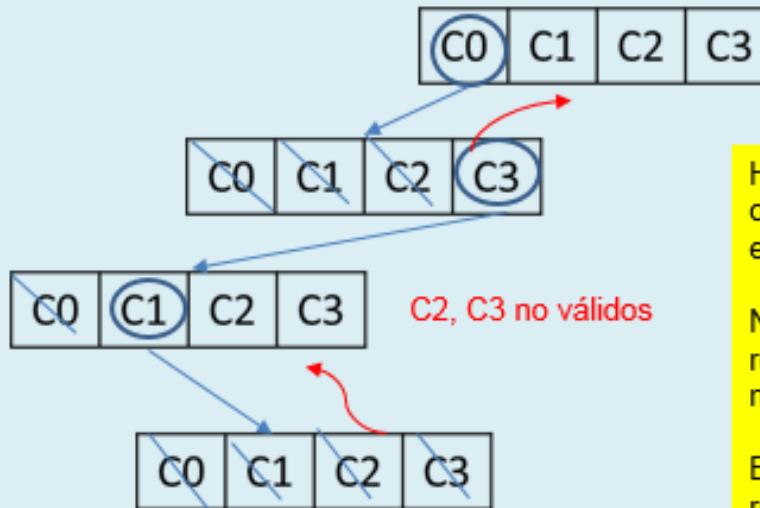
Y ahora hay que ubicar la reina de la cuarta fila

Pero observamos que, dada esta situación,



La cuarta reina no puede ser ubicada ni en C0 ni en C1, ni en C2 ni en C3

Es decir que, planteando el árbol de decisiones,

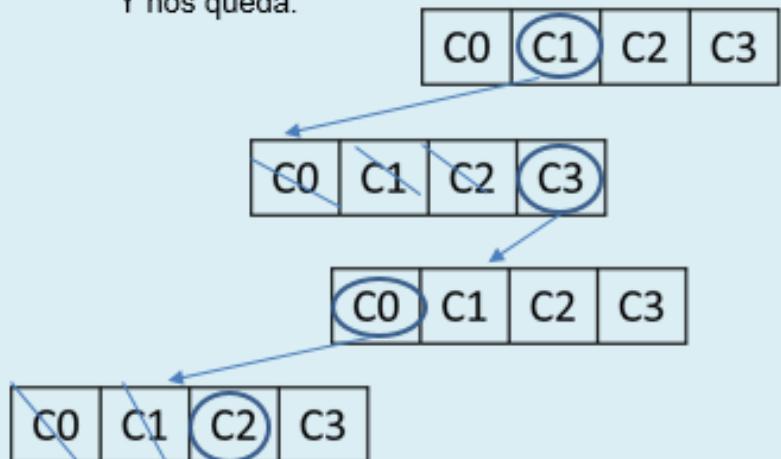


Hay que hacer una "vuelta atrás", pero observamos que en la fila 3, no se puede ubicar en la columna C2 ni en la C3.

Necesitaremos otra "vuelta atrás" y reubicar la reina de la segunda fila, pero, como vemos, ya no hay ahí más columnas disponibles.

Entonces realizaremos otra "vuelta atrás" para retroceder hasta la fila 1 y elegir

Y nos queda:



		O	
			O
O			
			O

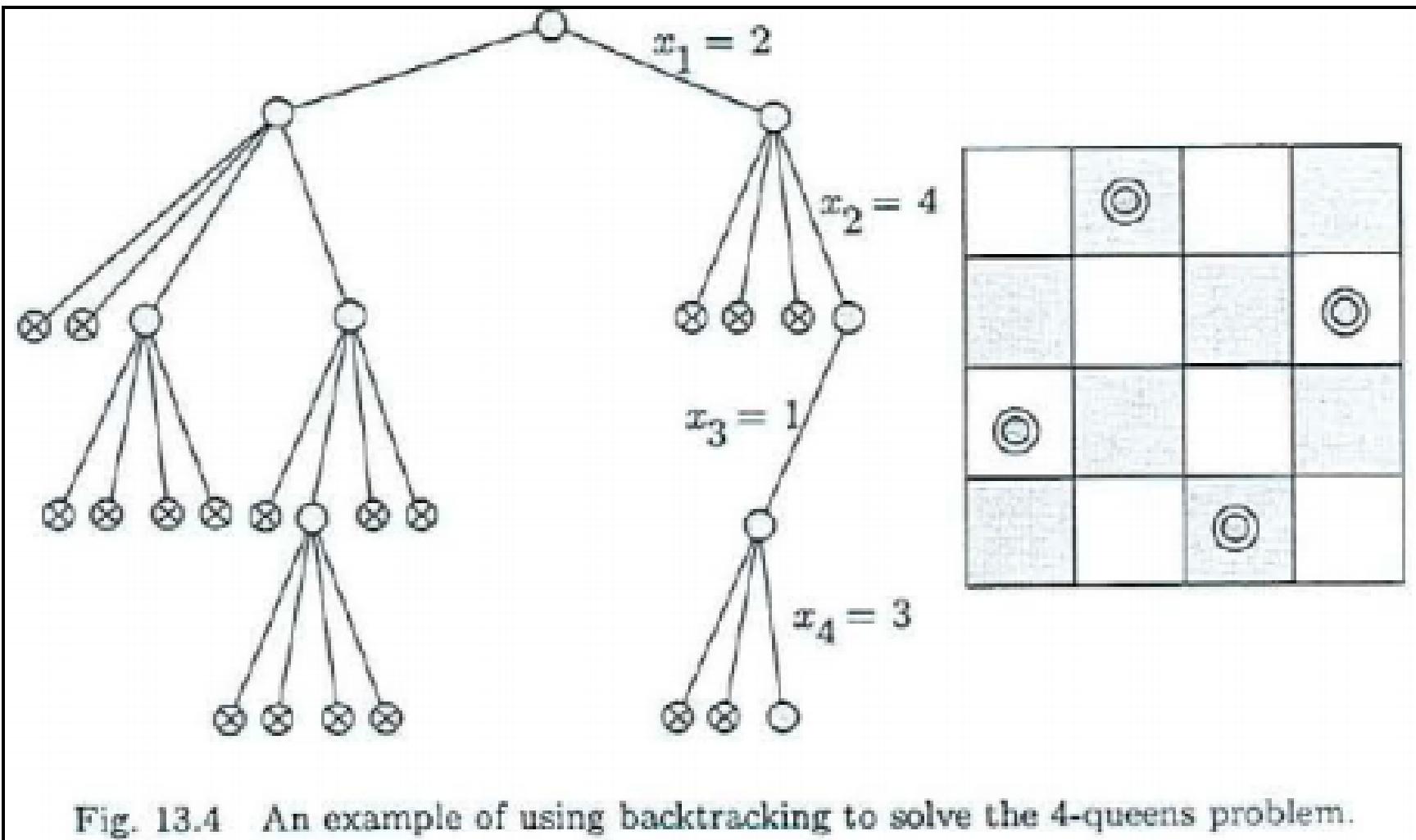


Ilustración 2, árbol de búsqueda generado y tablero solución (4-reinas)

Tomado de Esteban Sánchez-Velázquez Iturbide
 "Revisión Bibliográfica de Problemas Resolubles por la Técnica de Vuelta Atrás"
 Número 2012-04

```
3 public class CuatroReinas {
4
5     // Método principal
6     public static void main(String[] args) {
7         int n = 4; // Número de reinas y tamaño del tablero
8         int[][] tablero = new int[n][n]; // Inicializamos un tablero vacío
9
10    if (resolver4Reinas(tablero, 0)) {
11        imprimirTablero(tablero);
12    } else {
13        System.out.println("No hay solución.");
14    }
15}
16
```

```
// Método para resolver el problema de 4 reinas
public static boolean resolver4Reinas(int[][][] tablero, int fila) {
    int n = tablero.length;

    // Caso base: todas las reinas han sido colocadas
    if (fila >= n) {
        return true;
    }

    // Intentar colocar una reina en cada columna de la fila actual
    for (int col = 0; col < n; col++) {
        if (esSeguro(tablero, fila, col)) {
            // Colocar la reina
            tablero[fila][col] = 1;

            // Recursivamente tratar de colocar el resto de las reinas
            if (resolver4Reinas(tablero, fila + 1)) {
                return true;
            }

            // Si no es posible, retrocedemos (backtracking)
            tablero[fila][col] = 0;
        }
    }

    // Si no se puede colocar una reina en esta fila, retornamos false
    return false;
}
```

```
45
46 // Método para verificar si es seguro colocar una reina en tablero[fila][col]
47 public static boolean esSeguro(int[][][] tablero, int fila, int col) {
48     int n = tablero.length;
49
50     // Comprobar la misma columna en filas anteriores
51     for (int i = 0; i < fila; i++) {
52         if (tablero[i][col] == 1) {
53             return false;
54         }
55     }
56
57     // Comprobar la diagonal superior izquierda
58     for (int i = fila, j = col; i >= 0 && j >= 0; i--, j--) {
59         if (tablero[i][j] == 1) {
60             return false;
61         }
62     }
63
64     // Comprobar la diagonal superior derecha
65     for (int i = fila, j = col; i >= 0 && j < n; i--, j++) {
66         if (tablero[i][j] == 1) {
67             return false;
68         }
69     }
70
71     // Si pasa todas las comprobaciones, es seguro colocar la reina
72     return true;
73 }
74
75 // Método para imprimir el tablero
76 public static void imprimirTablero(int[][][] tablero) {
77     int n = tablero.length;
78     for (int i = 0; i < n; i++) {
79         for (int j = 0; j < n; j++) {
80             System.out.print((tablero[i][j] == 1 ? "Q " : ". "));
81         }
82         System.out.println();
83     }
84 }
85 }
```

Explicación del código

1. Backtracking:

- Intentamos colocar una reina en cada fila de manera recursiva.
- Si encontramos una posición segura, colocamos la reina y avanzamos a la siguiente fila.
- Si no se puede colocar una reina en ninguna columna de la fila actual, retrocedemos y probamos con otra posición en la fila anterior.

2. Función `esSeguro`:

- Comprueba si una posición es válida asegurando que:
 - No haya otra reina en la misma columna.
 - No haya otra reina en la diagonal superior izquierda.
 - No haya otra reina en la diagonal superior derecha.

3. Caso base:

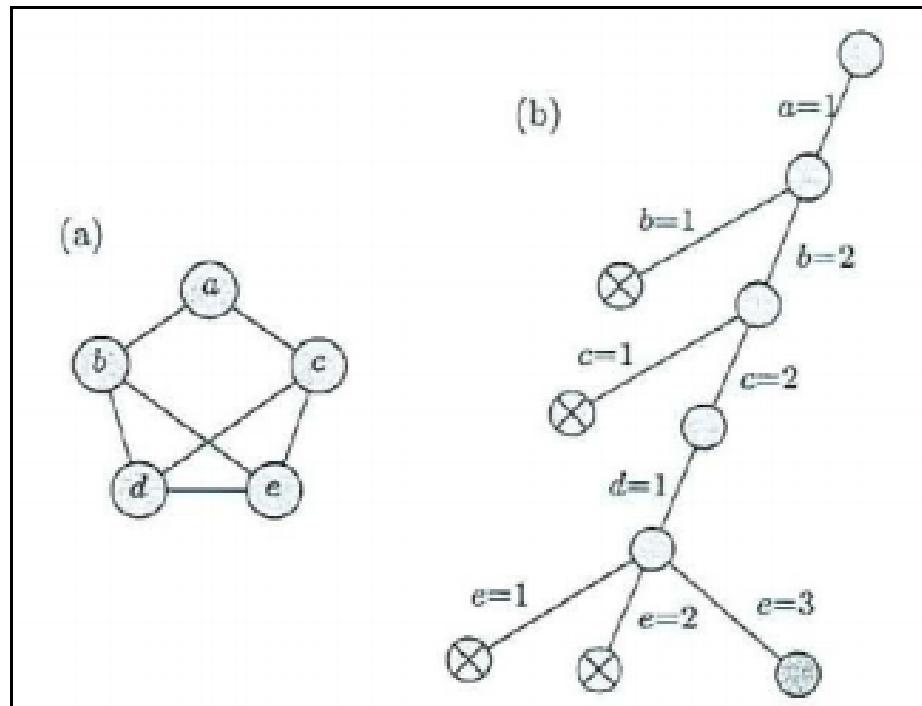
- Cuando todas las reinas están colocadas correctamente ($fila \geq n$), devolvemos `true`.

4. Impresión del tablero:

- Imprimimos el tablero como una cuadrícula donde `Q` representa una reina y `.` representa una celda vacía.

Coloreado de mapas o grafos usando estrategia de Backtracking:

Dado un grafo conexo y un número $m > 0$, llamamos colorear el grafo a asignar un número i ($1 \leq i \leq m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales.



Ejemplo de desarrollo para tres colores

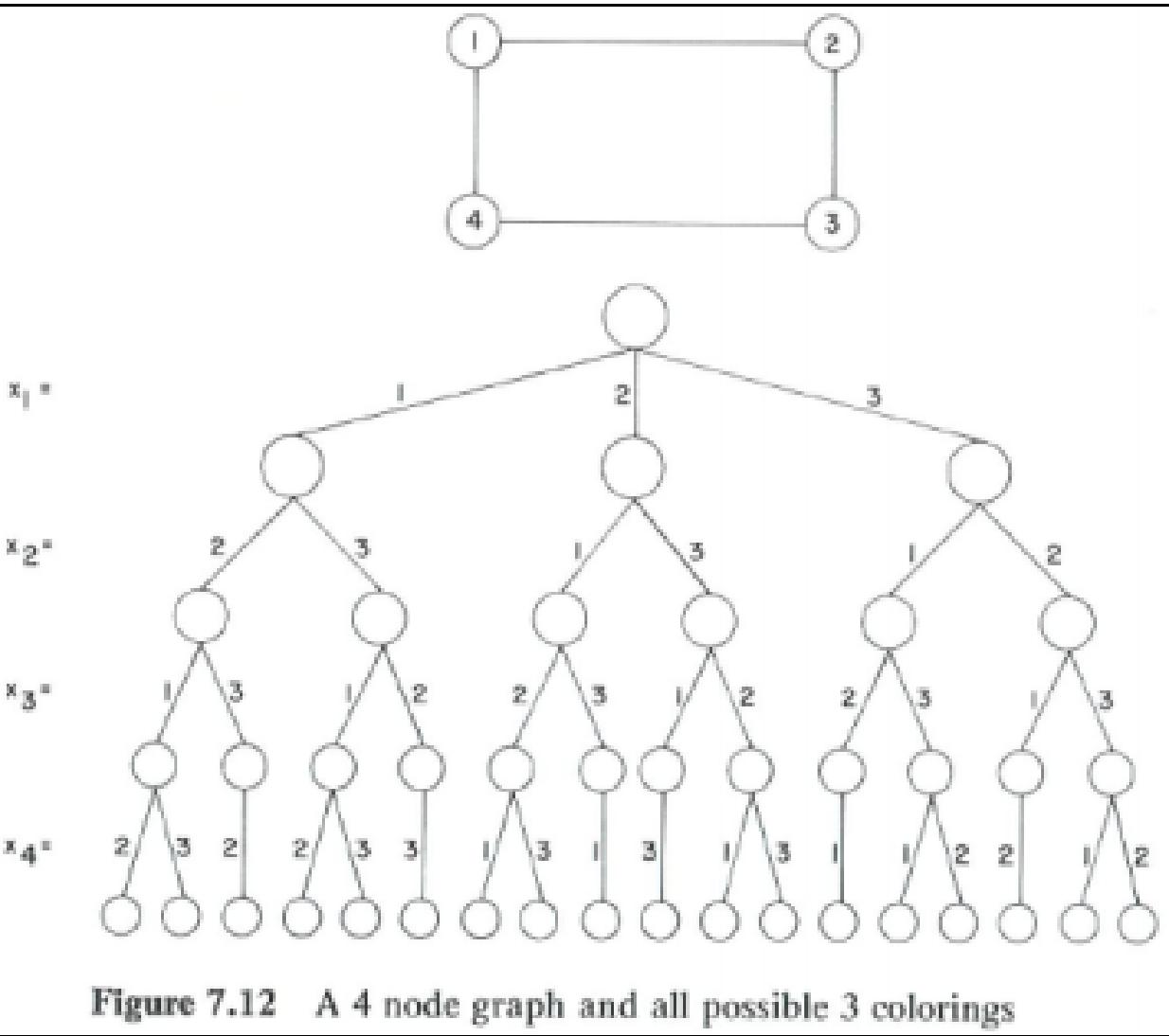


Figure 7.12 A 4 node graph and all possible 3 colorings

Ilustración 16, figura compuesta (grafo y árbol potencial, 3-coloring)

Tomado de Esteban Sánchez-Velázquez Iturbide

"Revisión Bibliográfica de Problemas Resolubles por la Técnica de Vuelta Atrás"

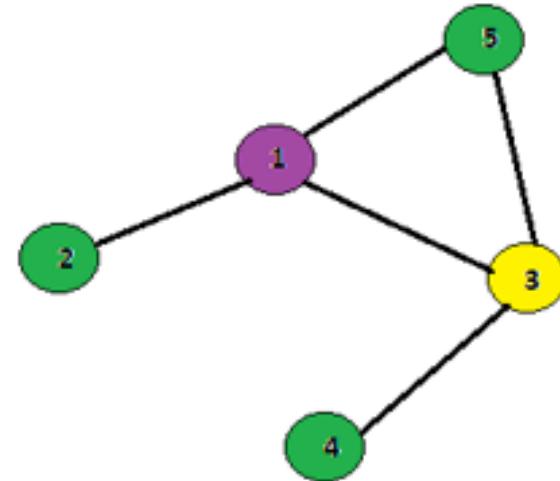
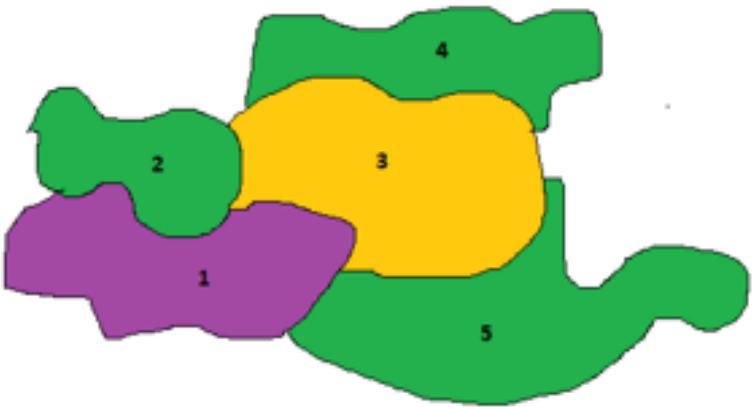
Número 2012-04

Consideremos este problema:

Dado un grafo no dirigido G , queremos saber cuántos colores necesitaremos para que los vértices adyacentes tengan colores diferentes.

Este valor se conoce como número cromático del grafo.

El problema se corresponde con el de la determinación del número mínimo de colores para pintar un mapa de modo que las zonas con fronteras en común tengan distinto color.



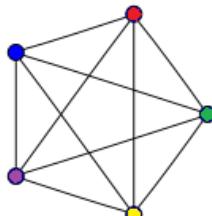
Observar que el grafo obtenido a partir de un mapa (que ha sido dibujado en un plano) siempre es un grafo plano, es decir que no hay cruce entre aristas.

Algunos grafos especiales tienen números cromáticos conocidos:

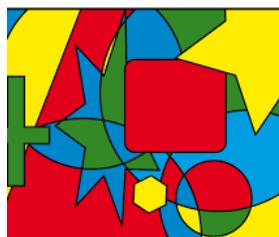
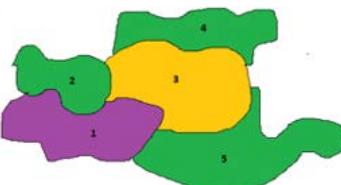
1. Un grafo con dos vértices y una arista tiene número cromático 2.



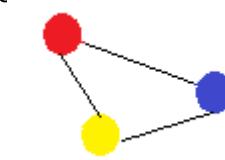
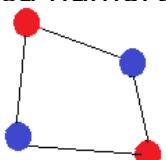
2. Un grafo no dirigido conexo de n nodos tiene número cromático n .



4. Si un grafo es plano, entonces su número cromático es menor o igual que 4 (teorema de los cuatro colores).



5. Si un grafo es un ciclo su número cromático es 2 si n es par y tres si n es impar

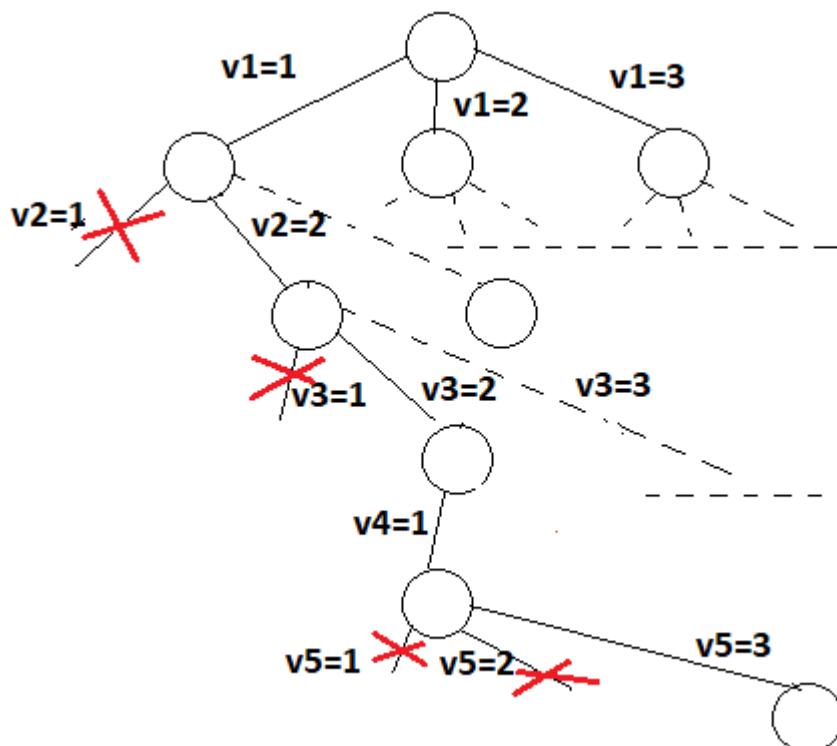


Utilizando la estrategia de backtracking podemos **probar** si es posible o no colorear un grafo con un número determinado de colores, o bien determinar el número cromático del mismo

Ejemplo 1: con este grafo, hacemos el desarrollo del árbol correspondiente y analizamos los resultados:



En estos gráfico se muestra un posible colooreo. Desarrollando por backtracking en el diagrama siguiente, y tomando la primera solución que aparece (ya que no se ha diagramado el árbol completo de posibilidades) obtenemos otra posible coloración.

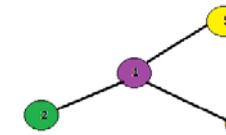


Restricciones según las fronteras:

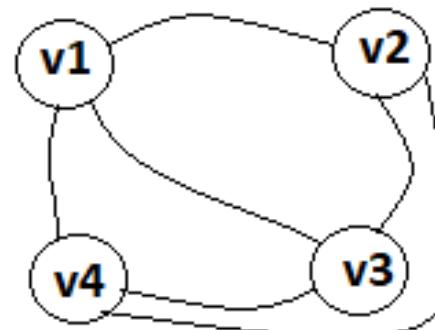
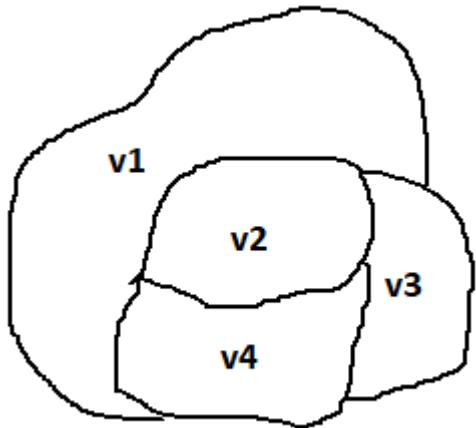
$$\begin{aligned} v1 &\neq v2 \\ v1 &\neq v5 \\ v1 &\neq v3 \\ v3 &\neq v4 \\ v3 &\neq v5 \end{aligned}$$

Primera solución obtenida (obtener las otras!)

$$\begin{aligned} v1 &= 1 \\ v2 &= 2 \\ v3 &= 2 \\ v4 &= 1 \\ v5 &= 3 \end{aligned}$$



Ahora consideremos el siguiente mapa y su grafo, y analicemos, usando backtracking, si es posible colorearlo con tres colores:



Restricciones:

$$v1 \neq v2$$

$$v1 \neq v3$$

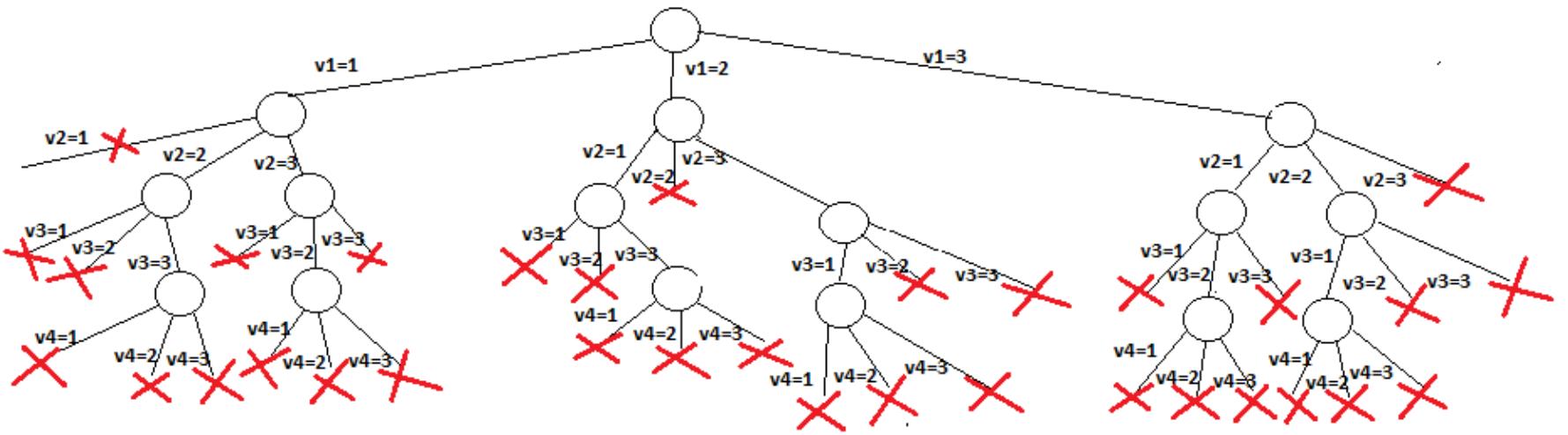
$$v1 \neq v4$$

$$v2 \neq v3$$

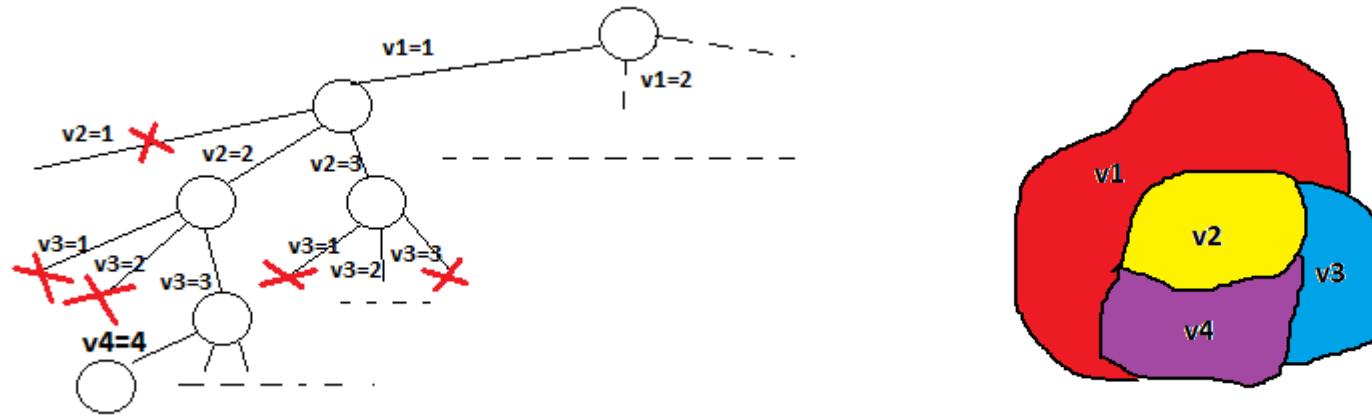
$$v2 \neq v4$$

$$v3 \neq v4$$

Como se observa en el desarrollo siguiente, no es posible colorear el mapa mostrado antes con sólo tres colores



Pero, como se ve en este gráfico, sí es posible encontrar combinaciones que permitan el coloreo con cuatro colores (sólo se ha representado parte del árbol de búsqueda)



Determinación del número cromático con backtracking

Dado un grafo conexo y un número $m > 0$, llamamos *colorear* el grafo a asignar un número i ($1 \leq i \leq m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales. Deseamos implementar un algoritmo que coloree un grafo dado.

Solución



El nombre de este problema proviene de un problema clásico, el del coloreado de mapas en el plano. Para resolverlo se utilizan grafos puesto que un mapa puede ser representado por un grafo conexo. Cada vértice corresponde a un país y cada arco entre dos vértices indica que los dos países son vecinos. Desde el siglo XVII ya se conoce que con cuatro colores basta para colorear cualquier mapa planar, pero sin embargo existen situaciones en donde no nos importa el número de colores que se utilicen.

Para implementar un algoritmo de Vuelta Atrás, la solución al problema puede expresarse como una n -tupla de valores $X = [x_1, x_2, \dots, x_n]$ donde x_i representa el color del i -ésimo vértice. El algoritmo que resuelve el problema trabajará por etapas, asignando en cada etapa k un color (entre 1 y m) al vértice k -ésimo.

En primer lugar, y para un grafo con n vértices y con m colores, el algoritmo que encuentra una solución al problema es el siguiente:

Tomado de Guerequeta y Vallecillo,
“Técnicas de Diseño de algoritmos”

```
3 public class ColoreadoGrafos {
4
5     // Método principal
6     public static void main(String[] args) {
7         // Ejemplo de grafo: Matriz de adyacencia
8         int[][] grafo = {
9             {0, 1, 1, 1}, // Nodo 0 está conectado con 1, 2 y 3
10            {1, 0, 1, 0}, // Nodo 1 está conectado con 0 y 2
11            {1, 1, 0, 1}, // Nodo 2 está conectado con 0, 1 y 3
12            {1, 0, 1, 0} // Nodo 3 está conectado con 0 y 2
13        };
14
15        int numColores = 3; // Máximo número de colores
16        int[] colores = new int[grafo.length]; // Array para almacenar los colores asignados
17
18        if (resolverColoreado(grafo, numColores, colores, 0)) {
19            imprimirColores(colores);
20        } else {
21            System.out.println("No es posible colorear el grafo con " + numColores + " colores.");
22        }
23    }
24}
```

```
// Método para resolver el problema de coloreado de grafos usando backtracking
public static boolean resolverColoreado(int[][] grafo, int numColores, int[] colores, int nodo) {
    // Caso base: si todos los nodos están coloreados, retornamos true
    if (nodo == grafo.length) {
        return true;
    }

    // Intentamos asignar un color a este nodo
    for (int color = 1; color <= numColores; color++) {
        if (esSeguro(grafo, colores, nodo, color)) {
            // Asignamos el color
            colores[nodo] = color;

            // Recursivamente tratamos de colorear el resto de los nodos
            if (resolverColoreado(grafo, numColores, colores, nodo + 1)) {
                return true;
            }
        }

        // Si no es posible, desasignamos el color (backtracking)
        colores[nodo] = 0;
    }
}

// Si no podemos asignar ningún color a este nodo, retornamos false
return false;
}
```

```
/*
52 // Método para verificar si es seguro asignar un color a un nodo
53 public static boolean esSeguro(int[][] grafo, int[] colores, int nodo, int color) {
54     for (int i = 0; i < grafo.length; i++) {
55         // Si el nodo es adyacente y tiene el mismo color, no es seguro
56         if (grafo[nodo][i] == 1 && colores[i] == color) {
57             return false;
58         }
59     }
60     return true;
61 }
62
63 // Método para imprimir la asignación de colores
64 public static void imprimirColores(int[] colores) {
65     System.out.println("Colores asignados a los nodos:");
66     for (int i = 0; i < colores.length; i++) {
67         System.out.println("Nodo " + i + " -> Color " + colores[i]);
68     }
69 }
70 }
```

Explicación del código

1. Representación del grafo:

- El grafo está representado mediante una **matriz de adyacencia**.
- $grafo[u][v] = 1$ significa que el nodo u está conectado con el nodo v .

2. Backtracking:

- El algoritmo intenta asignar colores a los nodos uno por uno.
- Si no es posible asignar un color válido a un nodo, retrocede (backtracking) y prueba con otro color para el nodo anterior.

3. Función `esSeguro`:

- Comprueba si el color actual puede ser asignado al nodo verificando que ningún nodo adyacente tenga el mismo color.

4. Caso base:

- Cuando todos los nodos han sido coloreados ($nodo == grafo.length$), el algoritmo retorna `true`.

5. Asignación de colores:

- Los colores son representados por números enteros (1, 2, ...).

Backtracking y Ciclos Hamiltonianos

Recordemos que dado un grafo conexo, se llama Ciclo Hamiltoniano a aquel ciclo que visita exactamente una vez cada vértice del grafo y vuelve al punto de partida.

El problema consiste en detectar la presencia de ciclos Hamiltonianos en un grafo dado.

```
procedure HAMILTONIAN(k)
    //This procedure uses the recursive formulation of backtracking//
    //to find all the Hamiltonian cycles of a graph. The graph//
    //is stored as a boolean adjacency matrix in GRAPH(1:n, 1:n). All//
    //cycles begin at vertex 1.//
    global integer X(1:n)
    local integer k, n
    loop //generate values for X(k)//
        call NEXTVALUE(k) //assign a legal next vertex to X(k)//
        if X(k) = 0 then return endif
        if k = n
            then print (X, '1') //a cycle is printed//
            else call HAMILTONIAN(k + 1)
        endif
        repeat
    end HAMILTONIAN
```

Tomado de Guerequeta y Vallecillo,
“Técnicas de Diseño de algoritmos”

Un vendedor ambulante de alfombras persas tiene que recorrer n ciudades volviendo tras ello al punto de partida.

El vendedor conoce todas las posibles conexiones directas por tren entre las ciudades (también conoce las tarifas correspondientes).

El vendedor desea conocer:

Todos los circuitos en tren que recorren cada ciudad exactamente una vez y regresen a la ciudad de partida

Suponiendo que los vértices del grafo están numerados desde 1 hasta n , la solución al problema puede expresarse como una n -tuple de valores $X = [x_1, x_2, \dots, x_n]$, donde x_i representa el i -ésimo vértice del ciclo Hamiltoniano.

El algoritmo que resuelve el problema trabajará por etapas, decidiendo en cada etapa qué vértice del grafo de los aún no considerados puede formar parte del ciclo.

Así, el algoritmo que resuelve el problema puede ser implementado como sigue:

```

CONST n = ...; (* numero de vertices *)
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;
    GRAFO = ARRAY[1..n],[1..n] OF BOOLEAN;
VAR g:GRAFO; X:SOLUCION; existe:BOOLEAN;
PROCEDURE Hamiltonianol(k:CARDINAL; VAR existe:BOOLEAN);
(* comprueba si existe un ciclo Hamiltoniano *)
BEGIN
    LOOP
        NuevoVertice(k);
        IF X[k] = 0 THEN EXIT END;
        IF k = n THEN existe:=TRUE
        ELSE Hamiltonianol(k+1,existe)
        END
    END
END Hamiltonianol;

```

Siendo el procedimiento NuevoVertice el que busca el siguiente vértice libre que pueda formar parte del ciclo y lo almacena en el vector solución X:

```

PROCEDURE NuevoVertice(k: CARDINAL);
    VAR j:CARDINAL; s:BOOLEAN;
BEGIN
    LOOP
        X[k]:=(X[k]+1) MOD (n+1);
        IF X[k]=0 THEN RETURN END;
        IF g[X[k-1],X[k]] THEN
            j:=1; s:=TRUE;
            WHILE (j<=k-1) AND s DO
                s:=(X[j]<>X[k]); INC(j)
            END;
            IF (j=k)AND((k<n)OR((k=n)AND(g[X[n],1)))) THEN RETURN END
        END
    END
END NuevoVertice;

```

Tomado de Guerequeta y Vallecello,
“Técnicas de Diseño de algoritmos”

El algoritmo termina cuando encuentra un ciclo o bien cuando ha analizado todas las posibilidades y no encuentra solución. Y respecto al estado de las variables y los parámetros de su invocación inicial, debe realizarse como sigue:

```
...
existe:=FALSE;
X[1]:=1;
Hamiltoniano1(2,existe);
IF existe THEN ComunicarSolucion(X) ELSE ...
...
```

Nos podemos plantear también una modificación al algoritmo para que encuentre todos los ciclos Hamiltonianos si es que hubiera más de uno. La modificación en este caso es simple

```
PROCEDURE Hamiltoniano2(k:CARDINAL);
(* determina todos los ciclos *)
BEGIN

LOOP
NuevoVertice(k);
IF X[k]=0 THEN RETURN END
IF k=n THEN ComunicarSolucion(X)
ELSE Hamiltoniano2(k+1)
END
END
END Hamiltoniano2;
```

Tomado de Guerequeta y Vallecillo,
“Técnicas de Diseño de algoritmos”

El problema de la salida del laberinto

Con una matriz bidimensional nxn puede representar un laberinto cuadrado.

En ella, cada posición contiene un entero no negativo; si es 0 indica que la casilla es transitable , o ∞ si no lo es .

Se ingresa al laberinto por la posición [1][1] . Se sale por [n][n] .

Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida.

Una solución aplicando backtracking consiste en avanzar por el laberinto en cada etapa; cada nodo representará el camino recorrido hasta el momento. En el esquema siguiente, con una variable global matriz se representa el laberinto y se registran los movimientos realizados, indicando en cada casilla el orden en el que ésta ha sido visitada.

Al llevar a cabo una vuelta atrás se marcan nuevamente con 0 las casillas correspondientes..

```
TYPE LABERINTO = ARRAY[1..n],[1..n] OF CARDINAL;
VAR lab:LABERINTO;

PROCEDURE Laberinto(k:CARDINAL;VAR fil,col:INTEGER;
                     VAR exito:BOOLEAN);
  VAR orden:CARDINAL; (*indica hacia donde debe moverse *)
BEGIN
  orden:=0; exito:=FALSE;
REPEAT
  INC(orden);
  fil:=fil + mov_fil[orden];
  col:=col + mov_col[orden];
  IF (1<=fil) AND (fil<=n) AND (1<=col) AND (col<=n) AND
    (lab[fil,col]=0) THEN
    lab[fil,col]:=k;
    IF (fil=n) AND (col=n) THEN exito:=TRUE
    ELSE
      Laberinto(k+1,fil,col,exito);
      IF NOT exito THEN lab[fil,col]:=0 END
    END
  END;
  fil:=fil - mov_fil[orden];
  col:=col - mov_col[orden]
END
UNTIL (exito) OR (orden=4)
END Laberinto;
```

Las variables *mov_fil* y *mov_col* contienen los posibles movimientos, y son inicializadas por el procedimiento *MovimientosPosibles* que mostramos a continuación:

```
VAR mov_fil,mov_col:ARRAY [1..4] OF INTEGER;

PROCEDURE MovimientosPosibles;
BEGIN
  mov_fil[1]:=1; mov_col[1]:=0; (* sur *)
  mov_fil[2]:=0; mov_col[2]:=1; (* este *)
  mov_fil[3]:=0; mov_col[3]:=-1; (* oeste *)
  mov_fil[4]:=-1; mov_col[4]:=0; (* norte *)
END MovimientosPosibles;
```

Tomado de “Diseño de Algoritmos”, de Guerequeta y Vallecillo

Fin