

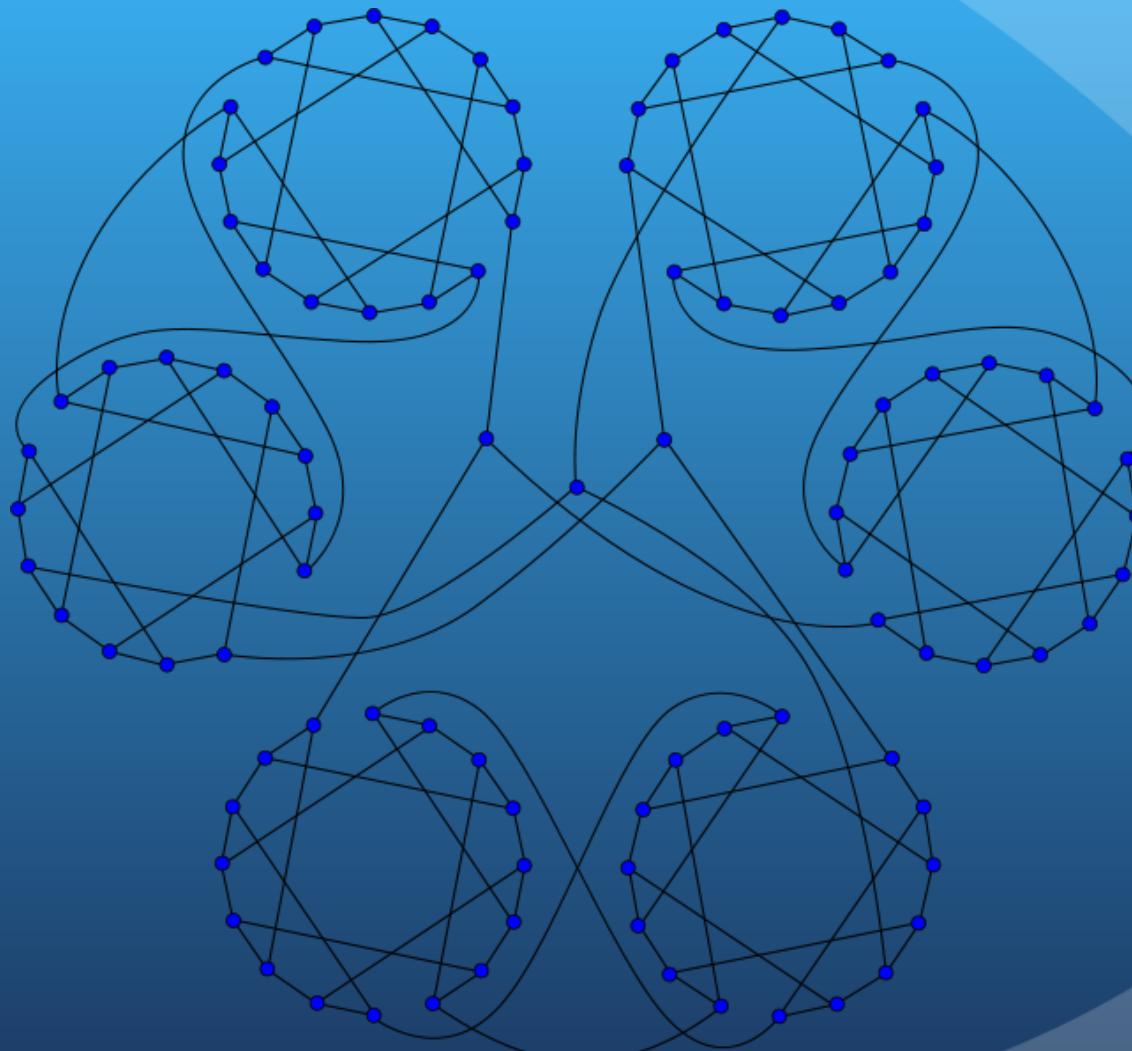


# Estructuras de Datos

## IIS262

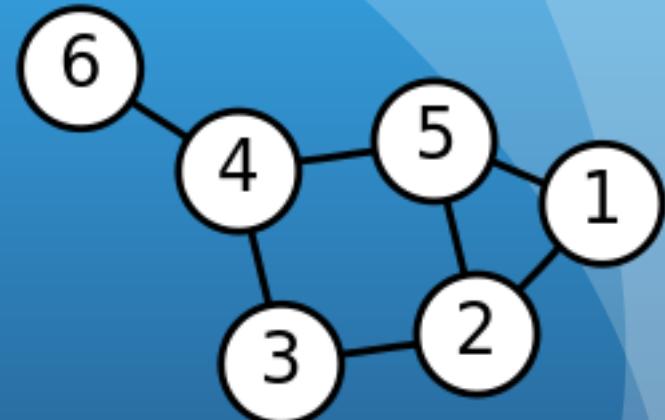
Profesor: Patricio Galeas

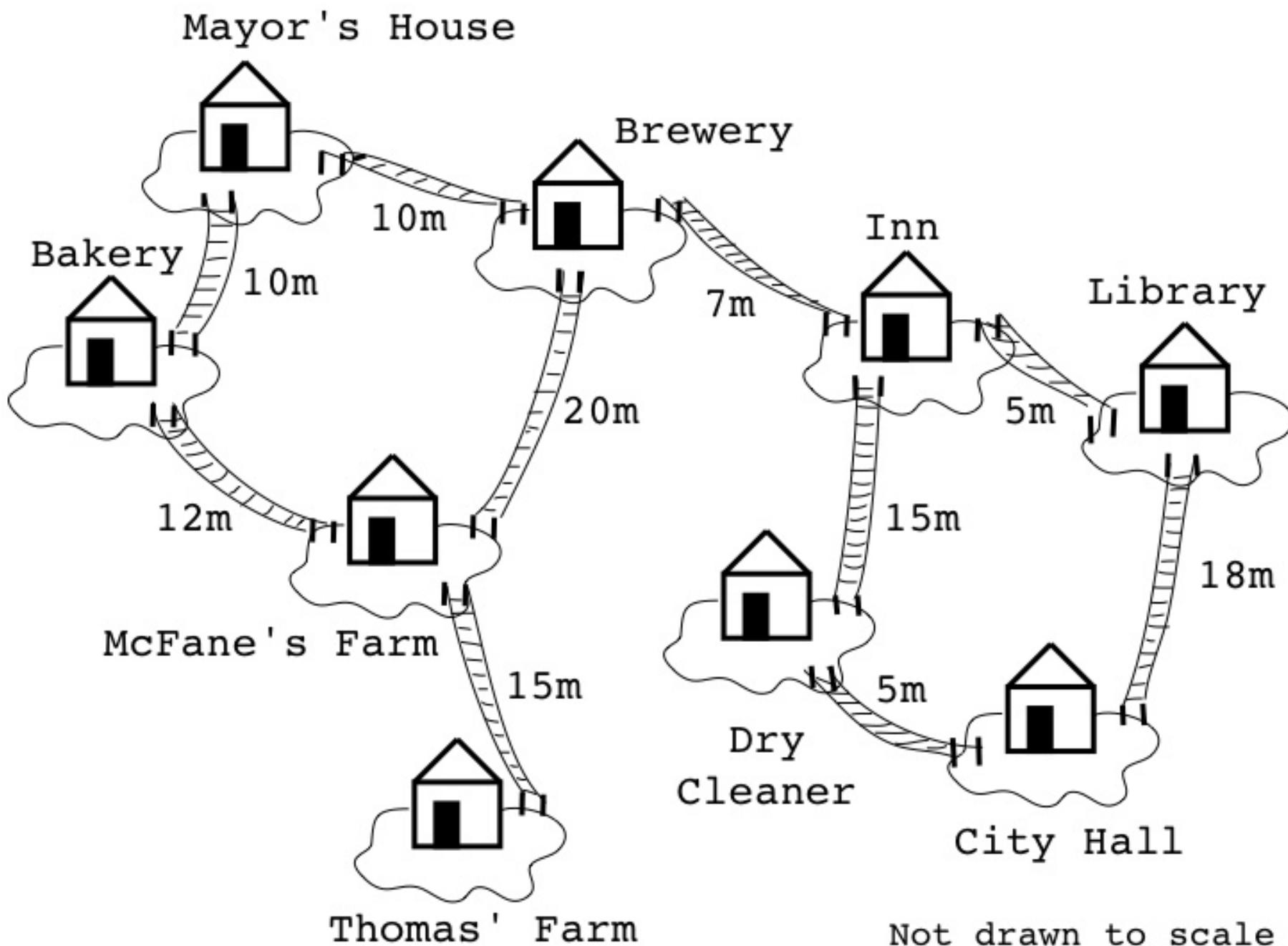
# CAPÍTULO 13 : Grafos



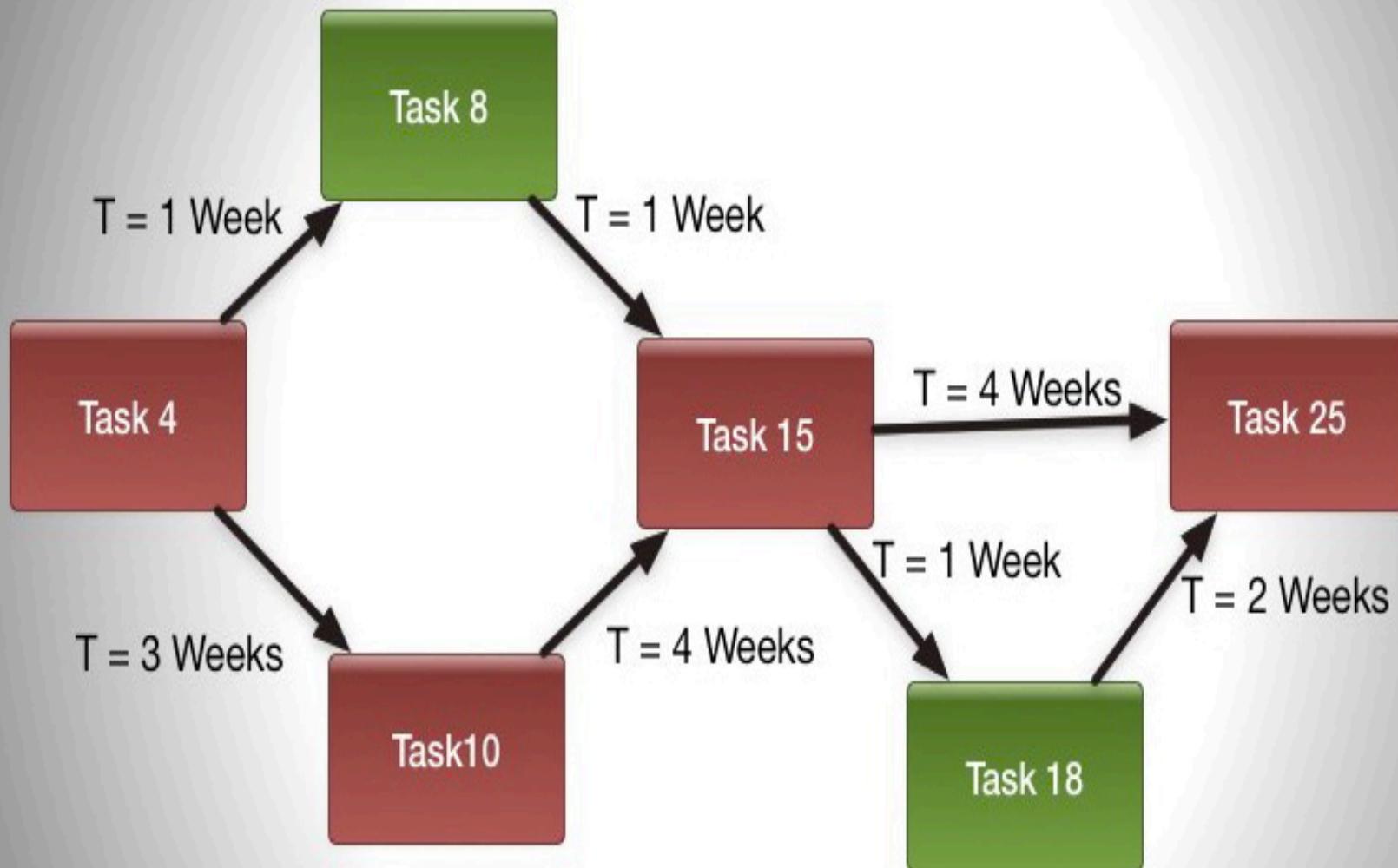
# Introducción

- Los grafos son estructuras más **flexibles** utilizadas en ciencias de la computación
- Estos permiten **solucionar una gran gama de problemas.**
- Los grafos son **parecidos a los árboles.**
- Del punto de vista matemático, **un árbol es un tipo de grafo.**
- La **forma** de los grafos se relaciona generalmente con algún **problema físico o abstracto** que se desea modelar o resolver.

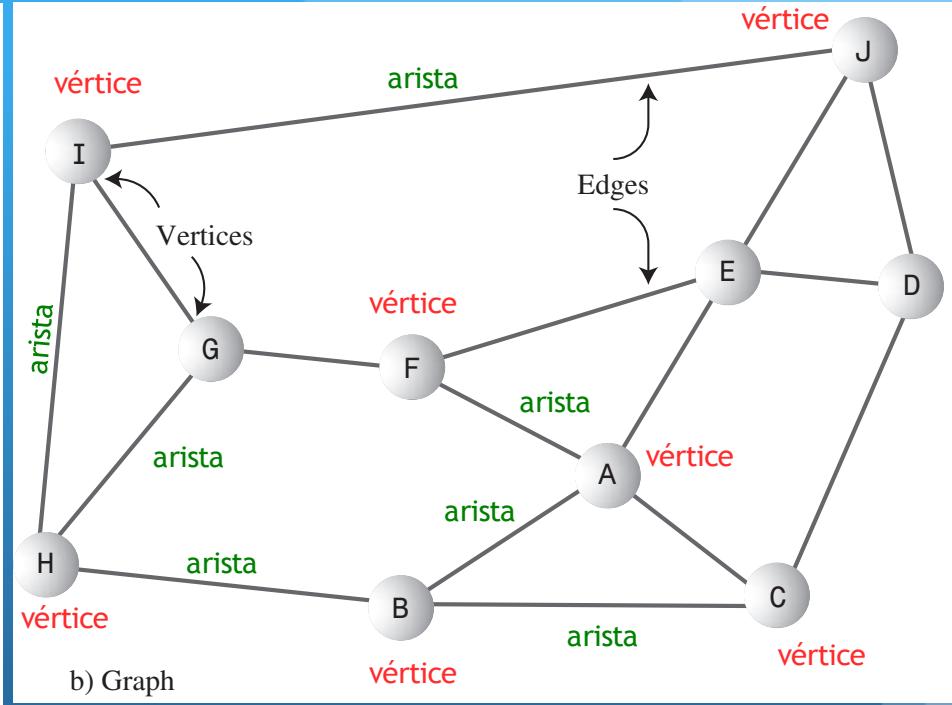
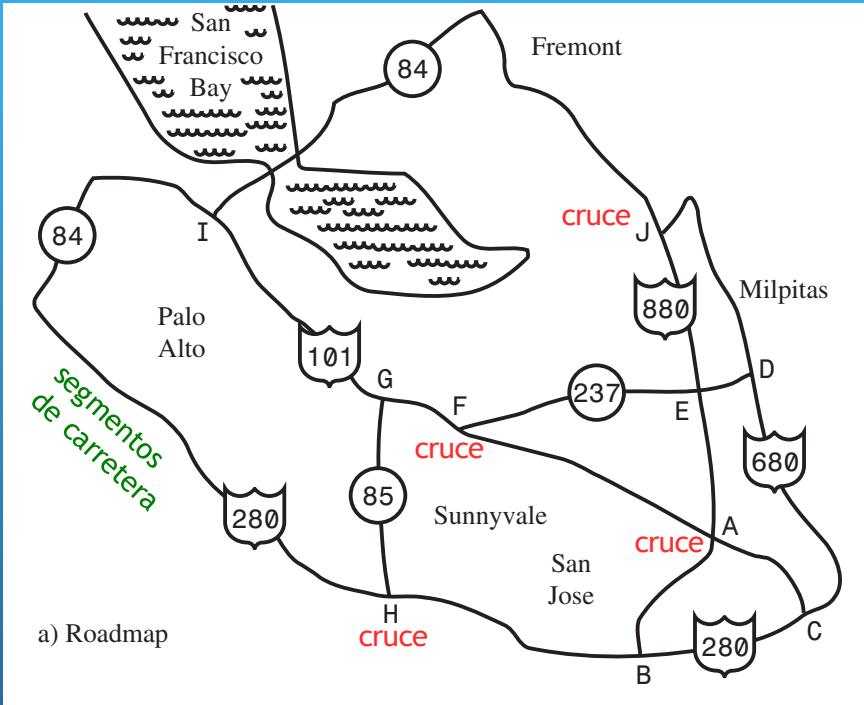








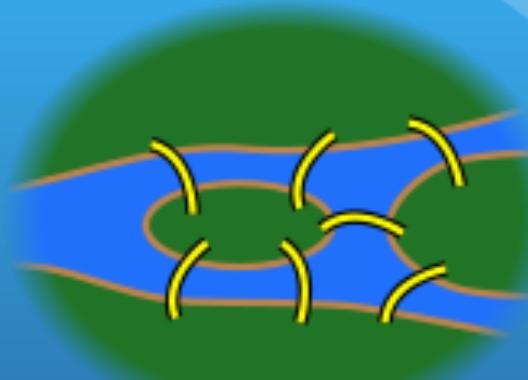
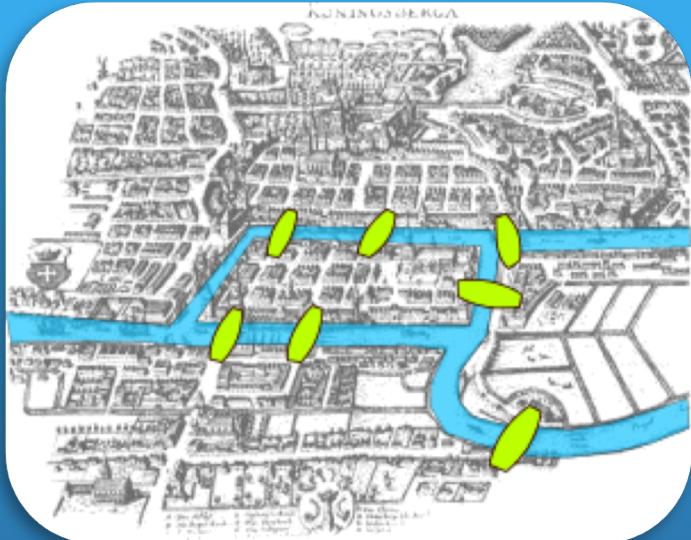
# Definiciones



- **Adyacencia** : nodos adyacentes (vecinos) están conectados por una única arista.
- **Camino** : es una secuencia de aristas. Ej: BAEJ o BCDJ.
- **Grafos Conectados** : son aquellos grafos donde existe al menos un camino entre cada vértice.
- **Orientación** : el grafo anterior es no-orientado, ya que las aristas no tienen dirección. En el caso contrario existe una dirección única entre dos vértices.
- **Pesos** : es cuando las aristas tienen asociados ciertos valores: distancias, costos de viaje, etc.

# Historia

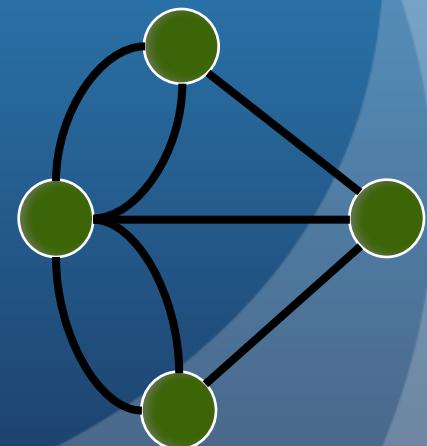
Los 7 Puentes de la ciudad de Königsberg (hoy Caliningrado)



**Leonhard Euler**  
Suizo (1707 - 1782),  
principal matemático del  
siglo XVIII



- **Problema:** Dado el mapa de Königsberg, con el río Pregolya dividiendo el plano en cuatro regiones, unidas a través de los siete puentes. ¿Es posible dar un paseo comenzando desde cualquiera de estas regiones, pasando por todos los puentes, recorriendo sólo una vez cada uno, y regresando al mismo punto de partida?
- La **solución** de este problema dio origen a la **teoría de grafos** (1736).
- La **respuesta** es que no existe una ruta con estas características, demostrada en forma generalizada por **Euler** en su publicación: “*Solutio problematis ad geometriam situs pertinentis*”



# Representando Grafos en un Programa

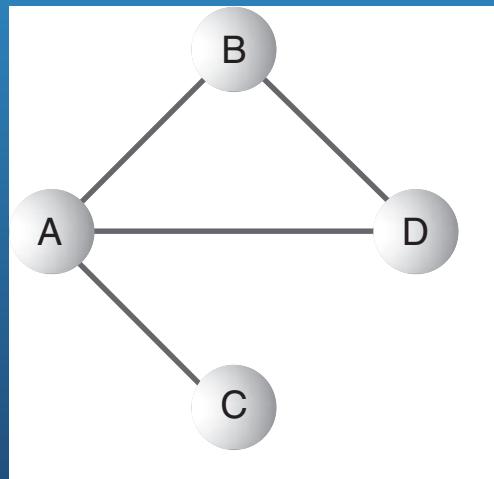
- Los Vértices : pueden (dependiendo del tipo de problema) contener distintos tipo de información. Luego, los podemos representar como objetos de una clase vértice (Vertex).
- Los vértices pueden ser almacenados en cualquier tipo de estructura. Por ejemplo: un arreglo (vertexList).
- Las Aristas son representadas a través de Matrices de Adyacencia o Listas de Adyacencia.

```
class Vertex
{
    public char label;          // label (e.g. 'A')
    public boolean wasVisited;

    public Vertex(char lab)    // constructor
    {
        label = lab;
        wasVisited = false;
    }
} // end class Vertex
```

# Matrices de Adyacencia

- Son **arreglos bidimensionales** que indican si existe una arista entre dos vértices
- En un grafo de N-vértices la matriz asociada es un arreglo de **NxN** elementos.



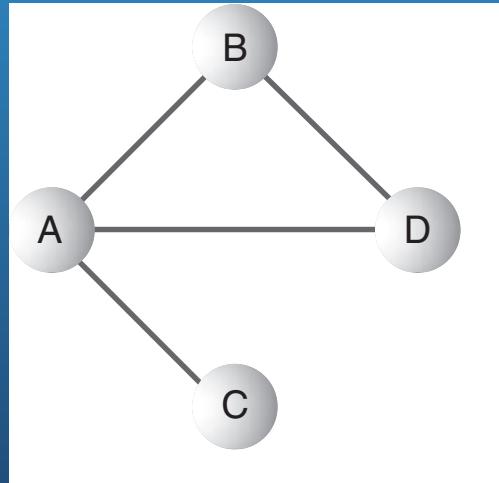
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

## Observaciones

- La diagonal no presenta mayor información
- La matriz es simétrica

# Lista de Adyacencia

- Otra forma de representar aristas son las **listas enlazadas**.
- En este caso se trata en realidad de **arreglos de listas** o **listas de listas**.
- Cada lista muestra cuales son los vértices adyacentes a un vértice determinado.



Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

# Agregando Vértices a un Grafo

Nuevo vértice

```
vertexList[nVerts++] = new Vertex('F');
```

Arreglo de vértices

- Para **agregar las aristas** del grafo, se utiliza la **matriz de adyacencia** o la **lista de adyacencia**.
- Por ejemplo: supongamos que queremos agregar una arista entre los vértices (1) y (3) del grafo. Donde (1) y (3) son las posiciones de los vértices en el arreglo vertexList.
- Se **crea una matriz adjMat**, la cual llenamos inicialmente con **ceros**.
- Y luego redefinimos los vértices:  
 $\text{adjMat}[1][3] = 1$   
 $\text{adjMat}[3][1] = 1$

# La Clase : Graph

## Dentro de la clase Graph:

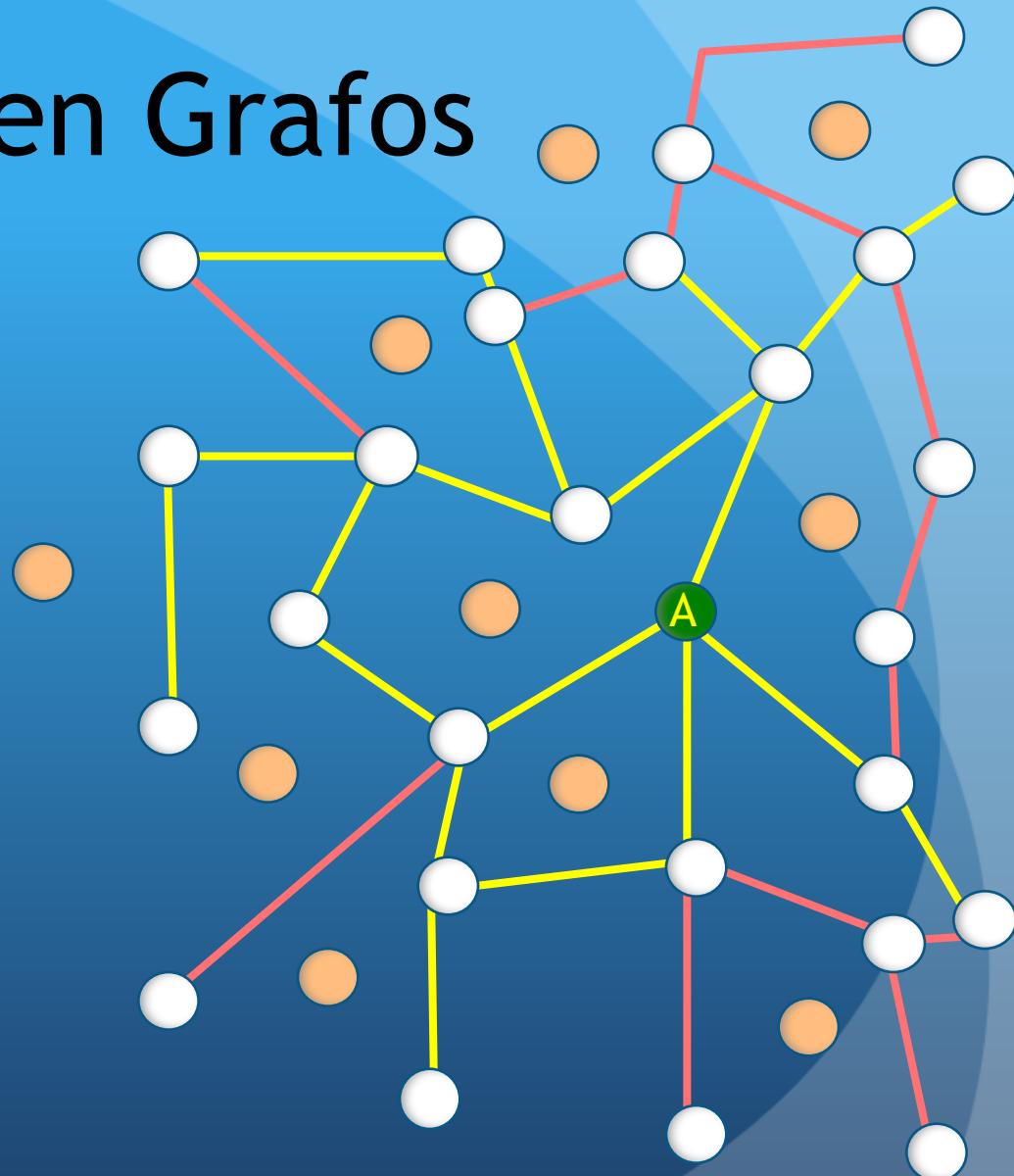
- Los vértices son identificados por sus índices en `vertexList`.
- La matriz de adyacencia `adjMat` provee información asociada a cada vértice: que vértices están conectados por una única arista a otro vértice.
- Para contestar preguntas mas generales con respecto a la ubicación de los vértices, existen varios algoritmos.

```
class Graph
{
    private final int MAX_VERTS = 20;
    private Vertex vertexList[]; // array of vertices
    private int adjMat[][];      // adjacency matrix
    private int nVerts;          // current number of vertices
// -----
    public Graph()                // constructor
    {
        vertexList = new Vertex[MAX_VERTS];
                                // adjacency matrix
        adjMat = new int[MAX_VERTS][MAX_VERTS];
        nVerts = 0;
        for(int j=0; j<MAX_VERTS; j++)      // set adjacency
            for(int k=0; k<MAX_VERTS; k++) // matrix to 0
                adjMat[j][k] = 0;
    } // end constructor
// -----
    public void addVertex(char lab) // argument is label
    {
        vertexList[nVerts++] = new Vertex(lab);
    }
// -----
    public void addEdge(int start, int end)
    {
        adjMat[start][end] = 1;
        adjMat[end][start] = 1;
    }
// -----
    public void displayVertex(int v)
    {
        System.out.print(vertexList[v].label);
    }
// -----
} // end class Graph
```

# Búsquedas en Grafos

## Ejemplo 1

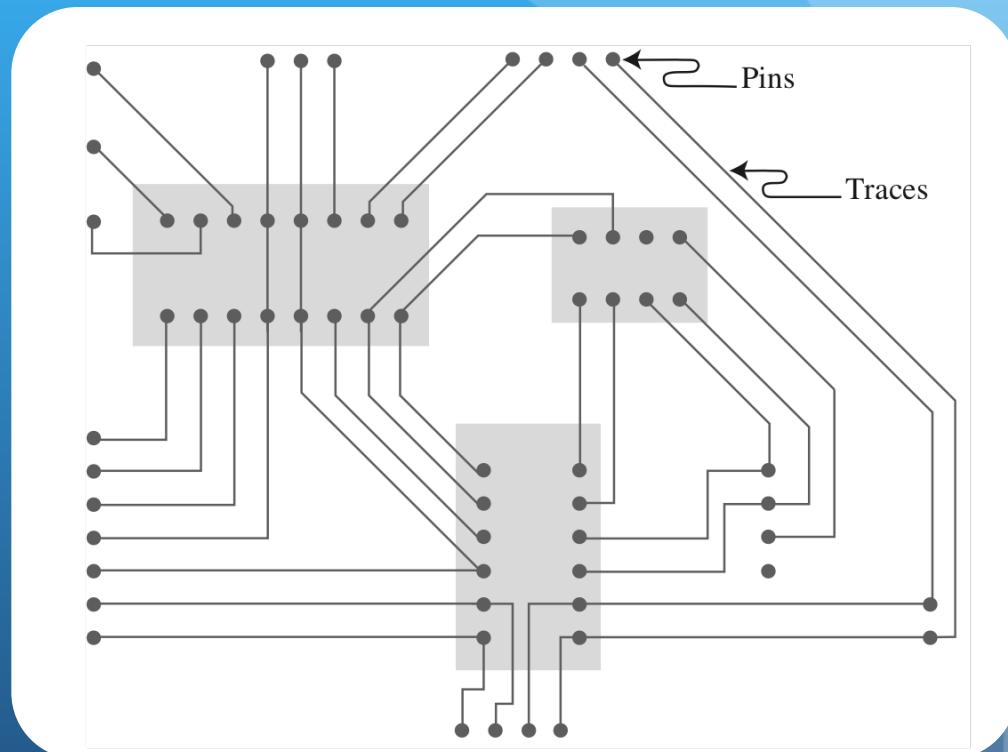
¿ A que ciudades se puede llegar desde la ciudad **A** sin cambiar de tren ?



# Búsquedas en Grafos

## Ejemplo 2 :

- En el circuito integrado, ¿cómo encontrar todos los vértices alcanzables desde un vértice en particular?
- Sería útil generar un grafo que indique que “pins” están conectados en un circuito electrónico...
- Una vez generado el grafo necesitamos el algoritmo que a partir de un vértice se mueva en forma sistemática a lo largo del grafo y que garantice una visita de todos los vértices conectados al nodo de partida...

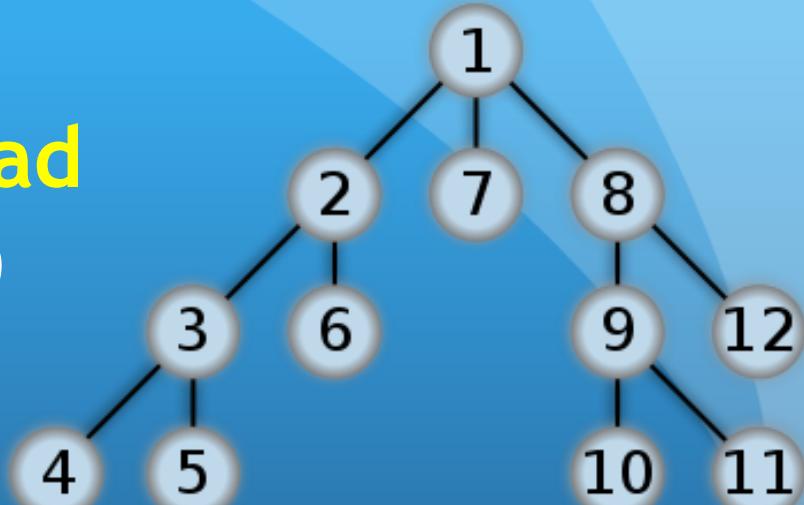


... algo parecido al algoritmo de visita de nodos implementada en árboles binarios

# Algoritmos para Búsquedas en Grafos

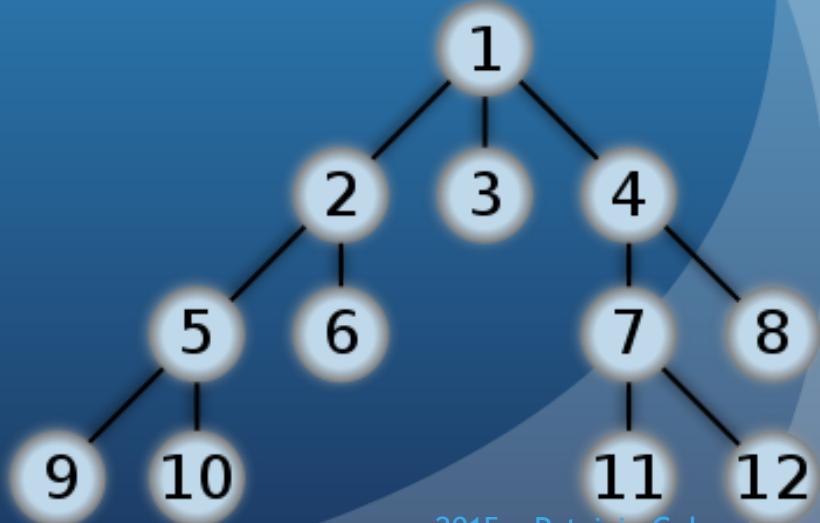
Búsqueda en Profundidad  
Deep-First Search (DFS)

Se implementa con una Pila



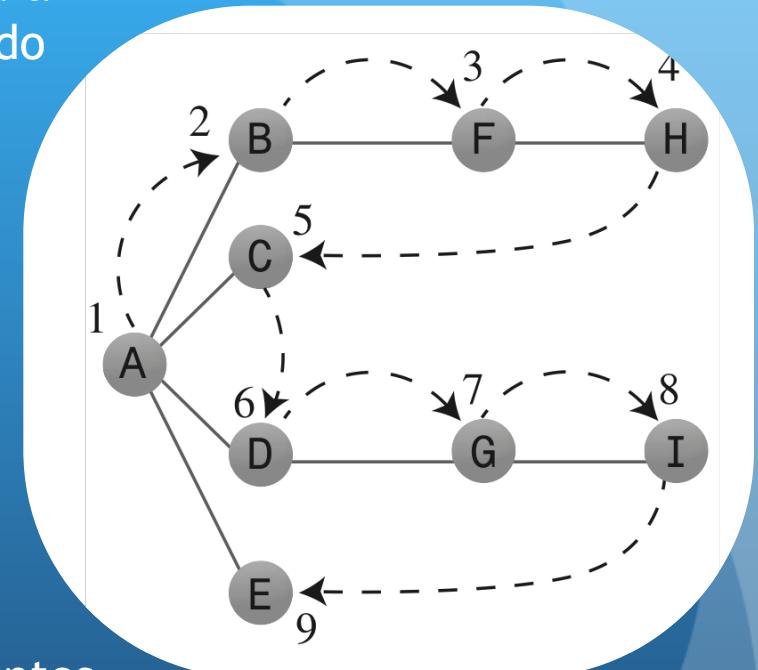
Búsqueda en Anchura  
Breadth-First Search  
(BFS)

Se implementa con una Cola



# Búsqueda en Profundidad

- Búsqueda en Profundidad **utiliza una pila** para recordar donde ir cuando se alcanza un nodo terminal.
- **Receta:**
  - Posicionarse en un vértice
  - ... y luego hacer 3 cosas:
    - Visitar el vértice.
    - Colocarlo en la pila.
    - Marcarlo.
- **Reglas**
  1. Mientras sea posible, **visite un vértice adyacente** que no halla sido visitado antes.
  2. Si no se puede ejecutar la Regla 1, entonces si es posible, **saque un vértice de la pila**.
  3. Si no se puede ejecutar ni la Regla 1 ni la Regla 2, **el procedimiento ha finalizado**.



**Analogía :** recorrer un laberinto utilizando una cuerda y un marcador. La cuerda es la pila.

# Applet de la Búsqueda en Profundidad

AppletViewer: GraphN.class

New DFS BFS Tree View

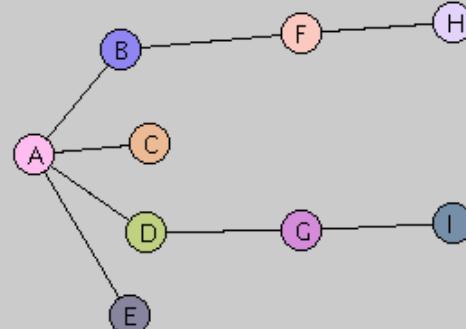
Double-click mouse to make vertex

Grafo

AppletViewer: GraphN.class

New DFS BFS Tree View

Single-click on vertex from which to start tree



Subprograma iniciado.

Visits:

Stack (b-->t):

Subprograma iniciado.

GraphN



Matriz de Adyacencia

AppletViewer: GraphN.class

New DFS BFS Tree View

Press View button again to show graph

	A	B	C	D	E	F	G	H	I
A	0	1	1	1	1	0	0	0	0
B	1	0	0	0	0	1	0	0	0
C	1	0	0	0	0	0	0	0	0
D	1	0	0	0	0	0	1	0	0
E	1	0	0	0	0	0	0	0	0
F	0	1	0	0	0	0	0	1	0
G	0	0	0	1	0	0	0	0	1
H	0	0	0	0	0	1	0	0	0
I	0	0	0	0	0	0	1	0	0

Subprograma iniciado.

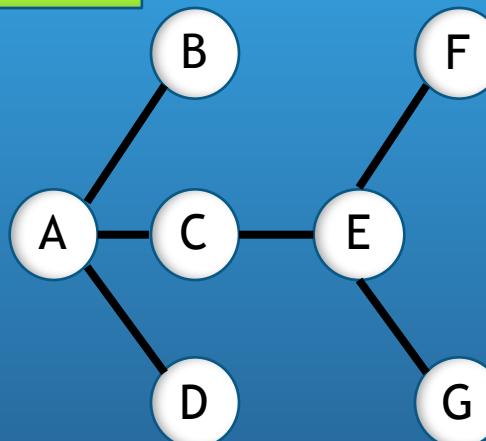
2015 - Patricio Galeas

# Código de la Búsqueda en Profundidad

- La clave del algoritmo es :

- Encontrar los vértices que **no visitados**
- Encontrar los vértices **adyacentes**

se hace con



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

matriz de adyacencia

Este método retorna un vértice no visitado adyacente a “v”

```

public int getAdjUnvisitedVertex(int v)
{
    for(int j=0; j<nVerts; j++)
        if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j; // return first such vertex
    return -1; // no such vertices
} // end getAdjUnvisitedVertex()
    
```

Ahora veamos el método de búsqueda de profundidad (DFS) ...

# Código de la Búsqueda en Profundidad

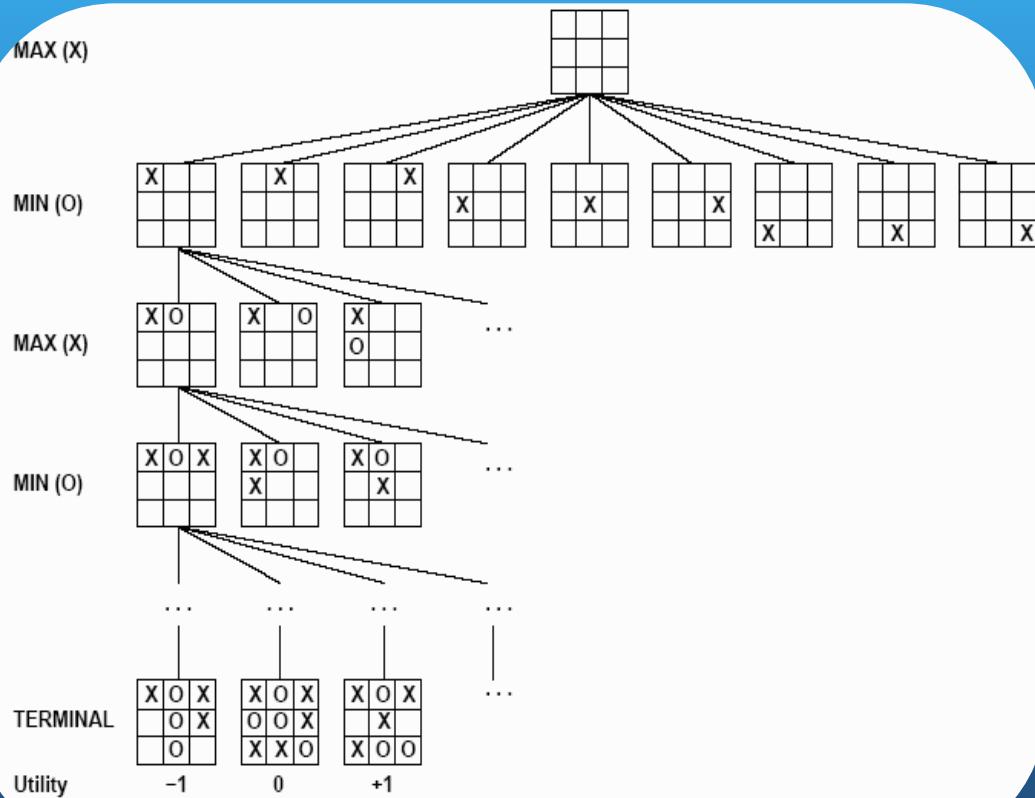
```
public void dfs() // depth-first search
{
    vertexList[0].wasVisited = true; // begin at vertex 0
    displayVertex(0); // mark it
    theStack.push(0); // push it

    while( !theStack.isEmpty() ) // until stack empty,
    {
        // get an unvisited vertex adjacent to stack top
        int v = getAdjUnvisitedVertex( theStack.peek() );
        if(v == -1) // if no such vertex,
            theStack.pop(); // pop a new one
        else // if it exists,
        {
            vertexList[v].wasVisited = true; // mark it
            displayVertex(v); // display it
            theStack.push(v); // push it
        }
    } // end while

    // stack is empty, so we're done
    for(int j=0; j<nVerts; j++) // reset flags
        vertexList[j].wasVisited = false;
} // end dfs
```

- Este es el método DFS de la clase Graph, que ejecuta la búsqueda en profundidad.
- Aquí se puede observar como como este código sigue las 3 reglas mencionadas previamente.
- El ciclo itera hasta que la pila (theStack) esta vacía.
- Dentro del ciclo se hacen 4 cosas:
  1. Examina el vértice en la cima de la pila
  2. Trata de encontrar un vecino no visitado de este vértice.
  3. Si no lo encuentra saca al vértice de la pila.
  4. Si lo encuentra, lo visita y lo agrega a la pila.

# Búsqueda en Profundidad y Simulación de Juegos



El juego del Gato puede ser representado a través de un grafo.

La cantidad de caminos posibles es:

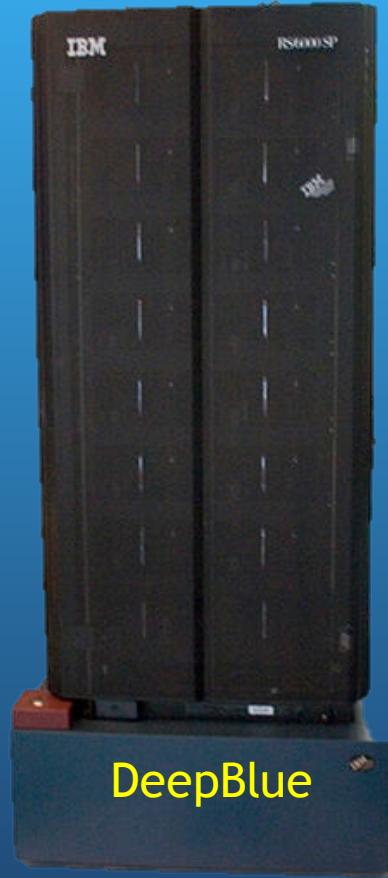
$$9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 9! \\ = 362,880$$

Uno podría “visualizar” el final del juego.

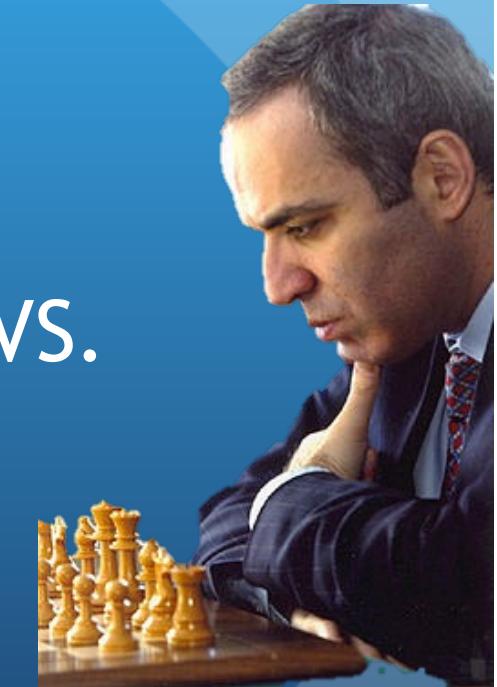
# Búsqueda en Profundidad y Simulación de Juegos

En otros juegos como el Ajedrez, hay muchas más posibilidades.

Solo es posible calcular si una ruta es más o menos favorable para ganar.



Gasparov



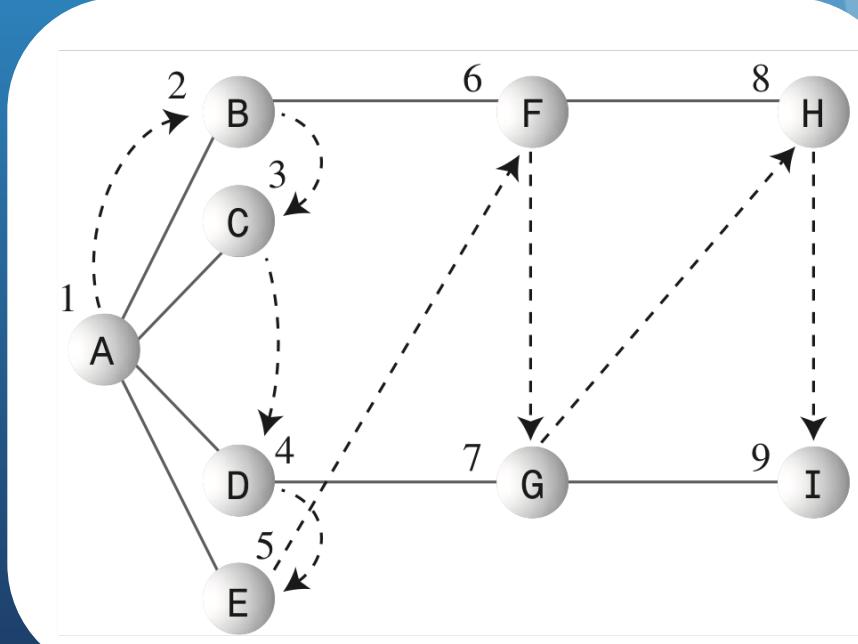
VS.

## Búsqueda en Amplitud

- La Búsqueda en Profundidad actúa alejándose lo más posible del punto de partida.
- La Búsqueda en Amplitud, se intenta permanecer lo más cerca posible del punto de partida.
- La idea es visitar **todos** los vértices adyacentes al inicio y luego continuar visitando otros vértices.
- Este tipo de búsqueda es implementada utilizando una Cola.

### Reglas

1. Visite el **próximo vértice no visitado** que sea adyacente al vértice actual. Márquelo e insértelo en la cola.
2. Si no se puede ejecutar la Regla 1, porque todos los vértices están visitados. Remueve un vértice de la cola y defínalo como vértice actual.
3. Si no se puede ejecutar ni la Regla 2, por que la cola esta vacía, **el procedimiento ha finalizado**.



# Applet de la Búsqueda en Amplitud

AppletViewer: GraphN.class

New DFS BFS Tree View

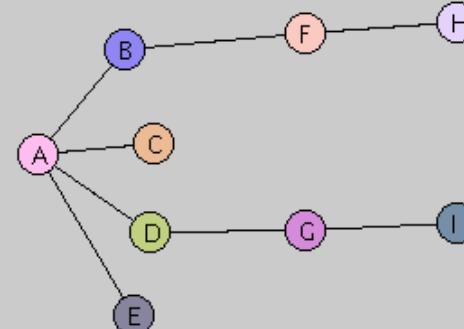
Double-click mouse to make vertex

Grafo

AppletViewer: GraphN.class

New DFS BFS Tree View

Single-click on vertex from which to start tree



Subprograma iniciado.

Visits:

Stack (b-->t):

Subprograma iniciado.

GraphN



Matriz de Adyacencia

AppletViewer: GraphN.class

New DFS BFS Tree View

Press View button again to show graph

	A	B	C	D	E	F	G	H	I
A	0	1	1	1	1	0	0	0	0
B	1	0	0	0	0	1	0	0	0
C	1	0	0	0	0	0	0	0	0
D	1	0	0	0	0	0	1	0	0
E	1	0	0	0	0	0	0	0	0
F	0	1	0	0	0	0	0	1	0
G	0	0	0	1	0	0	0	0	1
H	0	0	0	0	0	1	0	0	0
I	0	0	0	0	0	0	1	0	0

Subprograma iniciado.

2015 - Patricio Galeas

# Código de la Búsqueda en Amplitud

```
public void bfs()                                // breadth-first search
{
    vertexList[0].wasVisited = true; // mark it
    displayVertex(0);             // display it
    theQueue.insert(0);           // insert at tail
    int v2;

    while( !theQueue.isEmpty() )      // until queue empty,
    {
        int v1 = theQueue.remove();   // remove vertex at head
        // until it has no unvisited neighbors
        while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
        {
            vertexList[v2].wasVisited = true; // mark it
            displayVertex(v2);             // display it
            theQueue.insert(v2);           // insert it
        } // end while(unvisited neighbors)
    } // end while(queue not empty)

    // queue is empty, so we're done
    for(int j=0; j<nVerts; j++)          // reset flags
        vertexList[j].wasVisited = false;
    } // end bfs()
```

- El método **de búsqueda en amplitud** `bfs()`, es similar al método de **búsqueda en profundidad** `dfs()`.
- Excepto por el uso de una **cola** en vez de una pila. Y ejecuta **ciclos anidados** en vez de un solo ciclo.

# Propiedad de la Búsqueda en Amplitud

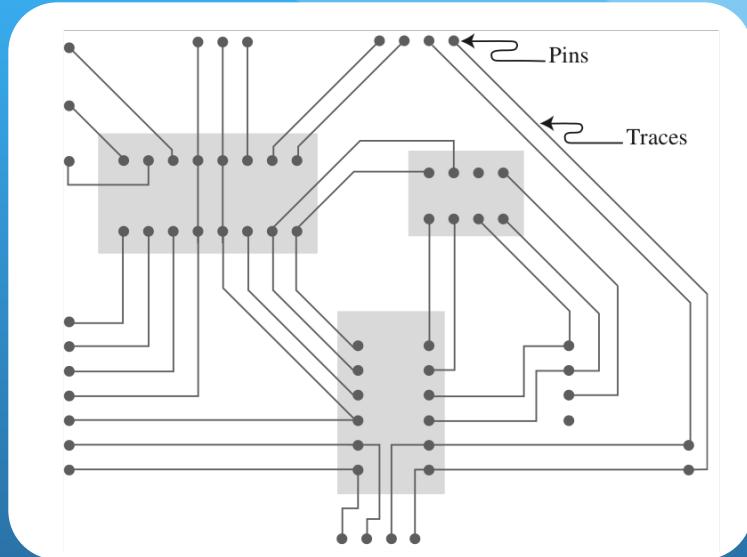
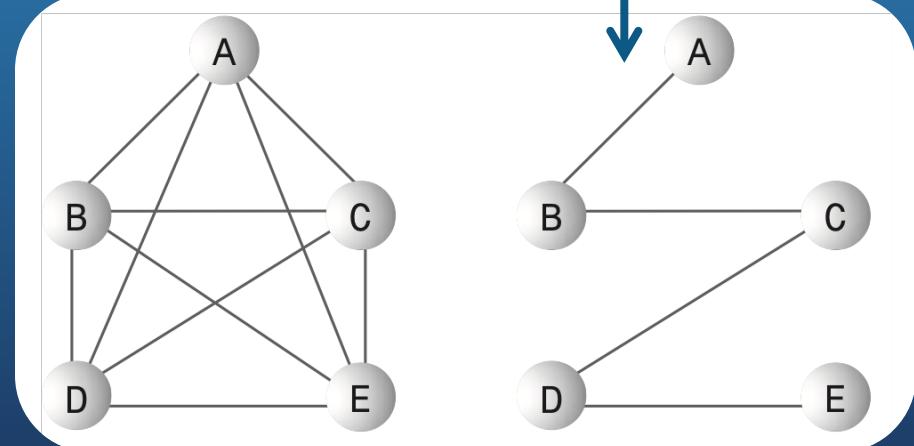
- El método de búsqueda en amplitud tiene una interesante propiedad:
- ... encuentra todos los vértices que se encuentran a una arista de distancia del punto de partida, luego todos los vértices que están a dos aristas, etc.
- ... esto es útil cuando se intenta buscar el camino más corto desde un vértice de partida a otro vértice.
- ¿cómo? .. Cuando se llega a un vértice específico, el camino recorrido es conocido, siendo este el más corto desde el vértice de inicio a este vértice.



# Árbol de Recubrimiento Mínimo (Minimum Spanning Tree)

Supongamos que deseamos diseñar un **circuito** como el de la figura, pero además estar seguros de haber usado la **cantidad mínima de conexiones** y así minimizar el tamaño y la dificultad del circuito.

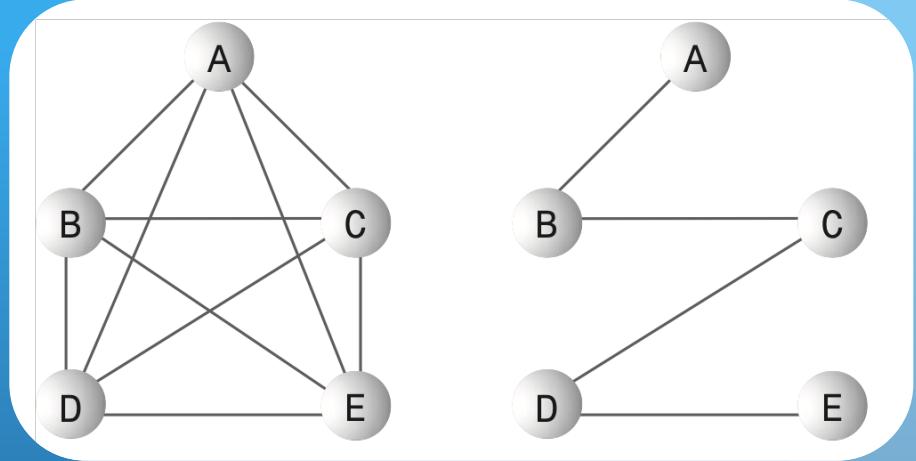
Un MST



Sería interesante tener un **algoritmo** que remueva todo las conexiones redundantes, obteniendo como resultado un **grafo** con el **número mínimo de aristas** para conectar todos los vértices.

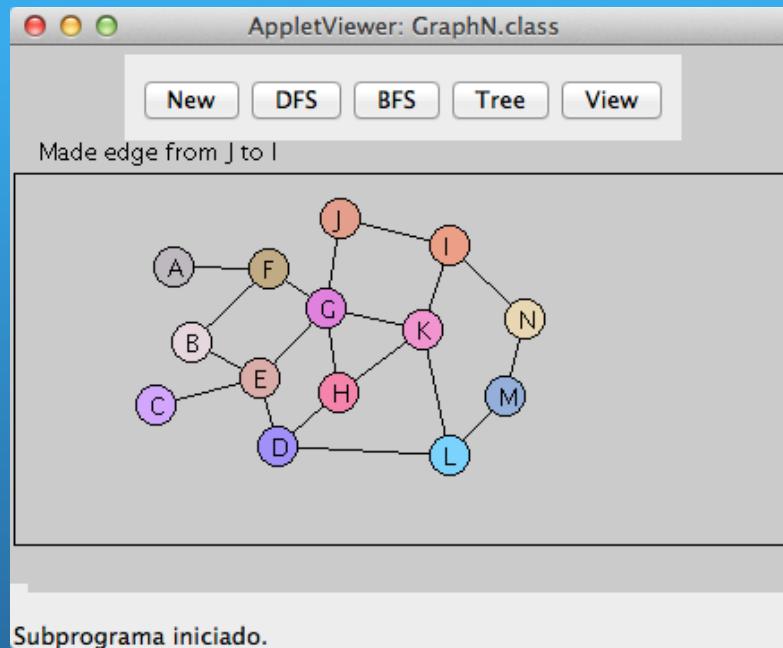
# Árbol de Recubrimiento Mínimo

- Para un grafo determinado, existen varios ARM posibles.
- En general, el número de aristas en un ARM es siempre uno menos que el número de vértices.  
*Aristas = Vértices - 1*
- El algoritmo para crear ARM son casi iguales a los de búsqueda vistos anteriormente.
- Si ejecutamos la Búsqueda en Profundidad y grabamos las aristas que recorremos, crearemos automáticamente un ARM.

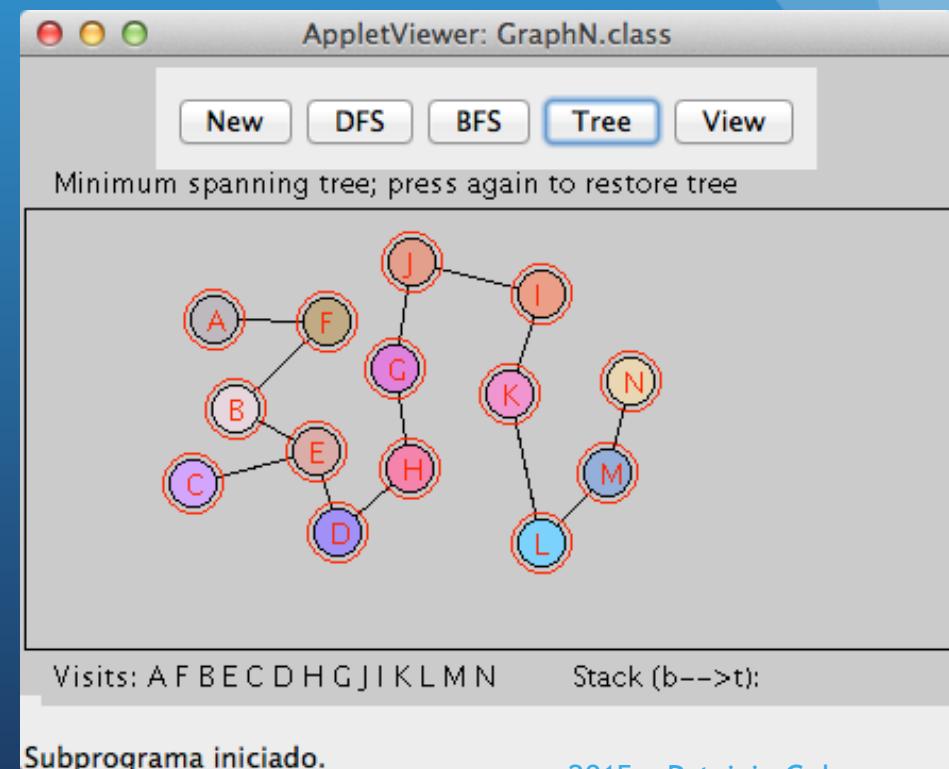


Sería interesante tener un algoritmo que remueva todo las conexiones redundantes, obteniendo como resultado un grafo con el número mínimo de aristas para conectar todos los vértices.

# Applet - Árbol de Recubrimiento Mínimo



- Presionando el botón “Tree”, el algoritmo del applet creará un Árbol de Recubrimiento Mínimo del grafo que hayamos definido.



# Código - Árbol de Recubrimiento Mínimo

```

while( !theStack.isEmpty() )      // until stack empty
{
    int currentVertex = theStack.peek(); // get stack top
    // get next unvisited neighbor
    int v = getAdjUnvisitedVertex(currentVertex);
    if(v == -1)                      // if no more neighbors
        theStack.pop();             // pop it away
    else                            // got a neighbor
    {
        vertexList[v].wasVisited = true; // mark it
        theStack.push(v);           // push it
        displayVertex(v);          // display edge
        displayVertex(currentVertex); // from currentV
        displayVertex(v);          // to v
        System.out.print(" ");
    }
} // end while(stack not empty)

// stack is empty, so we're done
for(int j=0; j<nVerts; j++)      // reset flags
    vertexList[j].wasVisited = false;
} // end mst()

```

```

public void dfs() // depth-first search
{
    vertexList[0].wasVisited = true; // begin at vertex 0
    displayVertex(0);             // mark it
    theStack.push(0);             // display it
    while( !theStack.isEmpty() )      // until stack empty,
    {
        // get an unvisited vertex adjacent to stack top
        int v = getAdjUnvisitedVertex( theStack.peek() );
        if(v == -1)                  // if no such vertex,
            theStack.pop();         // pop a new one
        else                          // if it exists,
        {
            vertexList[v].wasVisited = true; // mark it
            displayVertex(v);           // display it
            theStack.push(v);          // push it
        }
    } // end while

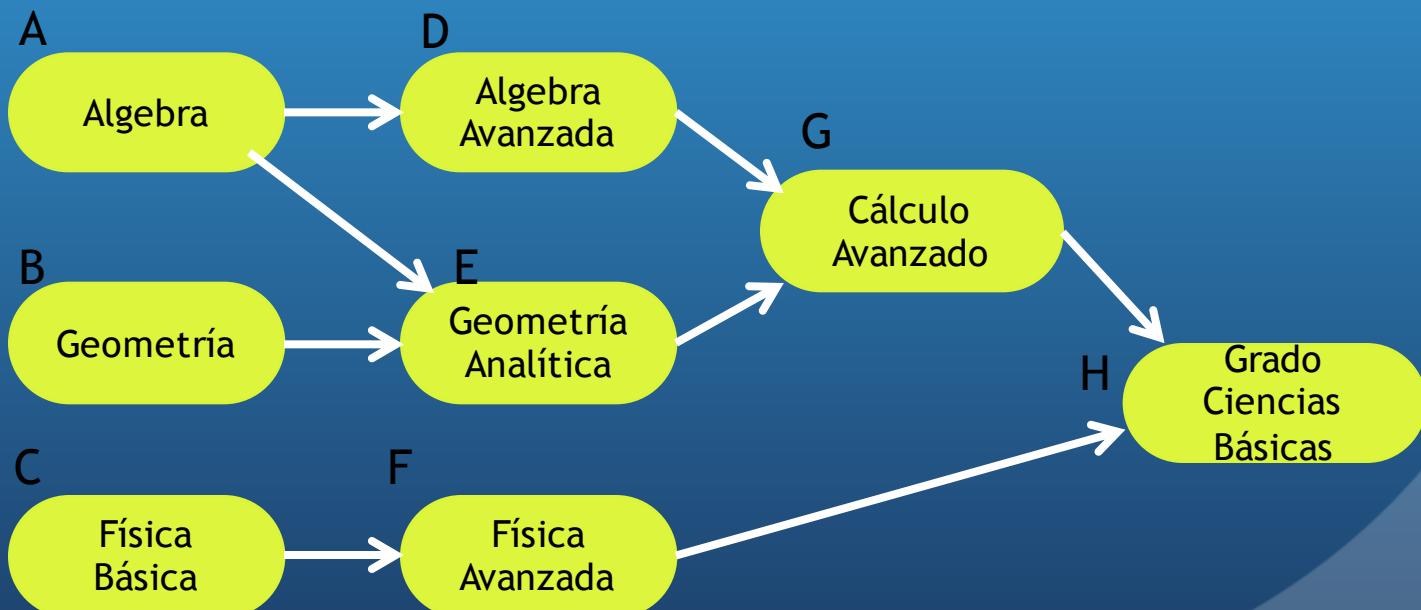
    // stack is empty, so we're done
    for(int j=0; j<nVerts; j++)      // reset flags
        vertexList[j].wasVisited = false;
} // end dfs

```

El código del ARM es muy parecido al de la búsqueda en profundidad (dfs). La diferencia esta en el “else” donde el vértice actual y el próximo son desplegados.

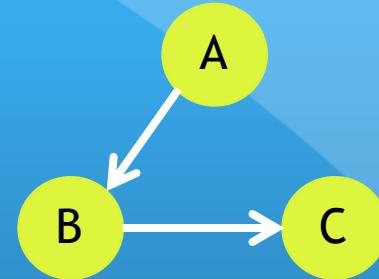
# Ordenamiento Topológico con Grafos Dirigidos

El ordenamiento topológico es otra operación que puede ser modelada con grafos. Esto es útil cuando los ítems o eventos tienen que ser dispuestos en un orden específico.



# Grafos Dirigidos

- Es un grafo donde las **aristas tienen dirección**, y el grafo sólo puede ser recorrido según esta dirección.
- Su **Matriz de Adyacencia** difiere en que hay **sólo una entrada por arista**. Los valores de la matriz **no son simétricos** (reflejado).

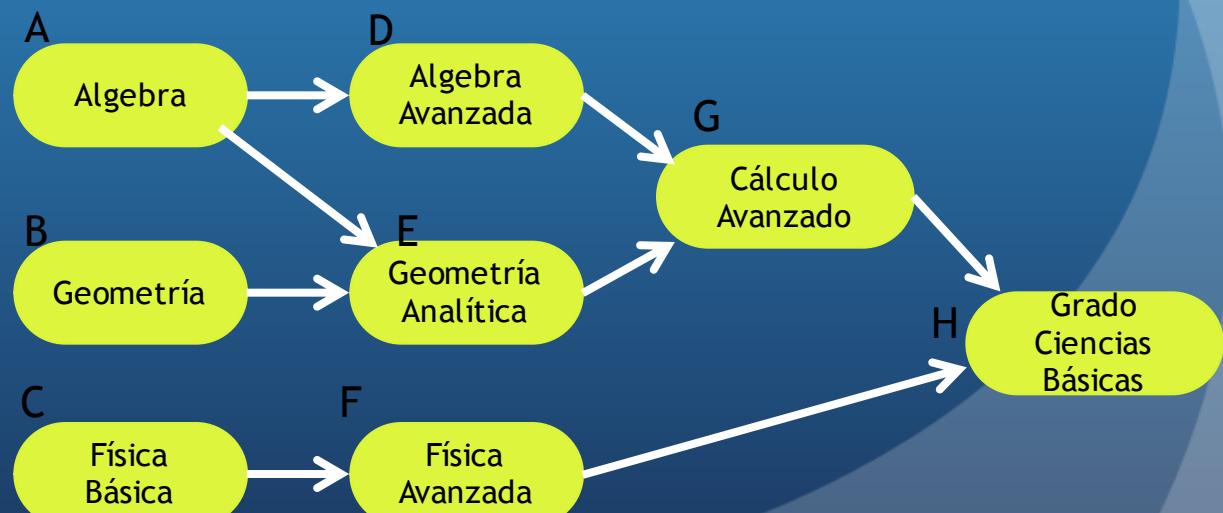


	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

```
public void addEdge(int start, int end) // directed graph
{
    adjMat[start][end] = 1;
}
```

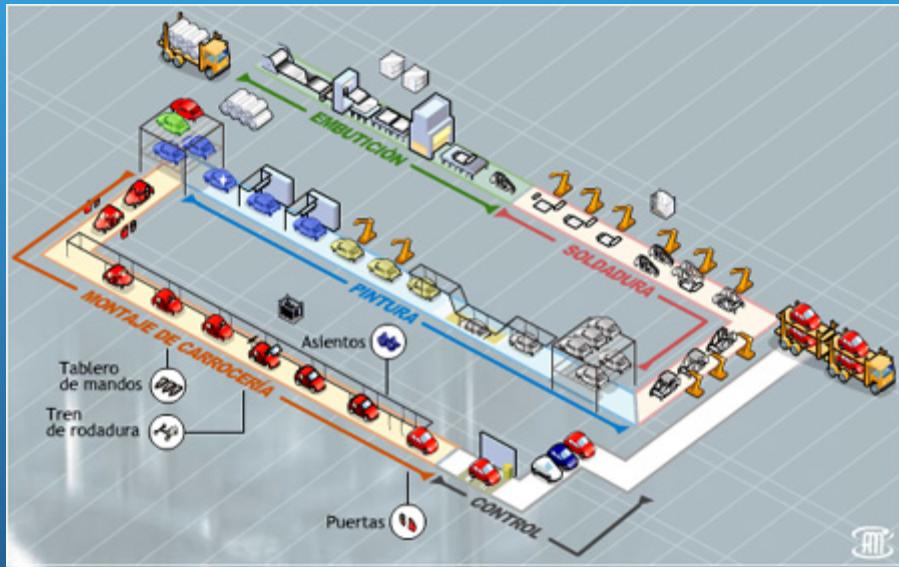
# Ordenamiento Topológico

- Considerando el ejemplo anterior, una forma de obtener el Grado en Ciencias Básicas, sería por ejemplo a través de la secuencia:  
**B A E D G C F H.**
- Ordenado de esta forma se dice que el grafo estaría **topológicamente ordenado**. Cada curso es tomado considerando el prerequisito definido a través de las aristas.
- Sin embargo, hay otras soluciones que satisfacen los prerequisitos :  
**C F B A E D G H**



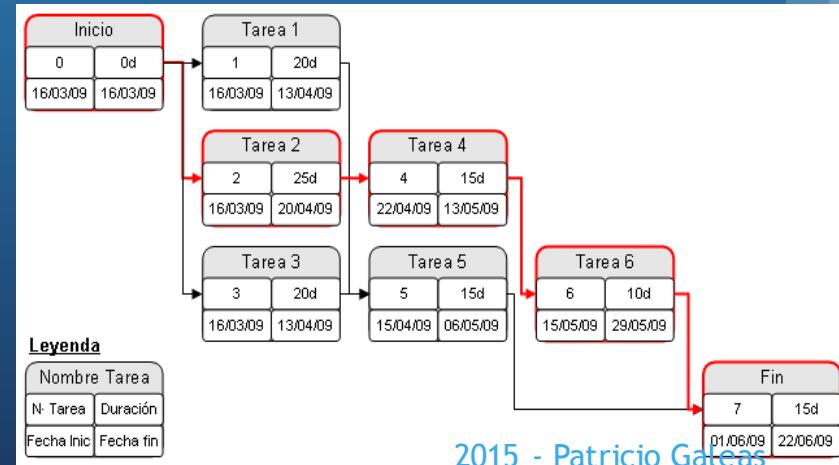
# Ordenamiento Topológico

Hay muchos otros ejemplos que se pueden modelar utilizando ordenamiento topológico.



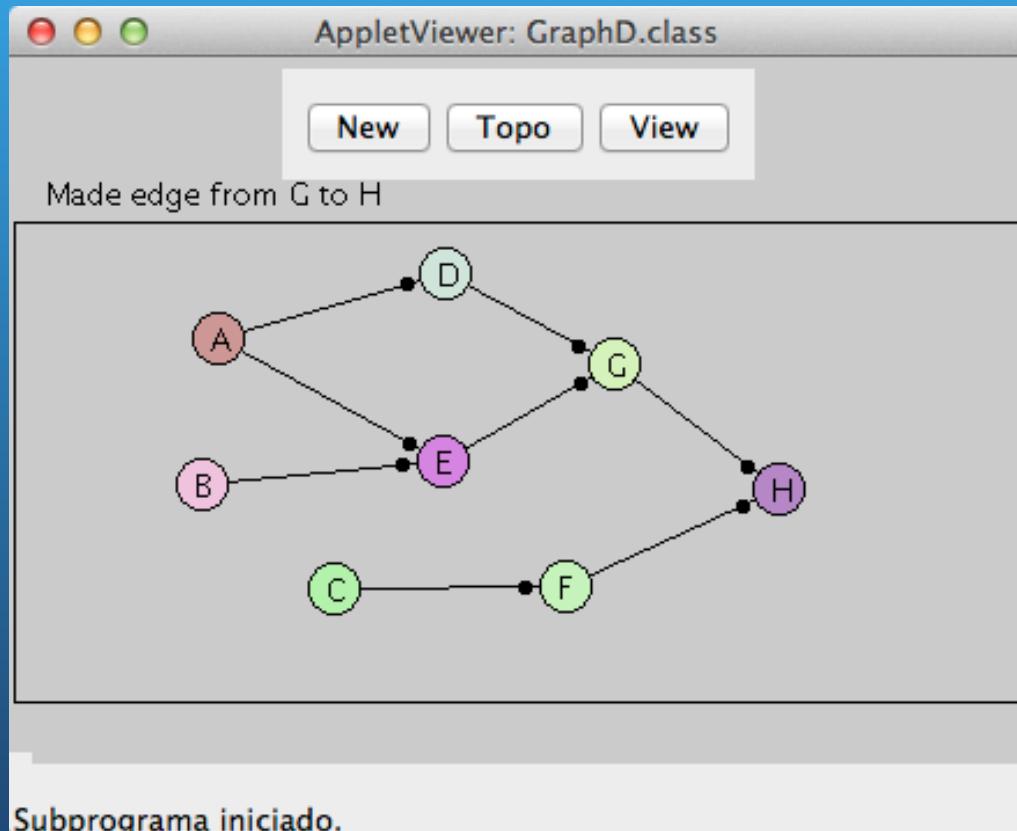
Agendas de trabajo

Manufactura de Automóviles



# Applet de Ordenamiento Topológico

- El applet GraphD modela grafos dirigidos.



Para el ordenamiento topológicos sólo se requieren 2 pasos:

- **Paso 1 :** Encuentre un vértice que no tenga sucesores
- **Paso 2 :** Elimine este vértice del grafo, y registre su valor al comienzo de una lista.

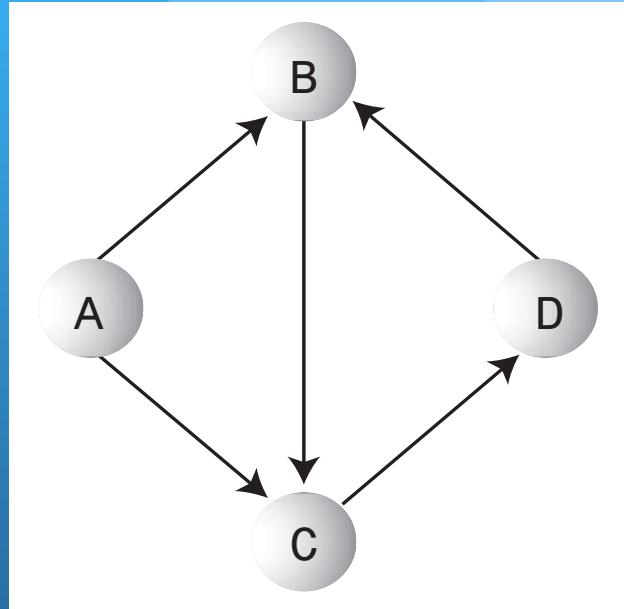
*La lista generada mostrará los vértices en orden topológico.*

El ordenamiento topológico **funciona** grafos **conexos o desconexos**.

Ejemplo: para modelar por ejemplos objetivos que no están relacionados.

# Ciclos y Árboles

- Los grafos que contienen ciclos no puede ser manejado por el algoritmo de ordenamientos topológico.
- Un ciclo sería por ejemplo: B C D B
- Los ciclos ayudan a modelar situaciones “trampa”.  
Ej.: si los vértices del grafo fueran asignaturas.
- Cuando un grafo no contiene ciclos se dice que es un árbol.
- Es fácil darse cuenta que un grafo no dirigido tiene ciclos:  
Si el grafo tiene N vértices y el número de aristas es mayor que N-1, entonces tiene ciclos.
- Un grafo dirigido sin ciclos se denomina: grafo dirigido acíclico (DAG).



# Código para Ordenamiento Topológico



```

public void topo()           // topological sort
{
    int orig_nVerts = nVerts; // remember how many verts

    while(nVerts > 0)         // while vertices remain,
    {
        // get a vertex with no successors, or -1
        int currentVertex = noSuccessors();
        if(currentVertex == -1)    // must be a cycle
        {
            System.out.println("ERROR: Graph has cycles");
            return;
        }

        // insert vertex label in sorted array (start at end)
        sortedArray[nVerts-1] = vertexList[currentVertex].label;

        deleteVertex(currentVertex); // delete vertex
    } // end while

    // vertices all gone; display sortedArray
    System.out.print("Topologically sorted order: ");
    for(int j=0; j<orig_nVerts; j++)
        System.out.print( sortedArray[j] );
    System.out.println("");
} // end topo

```

El código java **topo()** ejecuta el ordenamiento topológico para un grafo cualquiera.

El trabajo ocurre en el **ciclo while** que se ejecuta hasta que el número de vértices sea cero.

## Pasos:

1. Llama a **noSuccessors()** para encontrar los vértices que no tienen sucesores.
2. Si lo encuentra, lo coloca el valor del vértice al final del arreglo **sortedArray[]** y elimina el vértice del grafo.
3. Si no encuentra un vértice , el grafo **debe tener un ciclo**.

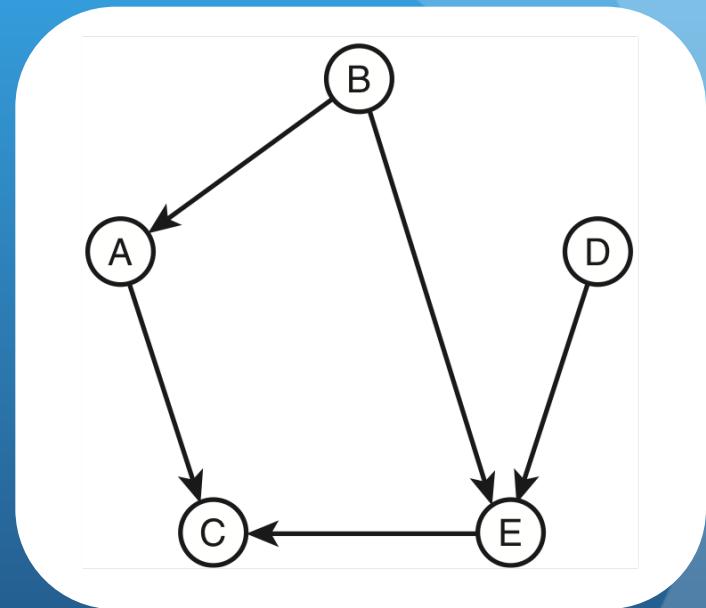
# Conectividad en Grafos Dirigidos

- En los grafos no dirigidos se pueden encontrar todos los vértices que están conectados utilizando la búsqueda en profundidad o la búsqueda en amplitud.
- En un grafo dirigido el proceso es un poco más complicado (figura). Si se parte en A no puedo alcanzar todos los otros vértices.
- Si realizamos una búsqueda en profundidad de cada nodo en secuencia, la salida sería la siguiente:

- AC
- BACE
- C
- DEC
- EC

{}

Esta es la Tabla de Conectividad  
del grafo

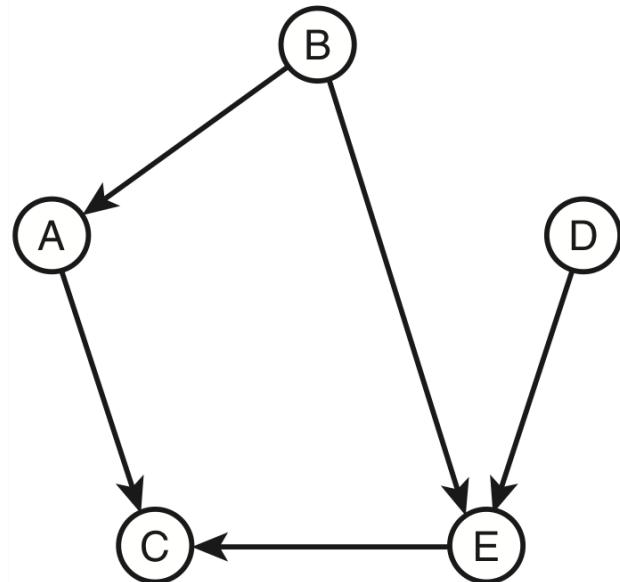


# Algoritmo de Warshall

- En algunos casos es importante saber rápidamente si es posible llegar desde un vértice a otro.  
Ej.: Queremos saber si es posible tomar de Temuco a Ushuaia.  
¿Es este viaje posible?
- Habría que revisar la Tabla de Conectividad, verificando todas las entradas para una fila determinada.  
Tiempo:  $O(N)$  , N es el promedio de nodos alcanzables desde el vértice.
- ¿Existe la posibilidad de calcular esto último en  $O(1)$  ?
- Si, modificando sistemáticamente la matriz Matriz de Adyacencia, y dando origen a lo que se denomina el Cierre Transitivo del grafo original.

# Algoritmo de Warshall

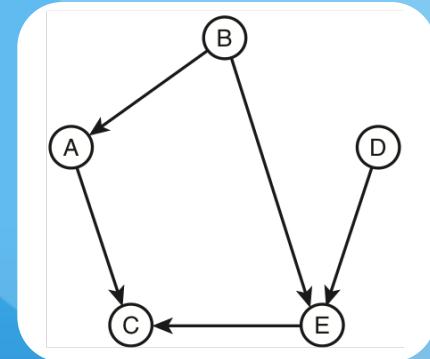
- El algoritmo que pasa de la Matriz de Adyacencia a al Grafo de Cierre Transitivo se basa en el siguiente principio:
- *Si puedo llegar del vértice L a M, y del vértice M a N. Entonces es posible llegar de L a N.*



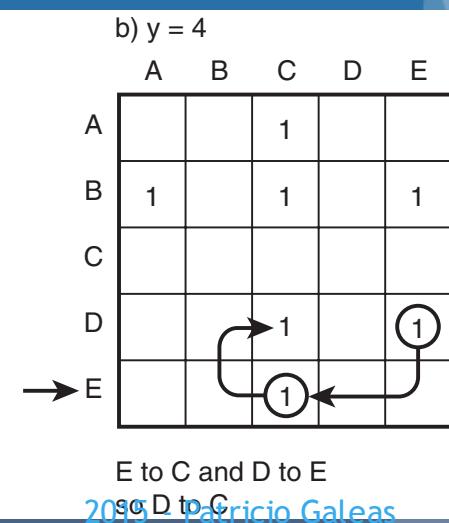
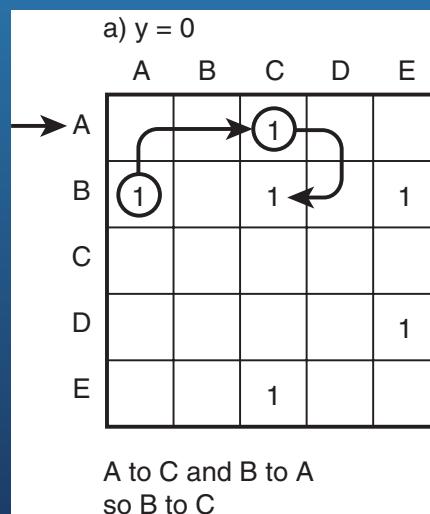
	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

# Algoritmo de Warshall

- Partamos en la fila A: el 1 en la columna C indica que hay una arista de A a C.
- Si existiera algún vértice (X) conectado con A podríamos decir que X también estaría conectado con C.
- Entonces, examinando la columna A nos damos cuenta que hay una arista entre B a A.
- Es decir, podemos llegar de B a C en dos pasos.
- Para guardar esta información colocamos un 1 en la intersección de B y C.
- Se repite lo mismo para la fila E. Y con el resultado podríamos construir el grafo de cierre transitivo.



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



# Resumen



- Los grafos consisten en vértices conectados a través de aristas.
- Los grafos pueden representar muchas situaciones de la vida real: rutas de líneas aéreas, circuitos electrónicos, etc.
- Los algoritmos de búsqueda de grafos permiten visitar sus vértices de una manera sistemática. Siendo esta búsqueda la base para una serie de otros procedimientos.
- Los dos principales algoritmos de búsqueda son: la Búsqueda en Profundidad (DFS) y la Búsqueda en Amplitud (BFS).
- La Búsqueda en Profundidad se basa en una Pila y la Búsqueda en Amplitud se basa en una Cola.
- El Árbol de Recubrimiento Mínimo (MST), consiste en el número mínimo de aristas para conectar todos los vértices de un grafo.
- El Árbol de Recubrimiento Mínimo puede ser generado a través con una versión levemente modificada del algoritmo de Búsqueda en Profundidad.
- En un Grafo Dirigido, las aristas tienen dirección (flecha).
- Un Algoritmo de Ordenamiento Topológico crea una lista de vértices organizados de manera que si el vértice A precede al vértice B en la lista, existe una ruta entre A y B.
- El Ordenamiento Topológico sólo puede ser llevado a cabo en grafos que no contienen ciclos.
- El Algoritmo de Warshall, permite determinar si existe una conexión desde un vértice a otro.

# Experimentos

- Generar una matriz de adyacencia para un grafo de 5 vértices con ceros y unos en forma aleatoria (no se preocupen de la simetría). Luego dibujen el grafo asociado a esta matriz. Verifiquen los resultados con el applet.
- Modifique el programa bfs.java para encontrar el árbol de recubrimiento mínimo (MST) utilizando la Búsqueda en Amplitud en vez de la Búsqueda en Profundidad, mostrada en mst.java. En el main() cree un grafo de 9 vértices y 12 aristas y encuentre su MST.

