Estructuras de Datos

Grafos
Introducción

ChangeLog

22/4/2025
- Versión inicial adaptada de la versión con cursores.

6/5/2025
- Añadida interfaz vertex(label:Int) a Graph.
- Renombradas interfaces getData por item() en Vertex y Edge.
- Renombradas interfaces setData() por setItem() en Vertex y Edge.
- Renombrada interfaz getLabel():Int por label() en Vertex.

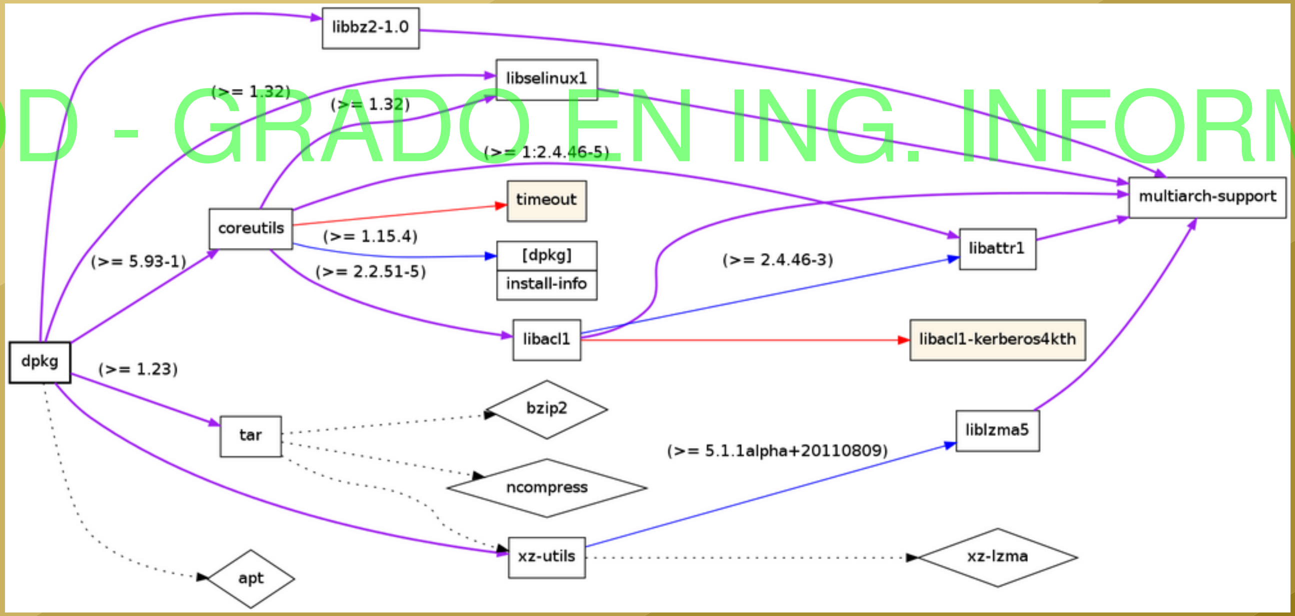
7/5/2025
- Añadido definición de orden y tamaño de un grafo.

9/5/2025
- Quitada interfaz findVertex(v:V):Vertex para simplificar (usar findFirst())
- Añadida post-c a addVertex(v:V) para asegurar que la etiqueta es única.
- Mejoradas algunas post-c de otros métodos.
- Usar un grafo no dirigido como ejemplo en el diseño con listas de incidencia.

13/5/2025
- Mejorado diseño con matriz de adyacencia.

Motivación

- Modelar dependencias de un paquete en una distribución Línux.



`apt install dpkg`

¿Podremos usar un árbol?

Grafo

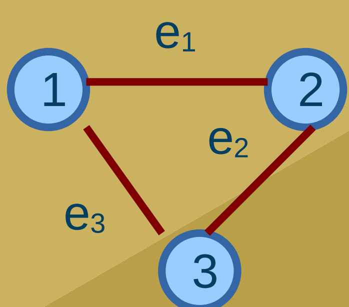
- Contenidos.
  - Introducción y definiciones.
  - Tipos de grafos.
  - Especificación.
  - Diseño.
  - Resumen final.
  - Referencias.

Introducción

- Representa la relación más general entre items: M:N.
- Aplicaciones:
  - Hay muchas situaciones cotidianas que se pueden modelar con grafos.
  - Aplicaciones en la ingeniería:
    - Planificación y gestión de proyectos/tareas.
    - Control de flujo en redes (de comunicaciones, aguas, eléctricas, …)
    - Modelar conocimiento: modelo Entidad-Relación, interacciones sociales, sistemas de recomendación...
  - Aplicaciones matemáticas:
    - Cálculo de caminos mínimos.
    - Recorridos óptimos.
    - Resolución de problemas topológicos en un red.
    - Estimar probabilidades.

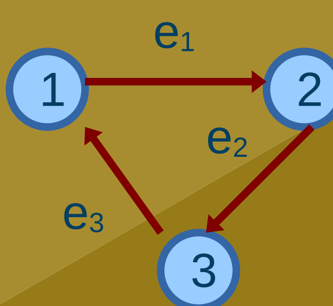
Definición

- El grafo como entidad matemática.
  - un grafo se define por dos conjuntos {V, E} y un mapa {f}.
    - V es el conjunto de vértices (nodos). N=|V| es el **orden** del grafo.
    - E es el conjunto de lados. M=|E| es el **tamaño** del grafo.
    - f es un mapa {E -> VxV} que puede ser “ordenado” o “no ordenado”.
  - Vértices y lados pueden tener atributos.



En un grafo no dirigido el mapa es “no ordenado” (1,3) = (3,1)

V:{1, 2, 3}
E:{e1, e2, e3}
f:{e1:(1,2), e2:(2,3), e3:(3,1)}

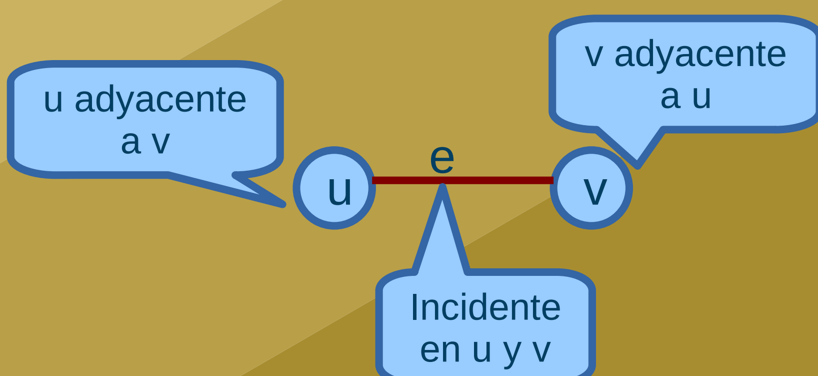


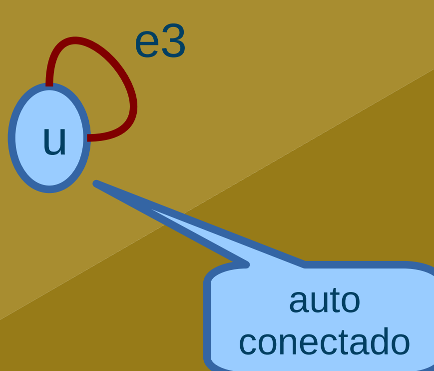
En un grafo dirigido el mapa es ordenado (1,3) <> (3,1)

Definición

- Definiciones en grafos no dirigidos.
  - Sea e:(u,v) un lado, se dice que los vértices “u”, “v” son sus **extremos**.
  - Si existe un lado e:(u,v), se dice que los vértices “u”, “v” **son adyacentes**.
  - Sea e:(u,v) un lado, se dice que el lado “e” **es incidente** en los vértices “u” y “v”.

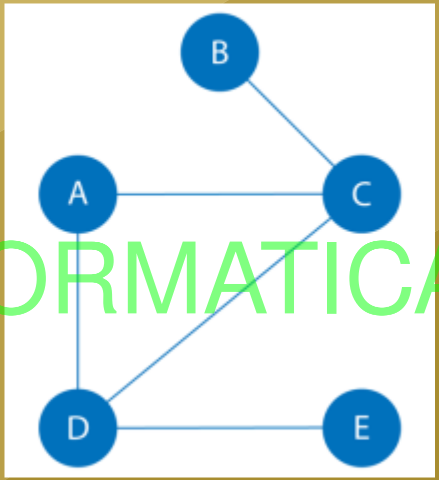
  - Sean e1:(u, v) y e2:(v, t), se dice que los lados e1 y e2 son adyacentes (comparten “v” como extremo).
  - Puede existir e:(u, u), es decir “u” está **auto-conectado**.





Definición

- Definiciones en grafos no dirigidos.
  - Grado de un vértice.
  - Camino.
  - Longitud de un camino.
  - Camino elemental y simple.
  - Ciclo.
  - Ciclo elemental y simple.
  - Conectividad.

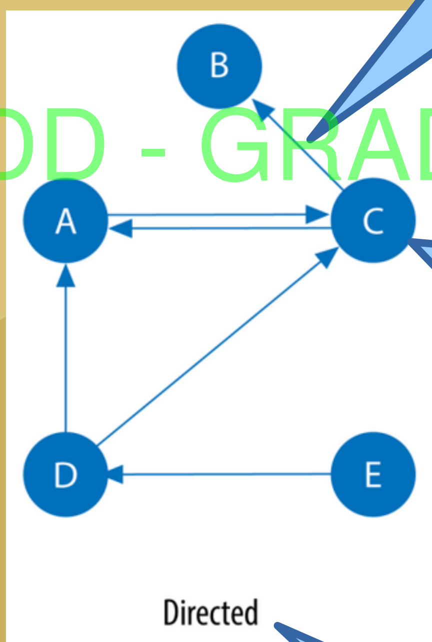


{B,C,D,E} camino elemental de L=3
{B,C,A,D,C} camino simple.
{A,C,D,A} ciclo elemental.
A y B están conectados por el camino {A, C, B}

Tipos de grafos

- Grafos dirigidos.

Extremo Origen: C
Extremo Destino: B



Grado salida: 2
Grado entrada: 2
Grado total: 4

Camino dirigido:
{E,D,A,C,B}

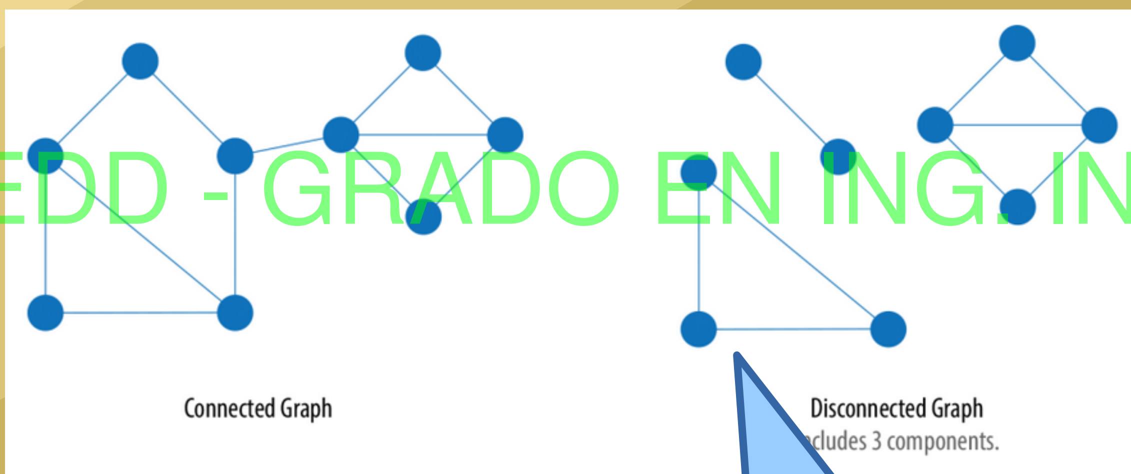
mapeo ordenado
(E,D) ≢ (D,E)

¿Son E y D adyacentes?:

- Suponer que el lado es no dirigido: E y D son adyacentes.
- Con criterio de entrada: D es adyacente a E, pero no al revés.
- Con criterio de salida: E es adyacente a D, pero no al revés.
- Con criterio de simetría: sólo si existen los lados (E,D) y (D,E), decimos que E y D son adyacentes.

Tipos de grafos

- Conectado / no conectado.



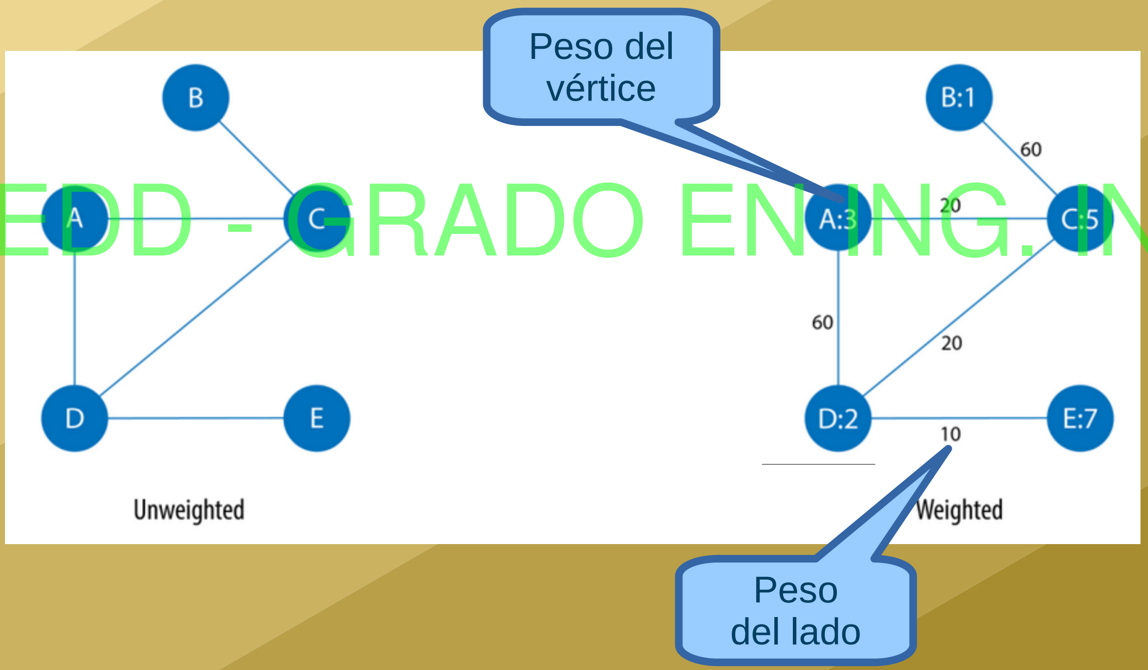
Componente conexa ("cluster")

En grafos dirigidos tenemos:

- Conectividad débil.
- Semi-conectividad.
- Conectividad fuerte.

Tipos de grafos

- Ponderado / no ponderado.



$$\begin{bmatrix} \infty & \infty & 20 & 60 & \infty \\ \infty & \infty & 60 & \infty & \infty \\ 20 & 60 & \infty & 20 & \infty \\ 60 & \infty & 20 & \infty & 10 \\ \infty & \infty & \infty & 10 & \infty \end{bmatrix}$$

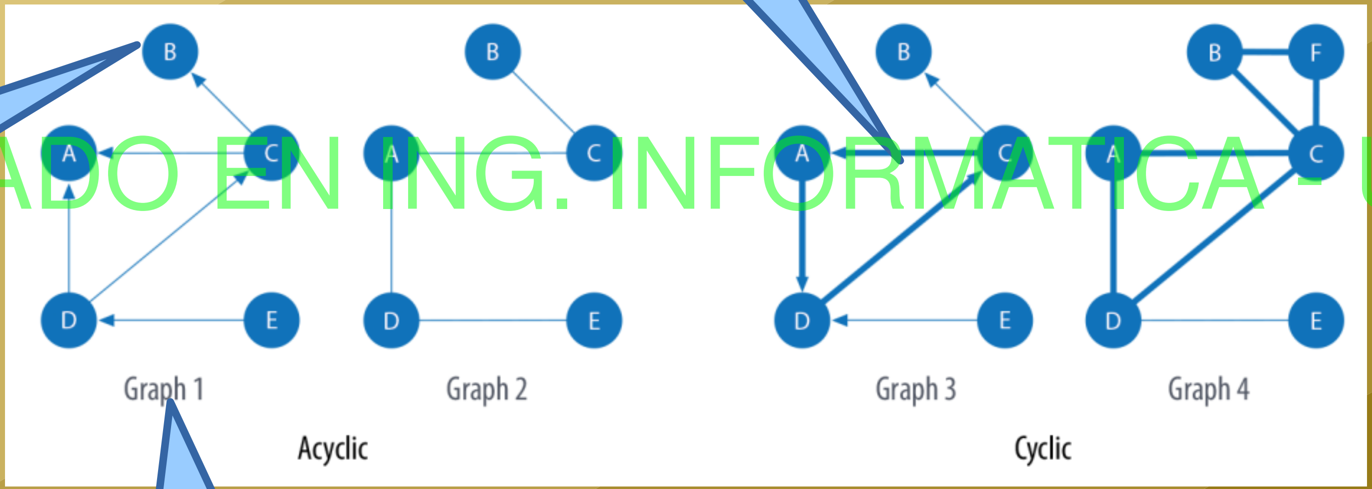
Matriz de pesos
(lados)

Tipos de grafos

- Con ciclos / sin ciclos.

Ciclo

Hoja
Grado
salida=0

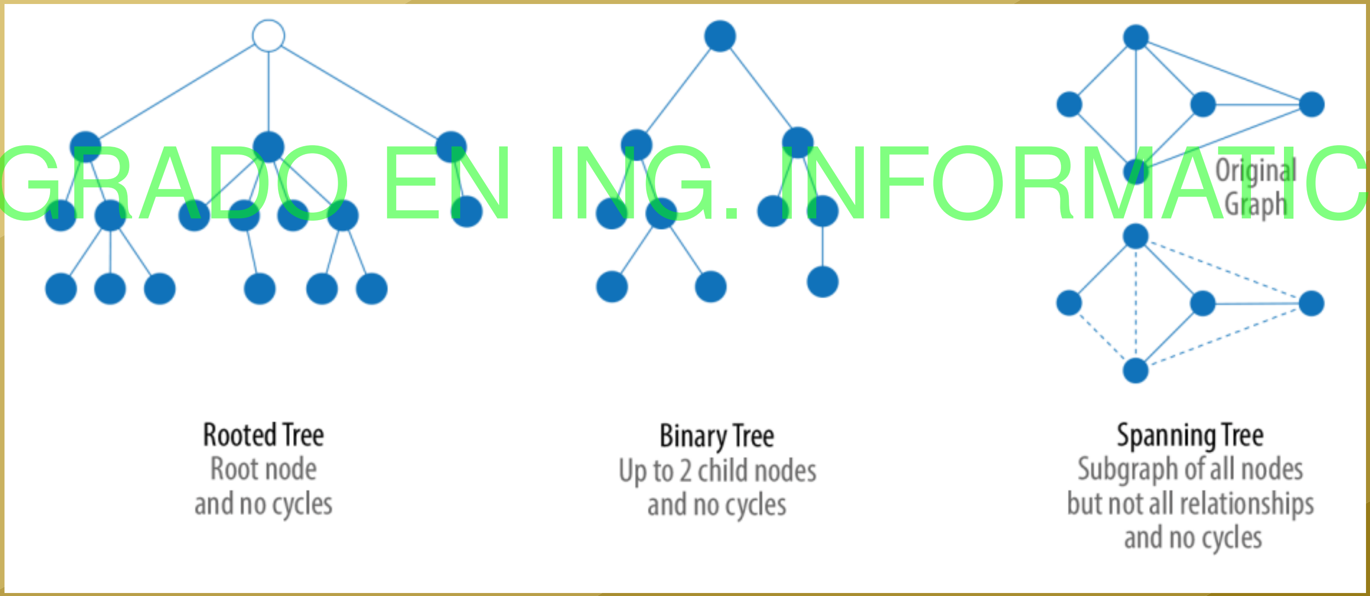


DAG: Directed
Acyclic Graph.

¿Qué tipo de grafo será
una lista?

Tipos de grafos

- Árboles.



¿Qué será el "Minimum Spanning Tree"?

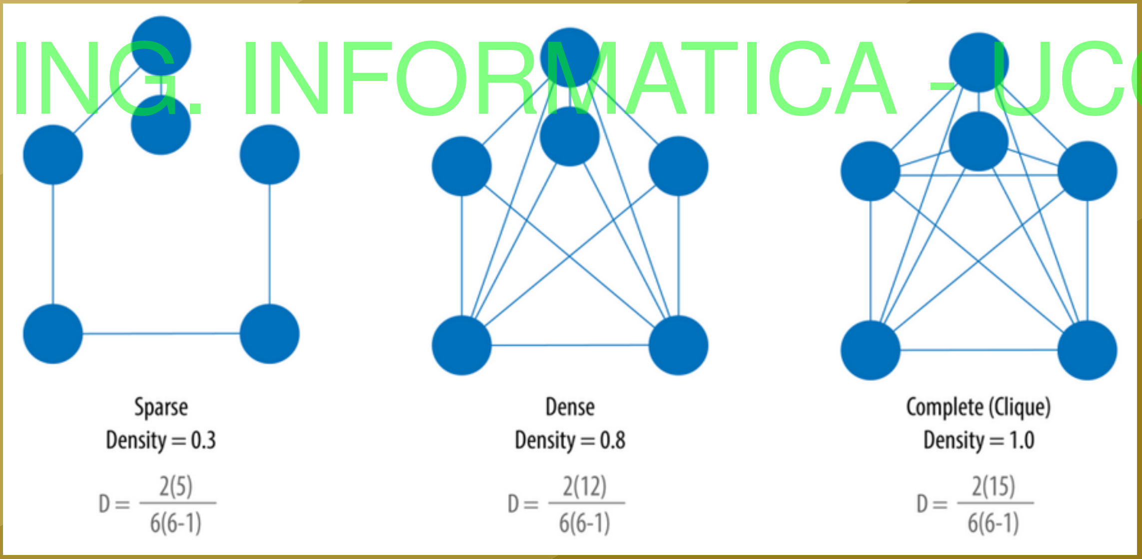
Tipos de grafos

- Denso / no denso.

Densidad de un grafo

	No auto conectado	Auto conectado
No dirigido	$\frac{2M}{N(N-1)}$	$\frac{2M}{N(N+1)}$
Dirigido	$\frac{M}{N(N-1)}$	$\frac{M}{N^2}$

Grafos no dirigidos sin auto conexiones.

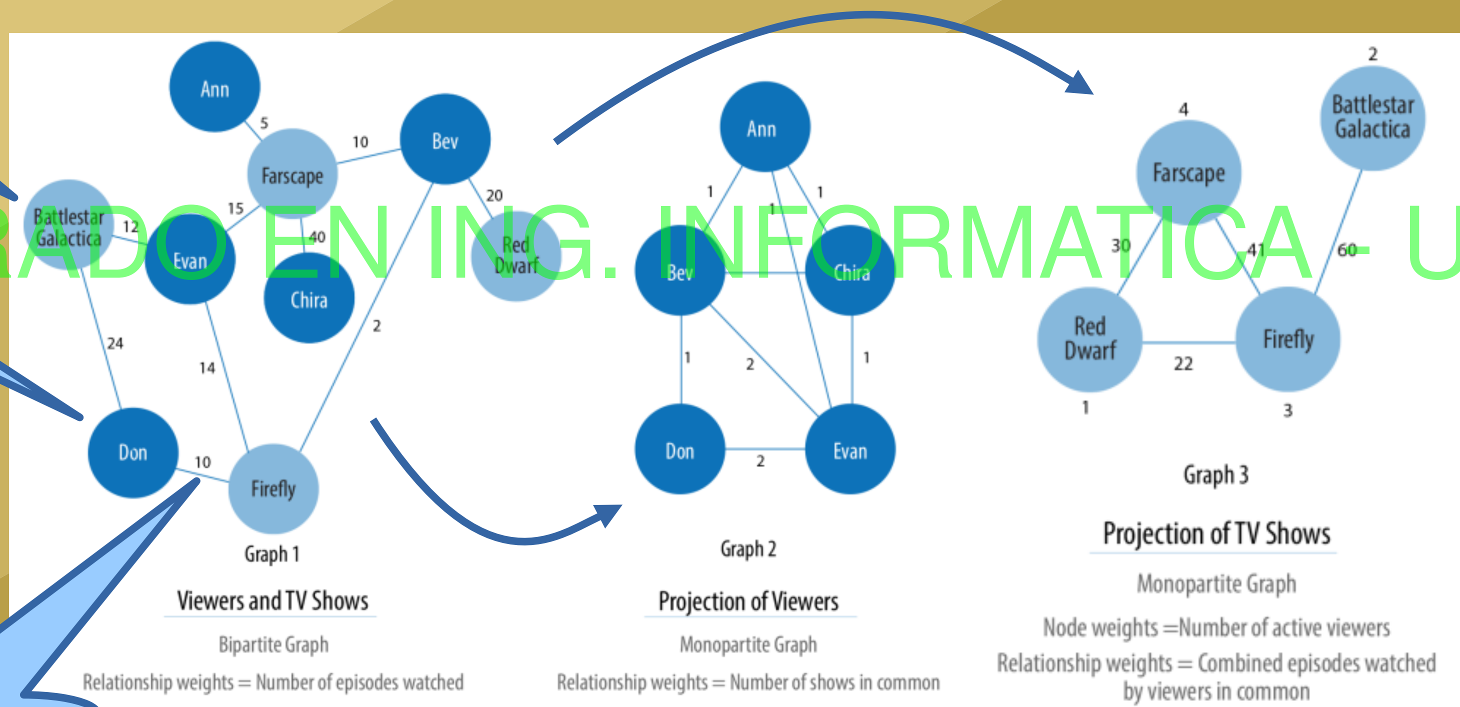


Tipos de grafos

- Mono partición / Bi partición / K partición.

Series

Espectadores



“Don” ha visto 10 capítulos de “Firefly”

Especificación

- TAD Vértice y TAD Lado.

ADT Vertex[V]

Observers:
- **item**():V // gets the data.
- **label**():Int // gets the vertex label.
  - post-c: the label is unique for this vertex in the graph.
- **isVisited**():Bool //Has the vertex been visited?

Modifiers:
- **setItem**(d:V) // set the data.
  - post-c: getData()==d
- **setVisited**(state:Bool) //Set visited state.
  - post-c: isVisited() iff state.

ADT Edge[E]

Observers:
- **item**():E // gets edge's data.
- **has**(u:Vertex):Bool // Is vertex u an end of this edge.
- **other**(u:Vertex):Vertex // the vertex other than u.
  - pre-c: has(u).
  - post-c: has(retVal) and other(retVal) = u
- **first**():Vertex //get the start vertex in directed graphs or one of the ends in undirected graphs.
  - post-c: other(retVal) = second()
- **second**():Vertex //get the last vertex in directed graphs or one of the ends in undirected graphs.
  - post-c: other(second()) = first())
- **isVisited**():Bool //Has the edge been visited?

Modifiers:
- **setItem**(d:E) // set the edge's data.
  - post-c: getData() == d
- **setVisited**(state:Bool) //Set visited state.
  - post-c: isVisited() iff state.

Especificación

- Iteradores de vértices y lados.

ADT VertexIterator[V]

Observers:

- **isValid**():Bool // is a valid iterator?
- **get**():Vertex // gets a reference to the vertex.
  - pre-c: isValid()
- **isEqual**(other:VertexIterator):Bool // is this equal to other?

Modifiers:

- **gotoNext**() // go to the next iterator position.
  - pre-c: isValid()

ADT EdgeIterator[V,E]

Observers:

- **isValid**():Bool // is a valid iterator?
- **get**():Edge // gets a reference to the edge.
  - pre-c: isValid()
- **isEqual**(other:EdgeIterator):Bool // is “this” equal to other?

Modifiers:

- **gotoNext**() // go to the next iterator position.
  - pre-c: isValid()

Especificación

- TAD Grafo.

```
ADT Graph[V,E]

Creators:
• make(isDirected:bool) //create a graph.
  • post-c: isEmpty()
  • post-c: isDirected()=isDirected

Observers:
• isEmpty():Bool //Is the graph empty?
• isDirected():Bool //Is the graph a directed one?.
• has(v:Vertex):Bool //Is v a vertex of this graph?.
• has(e:Edge):Bool //Is e an edge of this graph?.
  • pre-c: has(e.first()) And has(e.second())
• vertex(label:Int):Vertex //Get the vertex given its label.
  • post-c: retV=Void Or (g.has(retV) And retV.label()=label)
• edge(u,v:Vertex):Edge //get the edge linking u with v.
  • pre-c: has(u) And has(v)
  • post-c: retV=Void Or retV.has(u) And retV.other(u)=v
  • post-c: retV=Void Or (Not isDirected() OR (retV.first()=u And
    retV.second()=v))
• isAdjacent(u,v:Vertex):Bool // Is there any edge linking u with v?
  • pre-c: has(u) And has(v)
  • post-c: Not retV Or has(g.edge(u,v))
```

Especificación

- Especificación: TAD Graph.

```
ADT Graph[V,E]

Observers: //Vertex iterators.
• vertexBegin(): VertexIterator[V] //Get an interator points to the
  first vertex.
  • Post-c: retV=vertexEnd() Or retV.isValid()
• vertexEnd(): VertexIterator[V] // Get an interator at the end of the
  vertices.
  • post-c: Not retV.isValid()
• getIterator(v:Vertex[V]):VertexIterator[V] //Get an iterator points
  to vertex v.
  • Pre-c: has(v)
  • Post-c: retV.get().label()=v.label()
• findFirst(item:V): VertexIterator[V] //Get an iterator points to the
  first occurence of a vertex u with u.item()=item.
  • post-c: retV=vertexEnd() Or retV.get().item()=item
```

Especificación

- Especificación: TAD Graph.

```
ADT Graph[V,E]

Observers: //Edge iterators.
• edgeBegin(u:VertexIterator[V]): EdgeIterator[V,E] //Get an interator
  points to the first edge incident in u.
• edgeEnd(u:VertexIterator[V]): EdgeIterator[V,E] // Get an interator
  at the end of the edges incident in u.
• findFirst(u:VertexIterator[V], v:V): EdgeIterator[V,E] //Get an
  iterator points to the first occurence of an edge(u,s) with vertex
  s.item()=v.
```

Especificación

- Especificación. TAD Graph.

Modifiers:

- **addVertex**(item:V):Vertex //create a new vertex.
  - post-c: has(retV)
  - post-c: retV.item()=item
  - post-c: Not retV.isVisited()
  - post-c: for each u<>retV: has(u) And retV.label()<>u.label()
- **addEdge**(u,v:Vertex, d:E):Edge //create new edge to link u with v.
  - pre-c: has(u) And has(v)
  - post-c: retV.item()=e
  - post-c: Not retV.isVisited()
  - post-c: retV.has(u) And retV.other(u)=v
  - post-c: retV.has(v) And retV.other(v)=u
  - post-c: retV.first()=u And retV.second()=v
  - post-c: is_adjacent(u, v)
  - post-c: isDirected() || is_adjacent(v, u)
- **removeVertex**(v:Vertex) //remove a vertex and all the edges incidents in it.
  - pre-c: has(v)
  - post-c: not has(v)
- **removeEdge**(e:Edge) //remove and edge.
  - pre-c: has(e)
  - post-c: Not has(e) And Not isAdjacent(e.first(), e.second())
- **reset**(st:Bool) //set the visited state for all the vertexes and edges of the graph.

Especificación

- Ejemplo de uso.

```
Algorithm computeGraphDensity(g:Graph[V,E],
                              autoConnected:Bool):Float

End.
```

Especificación

- Ejemplo de uso.

```
Algorithm computeGraphDensity(g:Graph[V,E],
                              autoConnected:Bool):Float
Var
   N, M : Integer
   v_it : VertexIterator[V]
   e_it : EdgeIterator[V,E]
Begin
   N ← 0
   M ← 0
   v_it ← g.verticesBegin()
   While v_it <> g.verticesEnd() Do
      N ← N + 1
      e_it ← g.edgesBegin(v_it)
      While e_it <> g.edgesEnd(v_it) Do
         M ← M + 1
         e_it.gotoNext()
      End-While
      v_it.gotoNext()
   End-While
   If Not autoConnected Then
      N ← N*(N-1)
   Else If g.isDirected() Then
      N ← N*N
   Else
      N ← N*(N+1)
   Return  M / N //Si el grafo es no dirigido, ya hemos
                 //contado un lado (u,v) dos veces.
End.
```

Diseño

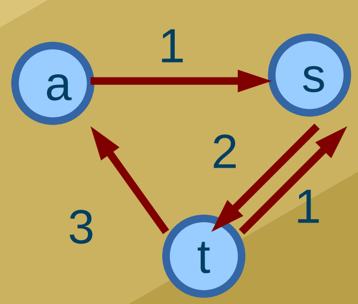
- Usando una matriz de adyacencia.
  - Ventaja: isAdjacent(u,v) O(1).
  - Inconveniente: gasto de memoria O(N²).

Vertex[V]

label_:Int
item_:V
isVisited:Bool

Edge[V,E]

first_:Vertex[V]
second_:Vertex[V]
a_:Array[E]



Graph[V,E]

v_:Array[V]
a_:Matrix[E]
isDirected_:Bool

0	1	2	N-1
a	s	t	...

second.label_

$$\begin{array}{c} \text{first.label\_} \end{array} \begin{bmatrix} \infty & 1 & \infty & \cdots \\ \infty & \infty & 2 & \cdots \\ 3 & 1 & \infty & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

NxN

En grafos ponderados almacena los pesos de los lados.

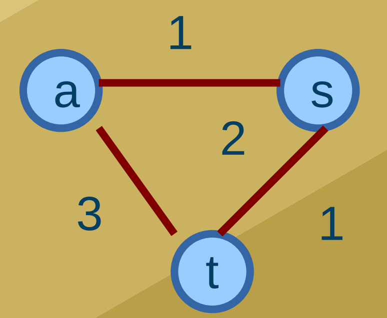
Se suele usar “infinito” para indicar no conexión.

Utilizamos la etiqueta cómo índice para acceder al vector/matriz.

Si el grafo es “no dirigido” sólo usamos el triángulo superior y tendremos el lado (u,v) si u.label()<v.label() o, en caso contrario, el lado (v,u)

Diseño

- Usando listas de incidencia.
  - Ventajas: espacio O(N+M).
  - Inconveniente: isAdjacent(u,v) O(N).



Si el grafo es “dirigido” el lado (u, v) se considera incidente sólo en “u”.

Graph[V,E]

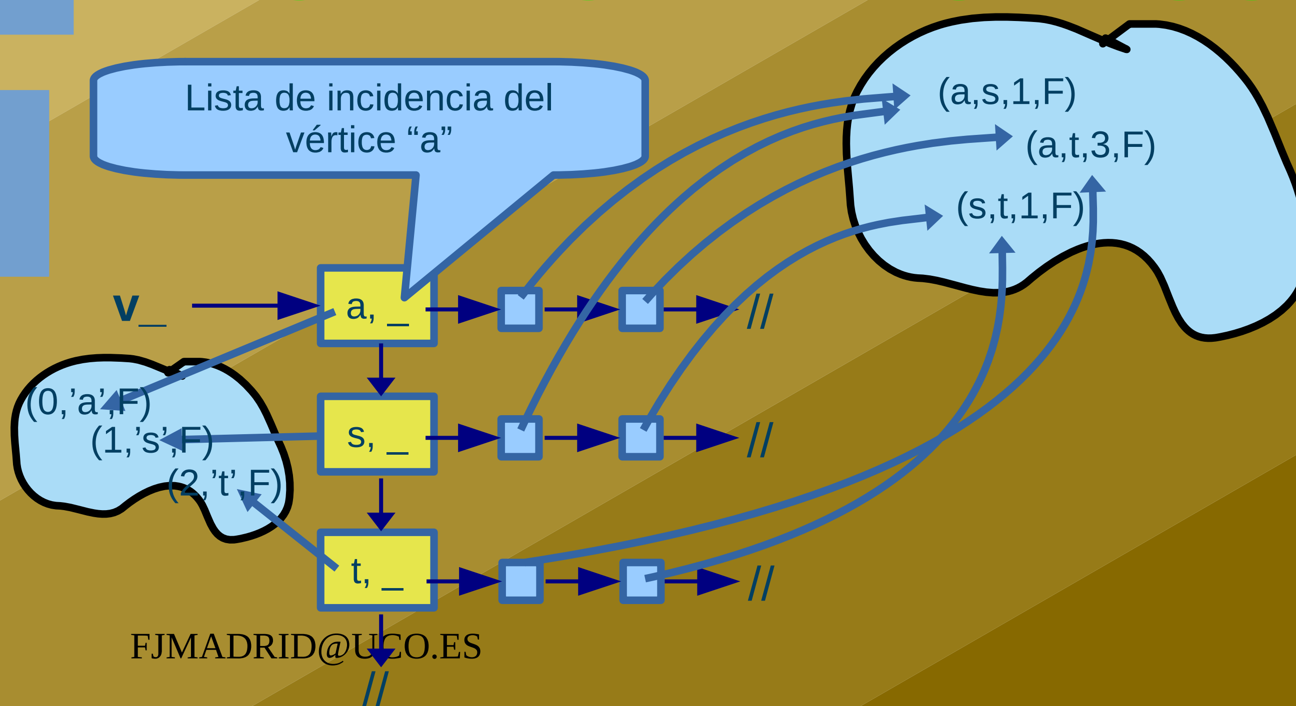
v_:List[Pair[Vertex[V], List[Edge[V,E]]]]
isDirected_:Bool

Vertex[V]

label_:Int
item_:V
isVisited_:Bool

Edge[V,E]

item_:E
first_:Vertex[V]
second_Vertex[V]
isVisited_:Bool



Diseño

- Comparación de diseños.

	Diseño	
Criterio de comparación	**Matriz de adyacencia**	**Listas de incidencia**
¿Es “u” adyacente a “v”?	O(1)	O(N)
Todos los lados incidentes en “u”	O(N)	O(grado de “u”)
Almacenamiento	$O(N^2)$	O(N+M)
Indicado para	Grafos Densos	Grafos no Densos

Un grafo se dice que es denso cuando su densidad > 0.5

Grafo

- Resumiendo.
  - Modelan relaciones M-N.
  - En un grafo “dirigido” (u,v)<>(v,u).
  - En un grafo “no dirigido” (u,v) = (v,u).
  - Un grafo es denso cuando su densidad > 0.5.
  - Dos diseños:
    - Con matriz de adyacencia indicada para grafos densos.
    - Con listas de incidencia indicada para grafos poco densos.

Grafo

- Lecturas recomendadas.
  - Cap. 14 de “Estructuras de Datos”, A. Carmona y otros. U. de Córdoba. 1999.
  - Cap. 15 de “Data structures and software develpment in an object oriented domain”,Tremblay J.P. y Cheston, G.A. Prentice-Hall, 2001.
  - Wikipedia:
    - en.wikipedia.org/wiki/Graph_(discrete_mathematics)
    - en.wikipedia.org/wiki/Glossary_of_graph_theory_terms
    - en.wikipedia.org/wiki/Graph_(abstract_data_type)
    - www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/