Estructuras de Datos

Grafos

Contenidos

- Introducción.
- Especificación.
- Implementaciones.
- Búsqueda de caminos.
- Recorridos.

Introducción

- Represente relaciones muchos-a-muchos.
- Estructura de datos más general de todas.
- Aplicaciones:
  - Hay muchas situaciones cotidianas que se pueden modelar con grafos.
  - Aplicaciones en la ingeniería:
    - Planificación y gestión de proyectos.
    - Control de flujo en redes (de agua, eléctrica, …)
  - Aplicaciones matemáticas:
    - Cálculo de caminos mínimos.
    - Recorridos en una red.
    - Resolución de problemas topológicos en un red.

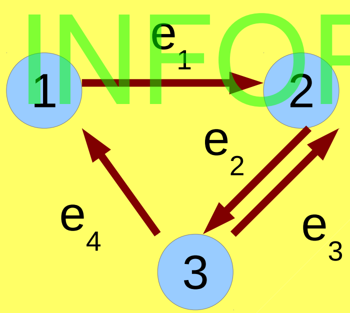
Introducción

- Definición:
  - Sea G:{V, E,*f*}
    - V es el conjunto de nodos o vértices.
    - E es el conjunto de lados.
    - *f*: E -> VxV (un mapeo de lados a pares de nodos: $f(e_i)=(v_j,v_k)$)
  - Varios tipos de grafos:
    - Grafo dirigido: f es un mapeo ordenado $(v_j,v_k)<>(v_k,v_j)$
    - Grafo no dirigido: f es un mapeo no ordenado $(v_j,v_k)=(v_k,v_j)$
    - Grafo mixto.

Introducción

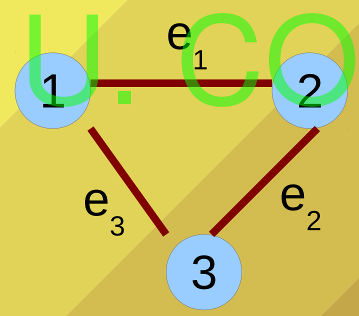
- Ejemplos:

Dirigido



V:{1, 2, 3}
E:{e1, e2, e3, e4}
f:{e1:(1,2), e2:(2,3), e3:(3,2), e4:(3,1)}

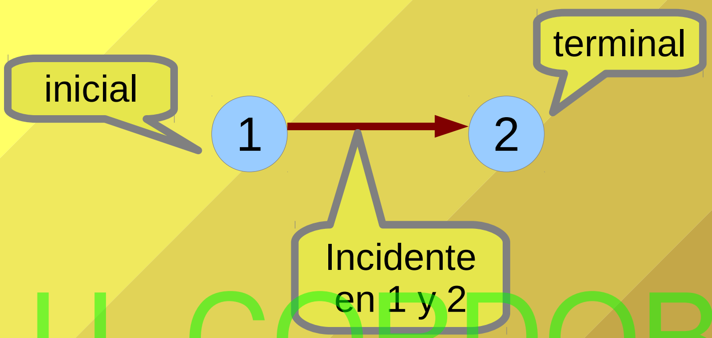
No dirigido

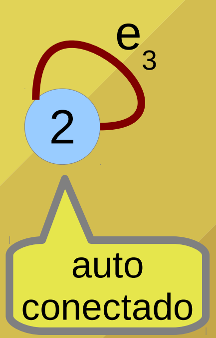


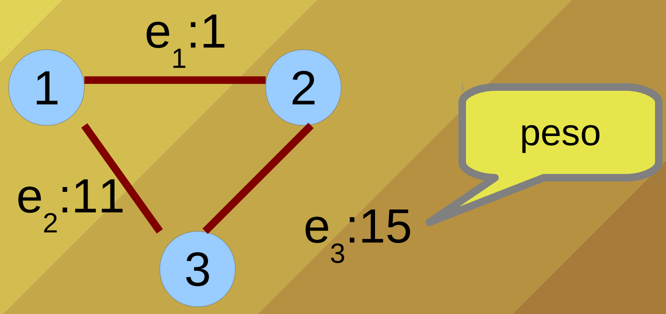
V:{1, 2, 3}
E:{e1, e2, e3}
f:{e1:(1,2), e2:(2,3), e3:(1,3)}
Ojo aquí: (1,2)=(2,1)

Introducción

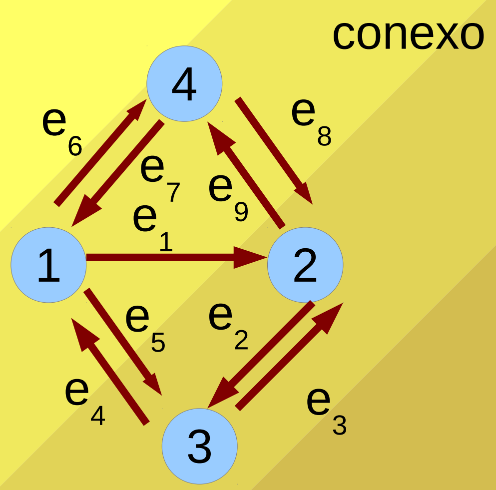
- Más definiciones:
  - Sea e:(u,v) un lado, se dice que los nodos u y v son **adyacentes**
  - Sea e:(u,v) un lado, se dice que el lado e es ***incidente*** en los nodos u, v.
  - Sea e:(u,v) un lado dirigido, u es el nodo ***origen/inicial*** y v el el nodo ***destino/terminal*** de e.
  - Puede existir e:(u,u), es decir u está ***auto-conectado, bucle o lazo***.
  - Puede asociarse un peso a cada lado e: ***grafo ponderado***.





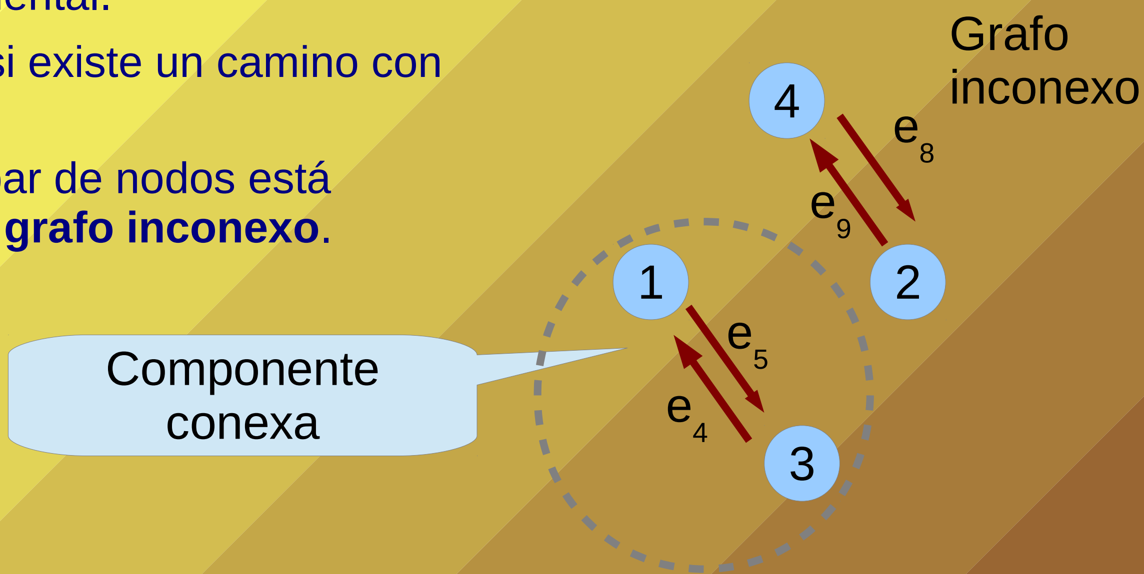


Introducción



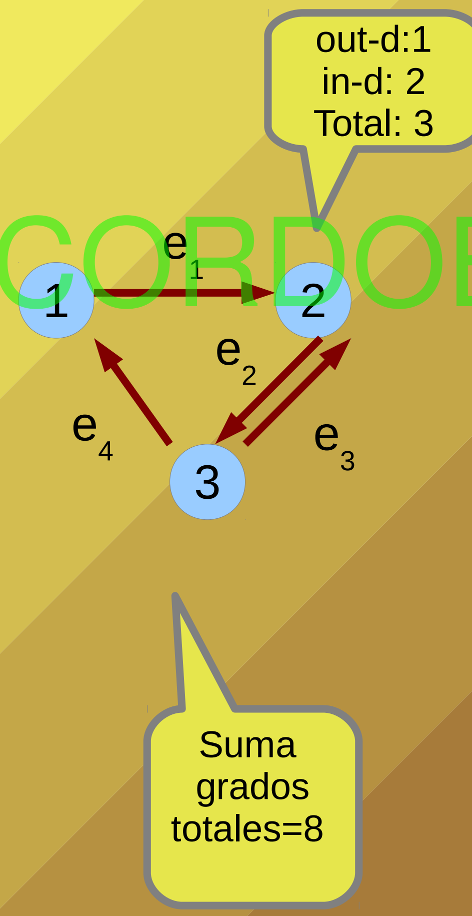
- Más definiciones:
  - **Camino** de v,u: sucesión de $\{v_1,v_2,...,v_k\}$ con v_1=v y v_k=u, y existe un lado para cada par (v_i,v_{i+1}).
  - **Longitud** del camino= número de **lados**.
  - **Camino simple**: todos los lados son distintos.
  - **Camino elemental**: todos los nodos son distintos (salvo los nodos inicial y final que pueden ser iguales).
  - **Ciclo**: un camino donde el nodo inicial y final es el mismo.
  - **Ciclo simple**: el camino es simple.
  - **Ciclo elemental**: el camino es elemental.
  - Dos nodos u, v están **conectados** si existe un camino con origen u y destino v.
  - **Grafo conectado o conexo**: todo par de nodos está conectado. En caso contrario es un **grafo inconexo**.

{1,2,3} camino elemental de L=2
{1,2,4,1,3} camino simple.
{1,2,3,1} ciclo elemental.
{1,2,3,2,4,1} ciclo simple.



Introducción

- Más definiciones:
  - **En un grafo dirigido:**
    - **<u>Nodo sucesor</u>**: n_i es sucesor de n_j si hay un camino desde n_j a n_i.
    - A la inversa n_j es un **<u>nodo predecesor</u>** de n_i.
    - **<u>Grado de salida</u>** de u: número de lados con u como nodo inicial.
    - **<u>Grado de entrada</u>** de u: número de lados con u como nodo terminal.
    - **<u>Grado total</u>** de u: grado salida + grado entrada.
  - En grafos no dirigidos hablamos sólo de grado de un nodo.
  - Suma de grados totales = 2*número de lados.



ADT Graph

- Dos conceptos están implicados:
  - Nodos.
  - Lados.

Vertex[G]

Observers:
- G **getData**() // gets the data.
- int **getLabel**() // gets the vertex label.
  - post-c: the label is unique for this vertex in the graph.

Mutators:
- **setData**(d:G) // set the data.

Edge[G]

Observers:
- G **getData**() // gets edge's data.
- Vertex **first**() //get the first vertex.
- Vertex **second**() //get the second vertex.
- bool **has**(u:Vertex) // Is vertex u an end of this edge.
- Vertex **other**(u:Vertex) // the vertex other than u.
  - pre-c: has(u).

Mutators:
- **setData**(d:G) // set the edge's data.

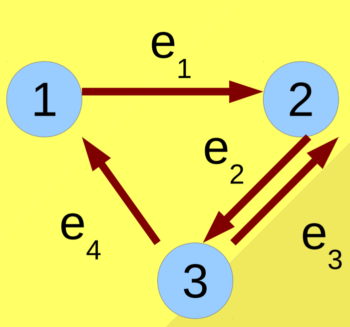
¿Por qué no hay constructores?

ADT Graph

ADT Graph[V,E]
Creators:
- **makeDirected**() //create a directed graph.
- **makeUndirected**() //create an undirected graph.

Observers:
- Integer **numVertexes**()
- Integer **numEdges**()
- Bool **isDirected**()
- Bool **isEmpty**()
- Bool **adjacent**(u,v:Vertex)// Is there any edge linking u,v?
  - pre-c: u,v are graph's vertexes.
- Bool **hasCurrVertex**() // true if the cursor points to a vertex.
- Vertex **currVertex**() //gets current vertex.
  - pre-c: hasCurrVertex()
- Bool **hasCurrEdge**() // true if the cursor points to a edge.
- Edge **currEdge**() //gets current edge.
  - pre-c: hasCurrEdge()

Mutators:
- **addVertex**(d:N) //create a new vertex.
- **addEdge**(u,v:Vertex, d:E) //insert edge to link u,v.
  - pre-c: u,v are graph's vertexes.
- **searchVertex**(d:N) //search vertex using data.
  - post-c: if it's found hasCurrVertex() and currVertex().getData()=d
- **goTo**(v:Vertex) //go to vertex.
  - pre-c: v is a graph's vertex.
  - post-c: currVertex().getData()=v.getdata()
- **searchEdge**(u,v:Vertex)//search the edge linking u,v.
  - pre-c: u,v are a graph's vertex.
  - post-c: if it's found hasCurrEdge() and currEdge().has(v) and currEdge.other(v)=u
- Vertex **beginVertex**()
- Vertex **nextVertex**()
- bool **afterEndVertex**()
- Edge **beginEdge**(v:Vertex)
- Edge **nextEdge**()
- bool **afterEndEdge**()

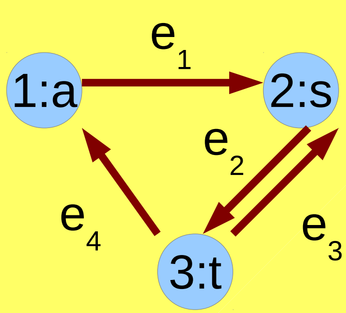


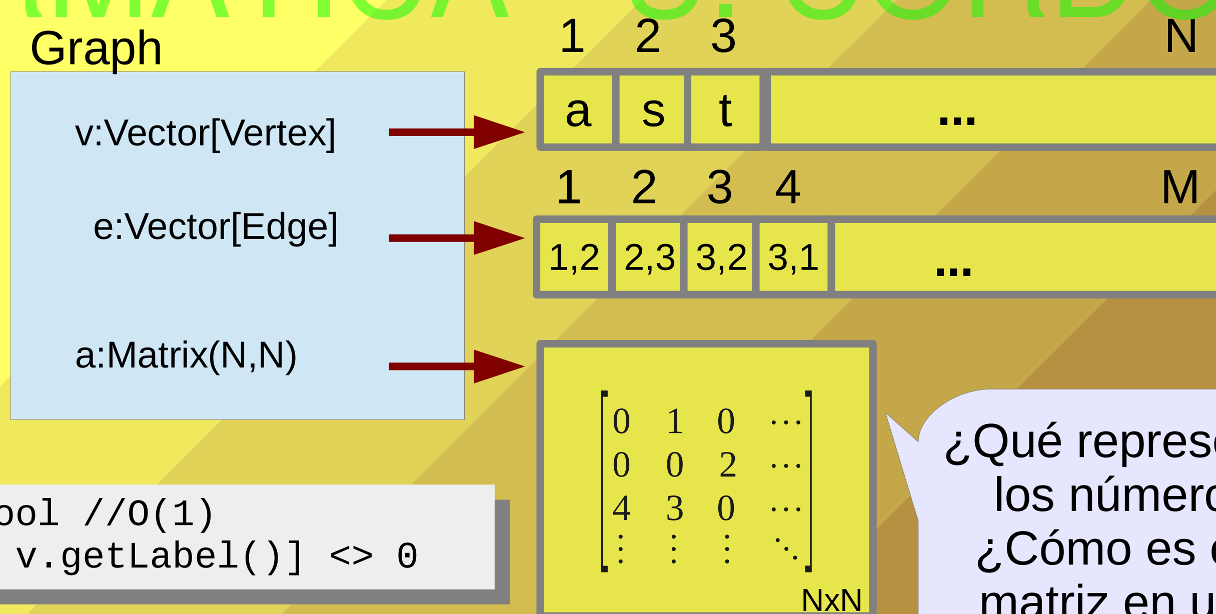

```
g.makeDirected()
g.addVertex(1)
g.addVertex(2)
g.addVertex(3)
g.searchVertex(1)
v1=g.currVertex()
g.searchVertex(2)
v2=g.currVertex()
g.addEdge(v1,v2)
```

```
g.searchVertex(3)
v3=g.currVertex()
g.addEdge(v3,v1)
g.addEdge(v3,v2)
g.addEdge(v2,v3)
```

Implementación

- Implementación basada en la matriz de adyacencia.
  - Ventaja optimiza la consulta adjacent(u,v).
  - Inconveniente: gasto de memoria. En grafos no dirigidos menos ¿por qué?



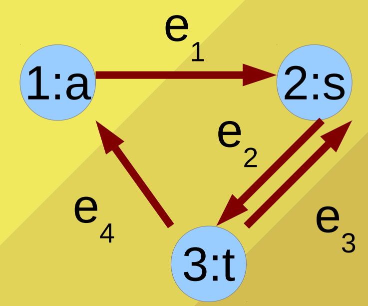



```
adjacent(u, v: Vertex):Bool //O(1)
  return a[u.getLabel(), v.getLabel()] <> 0
```

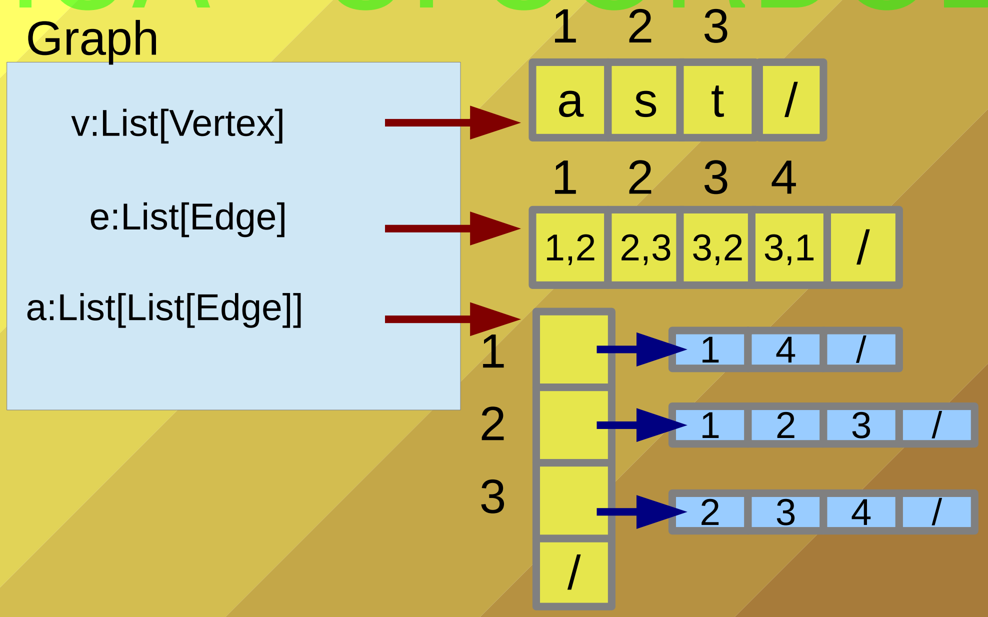
¿Qué representan los números? ¿Cómo es esta matriz en un g. no dirigido?

Implementación

- Basada en lista de adyacencias.
  - Ventajas:
    - optimiza encontrar los lados incidentes en un nodo v.
    - Reduce el espacio necesario O(n+m) (útil si m<<n^2).
  - Inconvenientes:
    - Más difícil determinar la adyacencia de dos nodos u,v.




```
adjacent(u, v: Vertex):Bool
  Found ← False
  a.goTo[u.getLabel()] //O(1)?
  l ← a.curr()
  l.begin()
  While not (l.pastEnd() or Found) Do
    If l.curr().data().second()=v or
      (not isDirected() and
        l.curr().data().has(v)) Then
      Found ← True
    Else
      l.next()
  End-While
  return Found
```

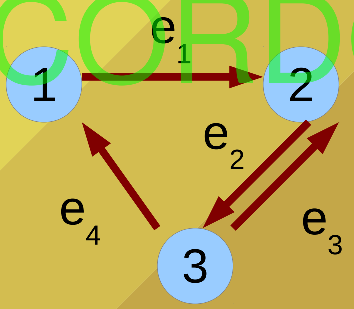


ADT Graph

- Ejemplo: cálculo de la matriz de adyacencia.

```
//Compute the adjacency matrix of a graph.
//We assume the vertex labels are
//consecutive indexes.

a.create(g.numVertex(),g.numVertex(), 0)
g.beginVertex()
While not g.afterEndVertex() Do
        u ← g.currentVertex()
        g.beginEdge(u)
        While not g.afterEndEdge() Do
                e ← g.currentEdge()
                a[u.getLabel(), e.other(u).getLabel()]←1
                g.nextEdge()
        End-While
        g.nextVertex()
End-While
```

Matriz de adyacencia: $a_{ij} = \begin{cases} 1, (v_i, v_j) \in E \\ 0, en\ otro\ caso \end{cases}$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Implementación

- Comparación de alternativas

	Representación	
Criterio de comparación	**Matriz de adyacencia**	**Listas de adyacencia**
adjacent(u,v)	O(1)	O(degree(u))
Todos los lados incidentes en u	O(N)	O(degree(u))
Almacenamiento	$O(N^2)$	O(N+M)