Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

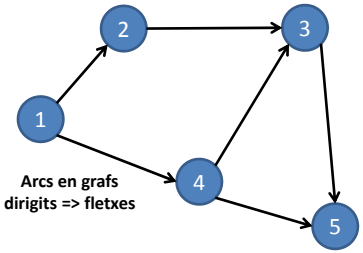
Grafs i algorismes per a la seva gestió

Programació Orientada a Objectes (POO)
ETSETB Telecom BCN

Jordi Perelló Muntan, curs 2018-2019

Concepte de graf

- Definició: "***Estructura*** *de dades no lineal formada per un* ***conjunt de Nodes*** *i un* ***conjunt d'Arcs****, on* ***cada Arc interconnecta un parell (ordenat o no) de Nodes*** *que mantenen una certa relació entre ells*"
- En aquesta assignatura **assumirem que els grafs són sempre dirigits**, on cada Arc interconnecta un parell ordenat de Nodes (origen i destí). **Tanmateix**, els grafs **també poden ser no dirigits**
- Les estructures de dades de tipus graf presenten **moltes aplicacions**, permetent la representació de tot tipus de xarxes de transport (dades, electricitat, aigua, transport públic...), circuits elèctrics, relacions entre tasques, etc...



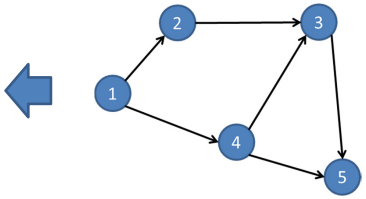
Arcs en grafs dirigits => fletxes

***Assignant un identificador a cada node, estem assumint que el graf és "etiquetat". Tendirem a assumir-ho sempre a POO**

Representació clàssica d'un graf (1/3)

- Existeixen diverses formes de **representació de grafs**, essent la **matriu d'adjacències**, la **llista d'arcs** i la **llista d'adjacències** les tres més habituals
- La **matriu d'adjacències** és una matriu, p.e. d'enters, de dimensió |N|x|N|, on |N| és el nombre de nodes del graf. La posició (i,j) val 1 si existeix un arc al graf des del node "i" fins al node "j", o 0 en cas contrari

```
[0, 1, 0, 1, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 0, 1]
[0, 0, 1, 0, 1]
[0, 0, 0, 0, 0]
```

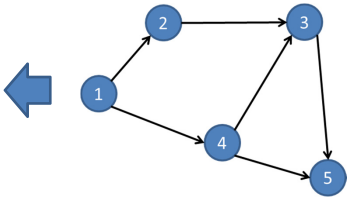


- **L'avantatge principal** de la matriu d'adjacències és que podem **obtenir directament si existeix un determinat arc al graf**. **L'inconvenient principal** és la **quantitat (innecessària) de memòria** que requerim per guardar-la

Representació clàssica d'un graf (2/3)

- La **llista d'arcs** és una forma alternativa de representar un graf, on **només guardem informació dels arcs existents** en aquest, aconseguint així un **estalvi de memòria** substancial vers la matriu d'adjacències

```
[1,2]
[1,4]
[2,3]
[3,5]
[4,3]
[4,5]
```



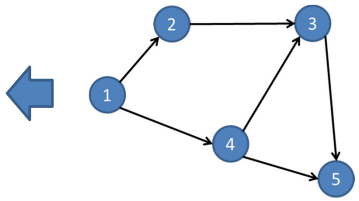
- En aquest cas, requerirem d'una **matriu de dimensió |A|x2 per guardar el graf**, on |A| és el nombre d'arcs del graf. Com a **inconvenient**, per saber **si un arc existeix al graf** serà necessària una **cerca seqüencial** al llarg de la llista d'arcs, fins a trobar-lo (en cas que existeixi)

Representació clàssica d'un graf (3/3)

- La **llista d'adjacències** també permet representar un graf. En aquest cas, guardem la **llista de nodes adjacents a cadascun dels nodes del graf**, una opció entre la llista d'arcs i la matriu d'adjacències

```
[2,4]
[3]
[5]
[3,5]
[-]
```

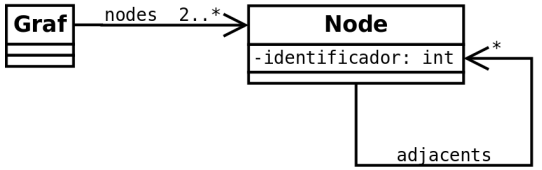
El node 5 no presenta cap arc que s'origini en ell



- La **llista d'adjacències** és la **forma més eficient** de guardar un graf **en termes de memòria requerida** d'entre les 3 que hem vist. Per guardar-la requerirem d'un vector de mida |N|, on cada posició apuntarà un altre vector de tantes posicions com nodes adjacents tingui el node representat per aquesta

Representació d'un graf en Java

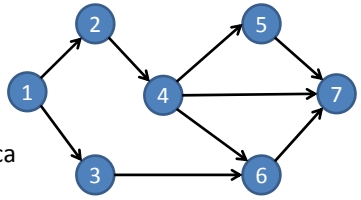
- Quan programem **en Java**, podem utilitzar també les seves **classes contenidores per a la representació de grafs en memòria**. Una opció molt interessant per aquest propòsit són els **mapes**



- El **Graf podria guardar un mapa de Nodes**, identificats pel seu identificador com a clau, i **cada Node guardaria un mapa de nodes adjacents** a aquest
  - Així, la cerca d'un determinat Node al Graf o de l'existència d'un arc determinat en aquest, podria efectuar-se de forma eficient

Exercicis

- **EXEMPLE 13.1**: Assumint el graf següent:



  1. Guardar-lo en tres fitxers de text, en forma de matriu d'adjacències, llista d'arcs i llista d'adjacències. Indicar a la primer línia de text la forma específica de representació (p.e., codi 1, 2 o 3)
  2. Definir la classe Node, tal i com es suggereix a la transparència anterior, definint un mètode constructor i els mètodes accessors que calguin
  3. Definir la classe Graf tal que implementi el mètode constructor:

     public Graf (String pathFitxer) throws FileNotFoundException, RepresentacioInvalidaException;

     on s'intenta carregar en memòria el graf del fitxer pathFitxer, segons la representació corresponent. La classe Graf guardarà la informació del graf en un mapa de Nodes, tal i com s'ha suggerit a la transparència anterior.

Exercicis

4. Implementar el mètode toString() de la classe Graf, tal que retorni un String amb la informació dels nodes del graf i les seves adjacències
5. Implementar la classe principal TestGraf per provar la càrrega correcte del graf, des de qualsevol dels tres fitxers de text

Sortida esperada del programa:

```
Node 1; adjacents: 2,3
Node 2; adjacents: 4
Node 3; adjacents: 6
Node 4; adjacents: 5, 6, 7
Node 5; adjacents: 7
Node 6; adjacents: 7
Node 7; adjacents:
```

Exercicis

- **EXEMPLE 13.2**: Implementar els següents mètodes addicionals a la classe Graf implementada a l'Exemple 13.1:

  public void addNode();

  public void addArc(int idOrigen, int idDest) throws NoNodeException;

  public void deleteNode(int idNode) throws NoNodeException;

  public void deleteArc(int idOrigen, int idDest) throws NoNodeException, NoArcException;

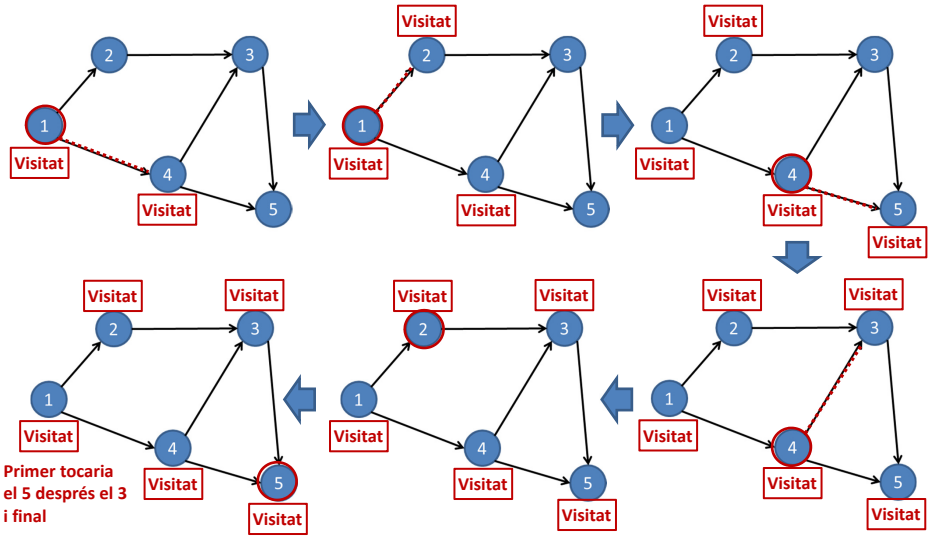
  El primer mètode afegeix un nou node al graf, assignant-li com a identificador el nou nombre de nodes del graf. El segon mètode intenta afegir un nou arc al graf, des del node amb identificador idOrigen fins al node amb identificador idDest. El tercer mètode intenta eliminar del graf el node amb identificador idNode. En aquest cas, s'hauran d'eliminar també del graf, tots els arcs dels quals el node en sigui origen o destí. El darrer mètode elimina l'arc del graf, des del node amb identificador idOrigen fins al node amb identificador idDest

Algorisme BFS (1/2)

- L'algorisme de **cerca en amplada** (en anglès *Breadth First Search*, BFS) és una opció molt usada per a realitzar un **recorregut d'un graf**, per exemple, per determinar si des d'un node origen podem arribar a la resta de nodes
- L'estratègia de BFS per a recórrer un graf és la següent:
  1. Seleccionar el node origen del recorregut, marcant-lo com a visitat
  2. Explorar els nodes adjacents a aquest, marcant-los com a visitats
  3. Per cadascun d'aquests nodes adjacents, en l'ordre en que s'havien visitat, explorar els seus nodes adjacents (no visitats encara) marcant-los també com a visitats
  4. I així successivament, fins que no puguem explorar més nodes
  5. En aquest punt, podrem assegurar que **des del node origen d'aquesta execució de BFS podem arribar a tots els nodes que haguem visitat**

Algorisme BFS (2/2)

- Per exemple, prenent el node 1 com a origen de BFS:



Implementació iterativa de BFS

- Per tal de mantenir l'ordre en que s'exploren els nodes, **BFS** empra una **estructura de dades de tipus cua**, que a efectes pràctics podrem implementar amb una llista, amb **dues operacions bàsiques**:
  - **afegir**: per tal d'afegir un nou element **al final** d'aquesta
  - **extreure**: per tal d'extreure el **primer element** d'aquesta
- Aleshores, el **pseudocodi** de la **implementació iterativa de BFS** és:
  1. Inicialitzar tots els nodes del graf com a no visitats
  2. Seleccionar node origen per al recorregut mitjançant BFS
  3. Marcar node origen com a visitat i afegir a la cua
  4. Mentre cua no buida:
     1. Extreure següent node de la cua (que anomenarem actual)
     2. Per cada node adjacent a actual, si no visitat encara:
        1. Marcar node adjacent com a visitat
        2. Afegir node adjacent a la cua
  5. Final

Exercici

- **EXEMPLE 13.3**: Prenent les classes Graf i Node implementades als exemples 13.1 i 13.2, definir la classe Algorismes. Aquesta classe ha d'implementar el següent mètode estàtic:

  public static List<Integer> breadthFirstSearch (Graf graf, int idOrigen)
  throws NoNodeException;

  Proporcionant-li un graf i l'identificador de node idOrigen, el mètode implementarà l'algorisme BFS des del node amb aquest identificador i retornarà una List<Integer> que contindrà tots els identificadors de nodes visitats, en l'orde amb el que s'han visitat.

  Nota: no introduir cap atribut addicional a la classe Node ni Graf per aquests propòsits.

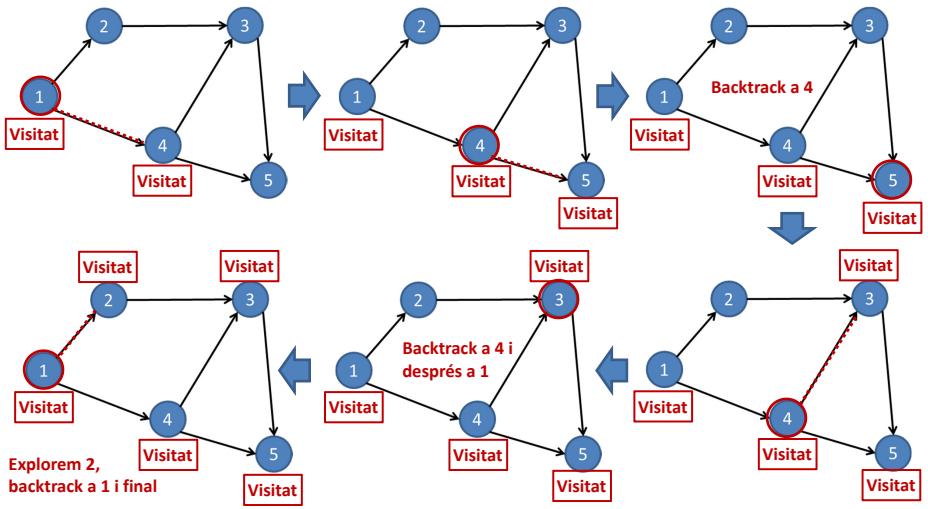
  Posteriorment, provar el correcte funcionament de l'algorisme implementat, prenent diferents nodes com a origen

Algorisme DFS (1/2)

- L'algorisme de **cerca en profunditat** (en anglès *Depth First Search*, DFS) especifica una **forma alternativa a BFS per recórrer un graf**:
  1. Seleccionar el node origen del recorregut, marcant-lo com a visitat
  2. Començar a explorar els nodes adjacents a aquest, marcant-los com a visitats (els que no ho estiguessin encara)
  3. En cas de trobar un node adjacent no visitat, passar directament a explorar-lo i així successivament (recorregut en profunditat)
  4. Si no hi ha més nodes adjacents no visitats per explorar, tornar enrere al node anterior (*backtracking*), continuant amb l'exploració dels nodes adjacents restants (pendents d'explorar encara) d'aquest
  5. I així fins a retornar altre cop al node origen, on ja no quedin nodes adjacents no visitats per explorar
  6. En aquest punt, podrem assegurar que **des del node origen d'aquesta execució de DFS podem arribar a tots els nodes que haguem visitat**

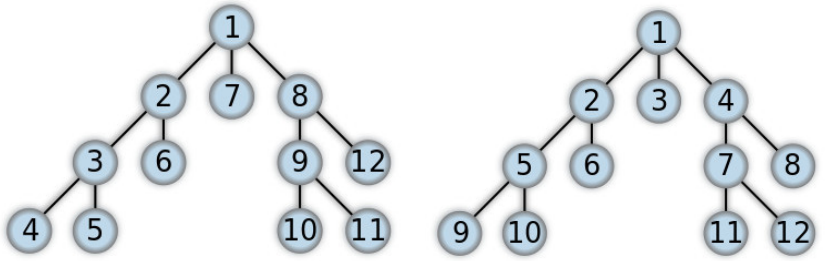
Algorisme DFS (2/2)

- Per exemple, prenent el node 1 com a origen de DFS:



Depth vs. Breadth First Search

- Una forma molt visual per destacar les diferències entre la funcionalitat de DFS i la de BFS és considerar un graf en forma d'arbre com el següent:



Ordre en el que DFS recorreria els nodes

Ordre en el que BFS recorreria els nodes

***Figures obtingudes https://en.wikipedia.org**

Implementació iterativa de DFS

- Per tal de mantenir l'ordre en que s'exploren els nodes, **DFS** empra una **estructura de dades de tipus pila** (que també podrem implementar amb una llista, a efectes pràctics), amb **dues operacions bàsiques**:
  - **push**: per tal d'afegir un **nou element dalt de la pila**
  - **pop**: per tal d'extreure el **darrer element apilat a la pila**
- Aleshores, el **pseudocodi** de la **implementació iterativa de DFS** és:

1. Inicialitzar tots els nodes del graf com a no visitats
2. Seleccionar node origen per al recorregut mitjançant DFS
3. Afegir (push) node origen a la pila
4. Mentre pila no buida:
   1. Extreure (pop) node de la pila (que anomenarem actual)
   2. Si node actual no visitat encara:
      1. Marcar node actual com a visitat
      2. Per cada node adjacent a actual, si no visitat encara:
         1. Afegir (push) node adjacent a la pila

No marcar com a visitat encara

5. Final

Exercici

- **EXEMPLE 13.4**: Reprendre la implementació de la classe Algorismes, començada a definir a l'Exemple 13.3, afegint el següent mètode estàtic:

```
public static List<Integer> depthFirstSearch (Graf graf, int idOrigen)
                                                    throws NoNodeException;
```

Proporcionant-li un graf i l'identificador de node idOrigen, el mètode implementarà l'algorisme DFS des del node amb aquest identificador i retornarà una List<Integer> que contindrà tots els identificadors de nodes visitats, en l'orde amb el que s'han visitat.

Nota: no introduir cap atribut addicional a la classe Node ni Graf per aquests propòsits.

Posteriorment, provar el correcte funcionament de l'algorisme implementat, prenent diferents nodes com a origen. Observar com l'ordre en el que DFS ha visitat els diferents nodes del graf és diferent al que obteníem mitjançant BFS a l'exemple anterior.

Implementació recursiva de DFS (1/2)

- La **recursivitat** és una **estratègia de programació** emprada freqüentment per tal d'obtenir **implementacions més simples** per dur a terme una determinada tasca
- Aquesta consisteix en permetre que un mètode pugui invocar-se ell mateix en un determinat punt de la seva implementació
- Tot i ser una estratègia molt beneficiosa en certs casos, cal tenir en compte però dos aspectes fonamentals:
  - Cal **assegurar sempre** que la **recursivitat finalitza** de forma adequada, evitant bucles infinits que exhaureixin la memòria del sistema
  - La recursivitat, per defecte, **requereix recursos del sistema**, doncs emplena la pila d'invocacions de mètodes. Per tant, **només és desitjable quan simplifiqui molt substancialment la implementació iterativa** de la tasca a resoldre

Implementació recursiva de DFS (2/2)

- L'algorisme DFS és un bon exemple on la recursivitat en simplifica molt la seva implementació i val la pena tenir-la en compte
- Pseudocodi de la implementació recursiva de DFS:

mètode **depthFirstSearch** (graf, nodeOrigen):

1. Marcar nodeOrigen com a visitat
2. Per a cada node adjacent nodeAdjacent:
   1. Si nodeAdjacent no visitat encara
      1. **depthFirstSearch** (graf, nodeAdjacent) //Crida recursiva
3. Final

- Observant el pseudocodi, fixem-nos que aquest finalitzarà adequadament quan tots els nodes adjacents al node origen inicial hauran estat visitats

Exercici

- **EXEMPLE 13.5**: A la classe Algorismes, començada a definir als exemples 13.3 i 13.4, afegir el següent mètode estàtic:

  public static void recursiveDepthFirstSearch (Graf graf, int idOrigen, List<Integer> resultat) throws NoNodeException;

  Proporcionant-li un graf i l'identificador de node idOrigen i una List<Integer> inicialment buida, el mètode implementarà l'algorisme DFS recursiu des del node amb aquest identificador i emplenarà la List<Integer> proporcionada com a paràmetre amb tots els identificadors de nodes visitats, en l'orde amb el que s'han visitat.

  Nota: no introduir cap atribut addicional a la classe Node ni Graf per aquests propòsits.

  És diferent l'ordre obtingut, en comparació amb l'algorisme DFS iteratiu? Per què creus que és així?

BFS per al còmput del camí més curt

- Tant BFS com DFS ofereixen un gran nombre d'aplicacions en estructures de dades de tipus graf. En particular, **BFS** és de gran utilitat per al **còmput del camí més curt** entre un node origen i un node destí d'un graf
- Cal destacar però que, en aquest cas, la **mètrica per determinar la llargada d'un camí** és el **nombre d'arcs (o "salts")** d'aquest. En altres paraules, s'assumeix que tots els arcs presenten valor de mètrica = 1
  - Si es volgués considerar valors de mètrica diferents per cada arc, caldria usar altres algorismes més complexos (p.e., algorisme de Dijkstra)
- Aleshores, l'estratègia a seguir és ben senzilla:
  - Com que BFS recorre el graf nivell a nivell des del node origen, **quan visitem el node destí** podem assegurar que **hi haurem arribat pel camí més curt** (ens quedarem amb el primer que trobem!)
  - Acte seguit, haurem de **reconstruir la seqüència de nodes que haurem seguit** per arribar-hi => El camí més curt trobat

Exercici

- **EXEMPLE 13.6**: Seguint amb la implementació de la classe Algorismes, definir el següent mètode estàtic:

  public static List<Integer> shortestPathBFS (Graf graf, int idOrigen, int idDesti) throws NoNodeException, NoCamiException;

  Proporcionant-li un graf i els identificadors de node origen i destí, el mètode retorna una List<Integer> emplenada amb els identificadors de la seqüència de nodes que descriuen el camí més curt entre aquests, computat mitjançant BFS. Cal tenir en compte que si BFS no arriba a explorar el node destí, això implica que no existirà cap camí possible des del node origen a aquest.

  Suggeriment: per tal de reconstruir el camí més curt un cop arribem a destí, **guardar els nodes visitats en un Mapa**, on les parelles <CLAU, VALOR> siguin <Node visitat, Node predecessor>.