Estructures de dades

Informàtica
curs 23/24
Part 2/2

Prof. D. Tost

Grafs: tipus de grafs

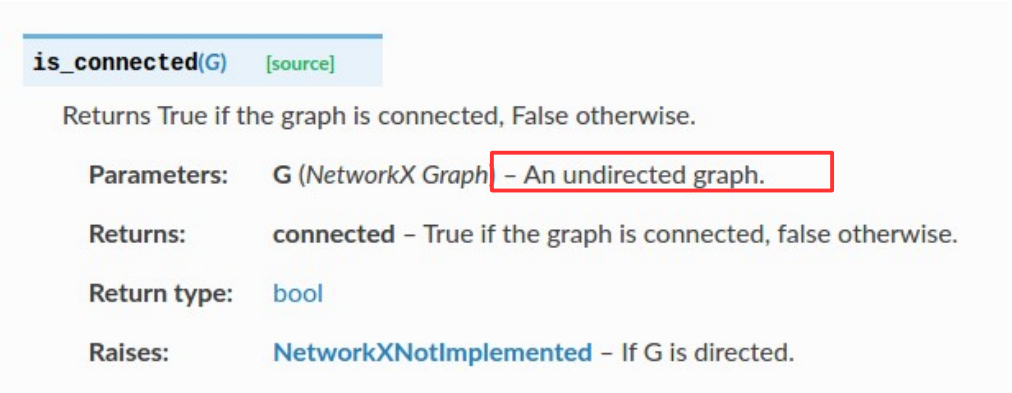
Graf no dirigit: les arestes no tenen orientació $(u, v) = (v, u)$
Exemple: `R` representa «ser amic de»

Graf dirigit (digraf): les arestes tenen orientació $(u, v) \neq (v, u)$
Exemple: `R` representa «ser parent (pare o mare) de»

Multigraf (no dirigit): hi pot haver més d'una aresta entre dos nodes
Exemple: `R` representa «haver quedat amb»

Multigraf dirigit: les arestes tenen orientació i hi pot haver més d'una entre dos nodes
Exemple: `R` representa «haver invitat a»

Alerta: algunes funcions s'apliquen només sobre un tipus específic de grafs.
e.g.

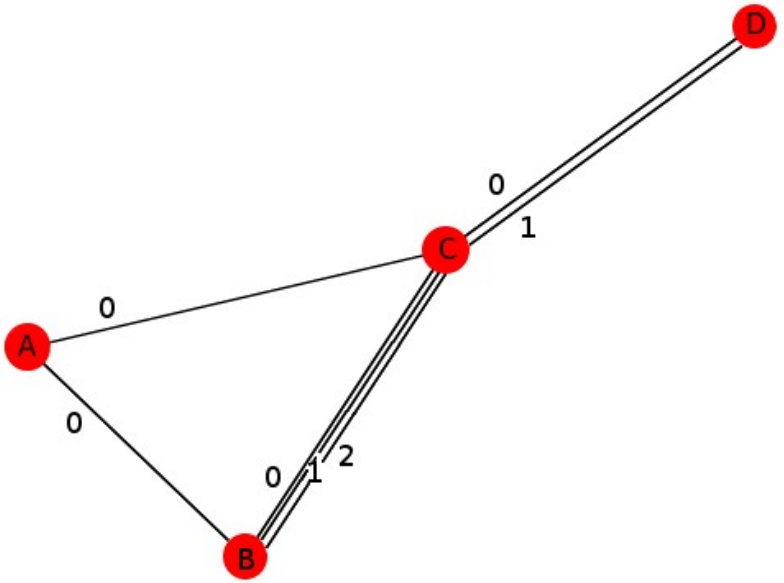


is_connected(*G*) [source]

Returns True if the graph is connected, False otherwise.

Parameters:	**G** (*NetworkX Graph*) – An undirected graph.
Returns:	**connected** – True if the graph is connected, false otherwise.
Return type:	bool
Raises:	**NetworkXNotImplemented** – If G is directed.

Multigraphs



g = nx.MultiGraph()

El graf és un diccionari de diccionaris de diccionaris...:
Clau = node
Valor = diccionari de veïns
- Clau: veí
- **Valor: diccionari d'arestes entre node i veí**
  - Clau: identificador d'aresta
  - Valor: diccionari d'atributs
    - Clau: nom de l'atribut
    - Valor: valor de l'atribut

```
>>> g['B']
{'C':
      {0:
         {'name': 'N3',
          'color':'red'},
       1:
         {'name': 'A5'},
       2:
         {'name': 'P4',
          'color':'blue'},
      },
 'A':
      {0:
         {'name': 'A6'}}
}
```

MultiGraph

Recorregut

```
for node in g:
    for vei in g[node] :
        for aresta in g[node][vei]:
            for atribut in g[node][vei][aresta] :
                g[node][vei][aresta][atribut] és el valor del atribut
```

MultiGraph

Intenteu fer un croquis del graf a partir de la informació següent:

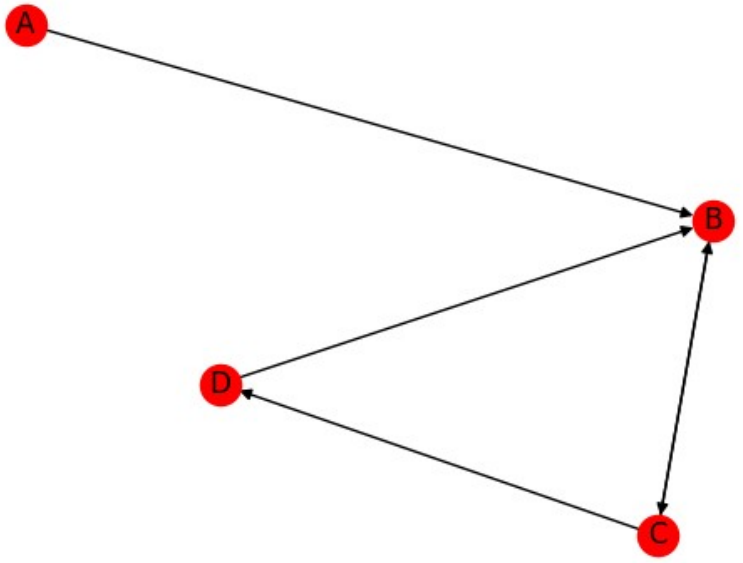
```
>>> for node in g:
...     for vei in g[node]:
...         for aresta in g[node][vei] :
...             for atribut in g[node][vei][aresta] :
...                 print("Node = {} Vei = {} Aresta = {} Atribut = {} Valor = {:}"
                          .format(node, vei, aresta, atribut,
                                  g[node][vei][aresta][atribut]))
...
Node = A Vei = B Aresta = 0 Atribut = name Valor = coco
Node = A Vei = B Aresta = 0 Atribut = color Valor = blanc
Node = A Vei = B Aresta = 1 Atribut = name Valor = poma
Node = A Vei = B Aresta = 1 Atribut = color Valor = red
Node = B Vei = A Aresta = 0 Atribut = name Valor = coco
Node = B Vei = A Aresta = 0 Atribut = color Valor = blanc
Node = B Vei = A Aresta = 1 Atribut = name Valor = poma
Node = B Vei = A Aresta = 1 Atribut = color Valor = red
Node = B Vei = C Aresta = 0 Atribut = color Valor = negre
Node = B Vei = C Aresta = 0 Atribut = name Valor = taronja
Node = C Vei = B Aresta = 0 Atribut = color Valor = negre
Node = C Vei = B Aresta = 0 Atribut = name Valor = taronja
```

Digrafs - Graf dirigit

La relació del graf no és simètrica, les arestes són dirigides

Un node té arestes d'entrada «in-edges» i arestes de sortida «out_edges»

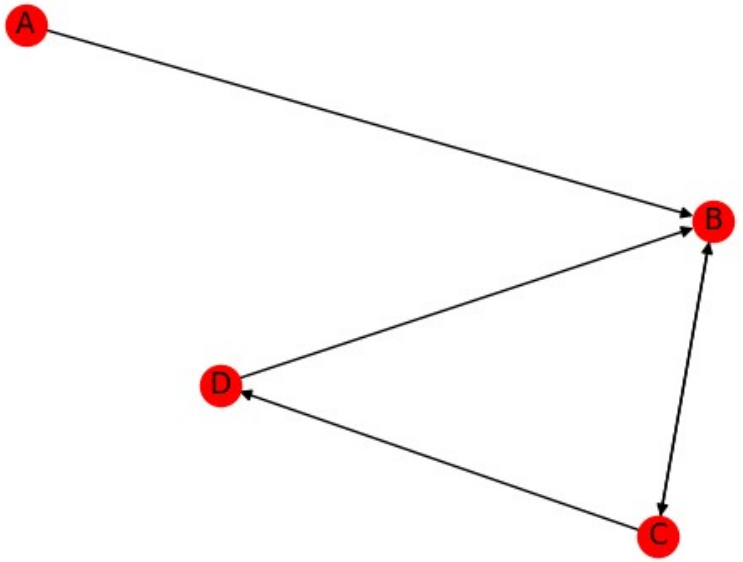
G = nx.DiGraph()




```
>>> g.in_degree('A')
0
>>> g.in_degree('B')
3
>>> g.in_degree('C')
1
>>> g.in_degree('D')
1
>>>
g.out_degree('A')
1
>>>
g.out_degree('B')
1
>>>
g.out_degree('C')
2
>>>
g.out_degree('D')
1
```

Graf dirigit

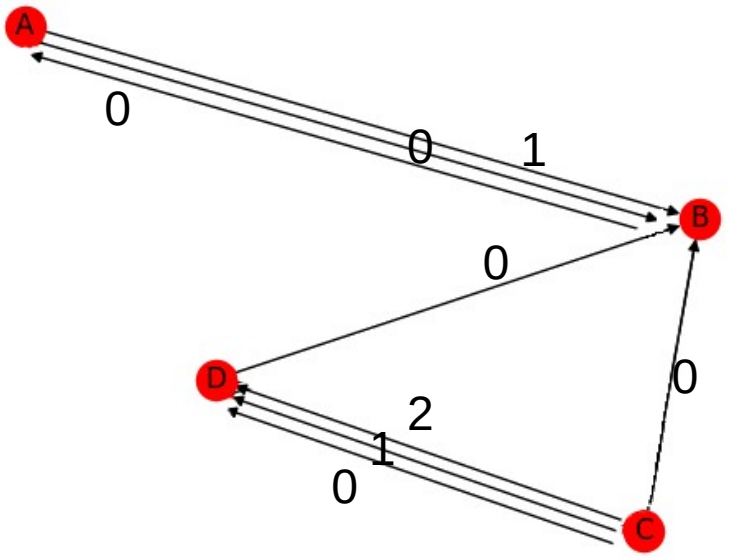
Els veïns d'un nodes són els nodes que reben una aresta del node donat.




```
>>> for node in g:
...     print("Node: ", node)
...     for vei in g[node] :
...         print("\t"+ vei)
...
Node:   A
    B
Node:   B
    C
Node:   C
    D
    B
Node:   D
    B
```

MultiDiGraf

Les arestes són dirigides i n'hi pot haver més d'una entre nodes

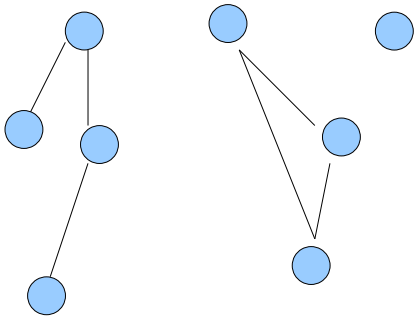



```
g = nx.MultiDiGraph()
```

```
>>> g['A']
AdjacencyView({'B': {0: {'name': 'A-B-1'}, 1: {'name': 'A-B-2'}}})
>>> g['B']
AdjacencyView({'A': {0: {'name': 'B-A-1'}}})
```

Algorismes sobre grafs

Components connexes



Una component connexa d'un graf **no dirigit** està formada per nodes que estan connectats entre ells de forma directa o indirecta

3 components connexes

Es pot calcular:

- els nodes de cadascuna de les components connexes
- els subgrafs de cada component connexa
- els nodes de la components connexa a la que pertany un node concret
- el subgraf de la component connexa a la que pertany un node concret

Components connexes

Connectivity

`is_connected` (G)	Return True if the graph is connected, False otherwise.
`number_connected_components` (G)	Return the number of connected components.
`connected_components` (G)	Generate connected components.
`connected_component_subgraphs` (G[, copy])	DEPRECATED: Use `(G.subgraph(c) for c in connected_component`
`node_connected_component` (G, n)	Return the set of nodes in the component of graph containing

Exemples d'aplicacions:

- Identificar els grups de persones relacionades en un graf de relacions

- Identificar les ubicacions on es pot arribar en bici des d'una ubicació en un graf de posicions geogràfiques relacionats amb camins ciclables

Nodes aïllats

No tenen cap veí ni directe ni indirecte. Formen per si sols una component connexa.

Isolates

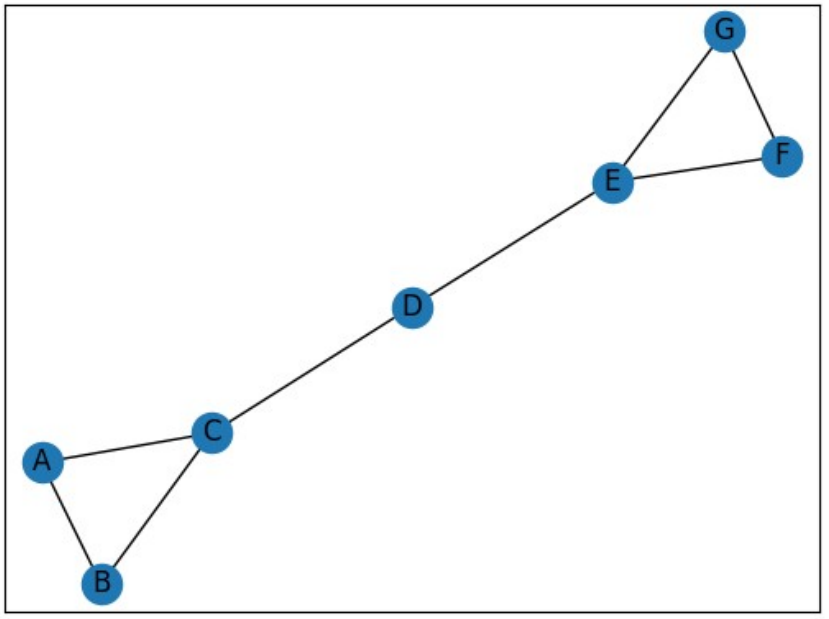
Functions for identifying isolate (degree zero) nodes.

`is_isolate`(G, n)	Determines whether a node is an isolate.
`isolates`(G)	Iterator over isolates in the graph.
`number_of_isolates`(G)	Returns the number of isolates in the graph.

Camins

Camins simples (només passen un cop per un node)

S'especifica el node origen, el node destí i es poden trobar tots els camins, les arestes de tots el camins o determinar si un camí és simple

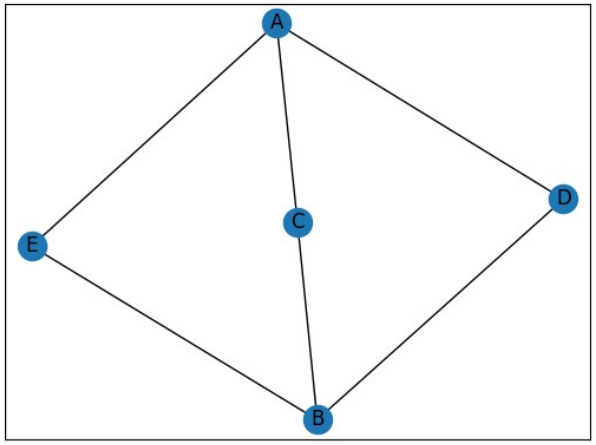



```
>>> it = nx.all_simple_paths(g, 'B', 'G')
>>> next(it)
['B', 'A', 'C', 'D', 'E', 'F', 'G']
>>> next(it)
['B', 'A', 'C', 'D', 'E', 'G']
>>> next(it)
['B', 'C', 'D', 'E', 'F', 'G']
>>> next(it)
['B', 'C', 'D', 'E', 'G']
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Camins més curts

Camins més curts

S'especifica el node origen, el node destí i es poden trobar tots els camins, les arestes de tots el camins o determinar si un camí és simple



Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

`shortest_path` (G[, source, target, weight, ...])	Compute shortest paths in the graph.
`all_shortest_paths` (G, source, target[, ...])	Compute all shortest simple paths in the graph.
`shortest_path_length` (G[, source, target, ...])	Compute shortest path lengths in the graph.
`average_shortest_path_length` (G[, weight, method])	Returns the average shortest path length.
`has_path` (G, source, target)	Returns *True* if *G* has a path from *source* to *target*.

```
>>> it = nx.all_shortest_paths(g, 'A', 'B')
>>> next(it)
['A', 'C', 'B']
>>> next(it)
['A', 'D', 'B']
>>> next(it)
['A', 'E', 'B']
```

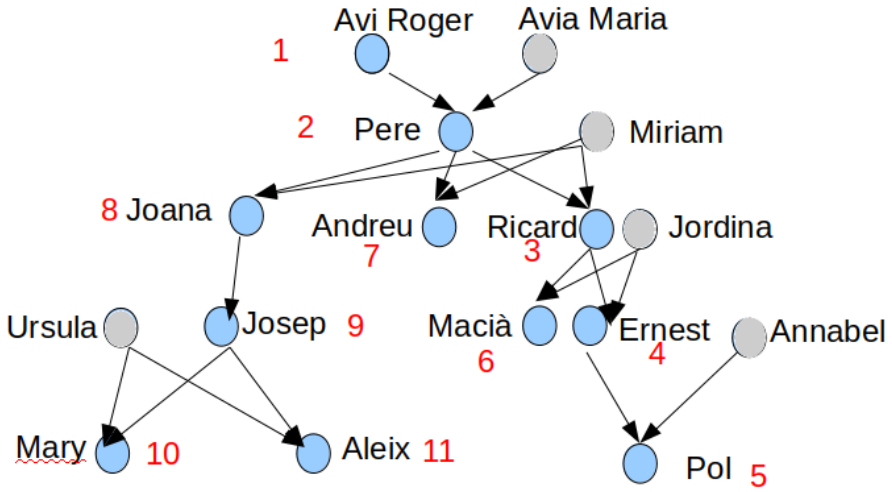
```
>>> nx.shortest_path(g, 'A', 'B')
['A', 'C', 'B']
```

Cerca / recorregut

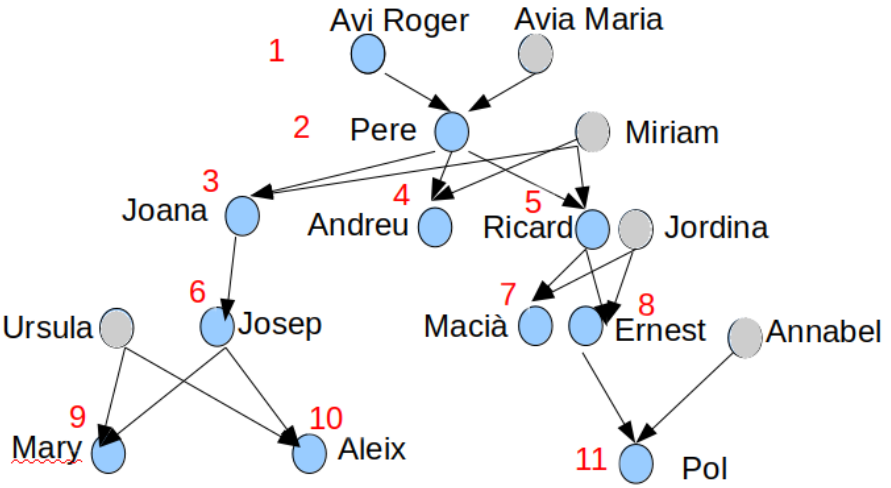
Es recorre el graf fins a trobar un node que compleix una condició (o fins a llistar tots els descendents). Hi ha dues estratègies: cercar (recorre) en profunditat o en amplada.

Estratègia de la cerca/recorregut en profunditat: començant pel node origen, arribar fins el més lluny possible, retrocedir un nivell i tornar a començar.

Estratègia de la cerca/recorregut en amplada: començant pel node origen, es visiten tots els descendents d'una generació, a continuació els seus descendents i així successivament.



Depth-first amb node inicial Avi Roger



Breadth-first amb node inicial Avi Roger

Els nodes en gris no es visiten

Recorregut/cerca DFS

Implementació

```
def IteradorDfs(g, arrel) :
    c=pila.Pila()
    c.empilar(arrel)
    s = set([arrel])
    while not c.esbuida()  :
        v = c.desempilar()
        yield v
        for x in g.successors(v):
            if x not in s:
                s.add(x)
                c.empilar(x)
```

'Avi Roger'	→ 'Avi Roger'
'Pere'	→ 'Pere'
'Joana', 'Andreu', 'Ricard'	→ 'Ricard'
'Joana', 'Andreu', 'Macià', 'Ernest'	→ 'Ernest'
'Joana', 'Andreu', 'Macià', 'Pol'	→ 'Pol'
'Joana', 'Andreu', 'Macià'	→ 'Macià'
'Joana', 'Andreu'	→ 'Andreu'
'Joana'	→ 'Joana'
'Josep'	→ 'Josep'
'Mary', 'Aleix'	→ 'Aleix'
'Mary'	→ 'Mary'
Pila buida	

Si es fa el yield després de processar els successors, s'obté un ordre diferent (post-ordre). L'indicat a l'algorisme és pre-ordre.

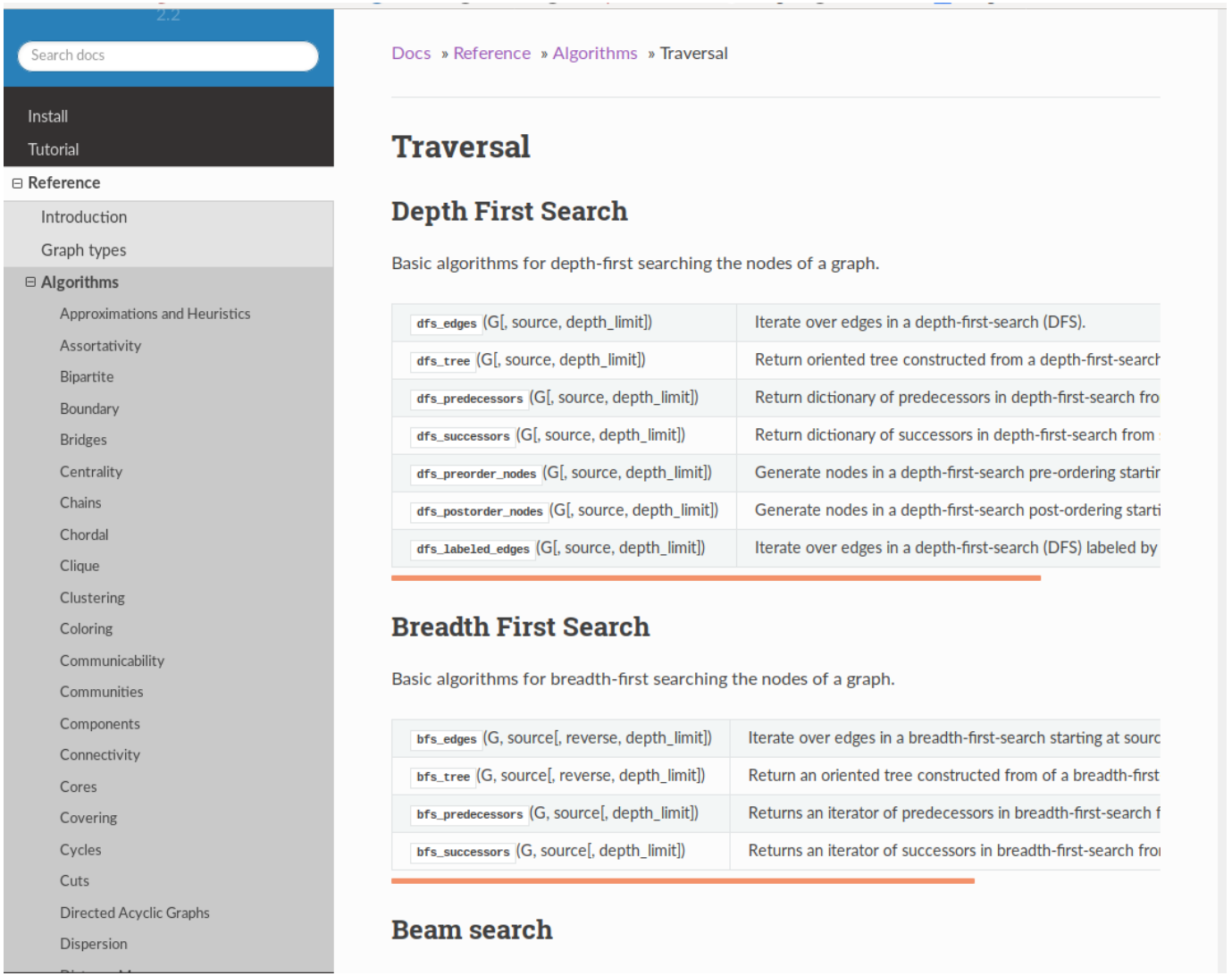
Recorregut/cerca BFS

Implementació

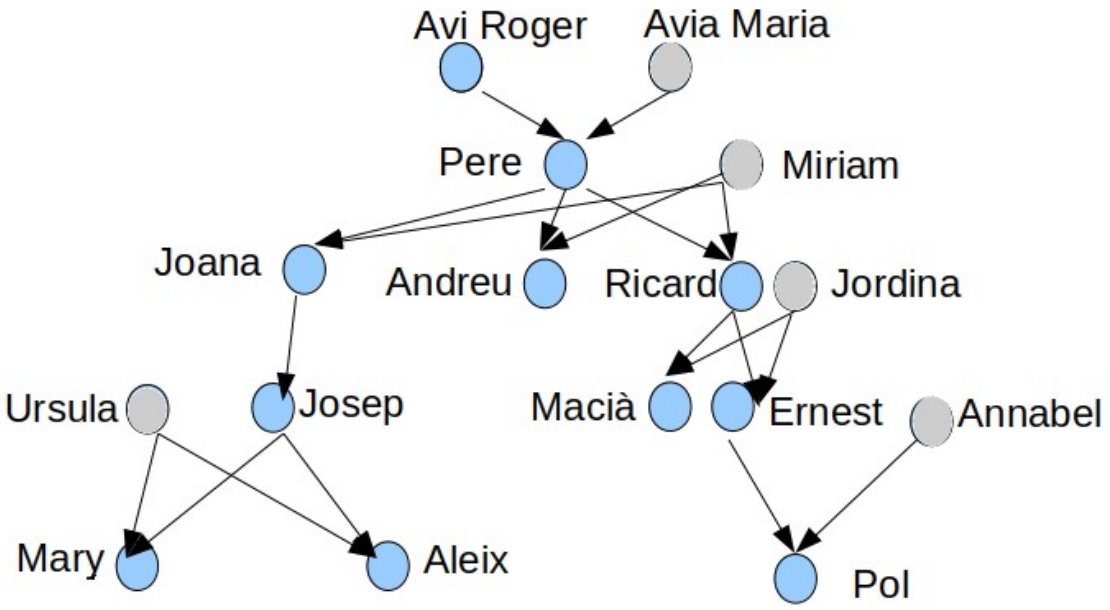
```
def IteradorBfs(g, arrel) :
    c=cua.Cua()
    c.encuar(arrel)
    s = set([arrel])
    while not c.esbuida()  :
        v = c.desencuar()
        yield v
        for x in g.successors(v):
            if x not in s:
                s.add(x)
                c.encuar(x)
```

'Avi Roger'	→ 'Avi Roger'
'Pere'	→ 'Pere'
'Joana', 'Andreu', 'Ricard'	→ 'Joana'
'Andreu', 'Ricard', 'Josep'	→ 'Andreu'
'Ricard', 'Josep'	→ 'Ricard'
'Josep', 'Macià', 'Ernest'	→ 'Josep'
'Macià', 'Ernest','Mary', 'Aleix'	→ 'Macià'
'Ernest','Mary', 'Aleix'	→ 'Ernest'
'Mary', 'Aleix', 'Pol'	→ 'Mary'
'Aleix', 'Pol'	→ 'Aleix'
'Pol'	→ 'Pol'
Cua buida	

Networkx DFS and BFS



Networkx DFS and BFS




```
>>> list(nx.dfs_edges(g, 'Avi Roger'))
[('Avi Roger', 'Pere'), ('Pere', 'Joana'), ('Joana', 'Josep'), ('Josep',
'Mary'), ('Josep', 'Aleix'), ('Pere', 'Andreu'), ('Pere', 'Ricard'),
('Ricard', 'Macia'), ('Ricard', 'Ernest'), ('Ernest', 'Pol')]

>>> list(nx.bfs_edges(g, 'Avi Roger'))
('Avi Roger', 'Pere'), ('Pere', 'Joana'), ('Pere', 'Andreu'), ('Pere',
'Ricard'), ('Joana', 'Josep'), ('Ricard', 'Macia'), ('Ricard', 'Ernest'),
('Josep', 'Mary'), ('Josep', 'Aleix'), ('Ernest', 'Pol')]
```