.

Ampliació d'Informàtica
Apunts d'estructures de dades

Maria Dolors Ayala Vallespí
Dept. Ciències de la computació
E.T.S.E.I.B.
Universitat Politècnica de Catalunya

Tardor 2023

Contents

1 Estructures de dades

Nota prèvia: Aquest capítol és una introducció a les possibilitats de diverses funcions i classes que representen estructures de dades. És imprescindible consultar sempre la documentació Python abans d'usar una funció o un mètode: allà hi ha l'especificació completa i actualitzada, així com altres mètodes que no s'introdueixen en aquest capítol.

1.1 Sets

Un conjunt (tipus `set`) és una col·lecció no ordenada d'elements únics: no hi ha elements repetits. Els conjunts són mutables. Python proporciona una classe predefinida, `set`, per representar-los (com amb els strings, llistes, etc.). Un conjunt es representa amb els seus elements separats per comes i entre claus: `{3, 5, 'hola'}`. Operacions amb la classe `set`:

- creació d'un conjunt buit: `s = set()`
- operador de pertinença: `x in s`
- nombre d'elements: `len(s)`
- recorregut dels elements: `for e in s:`

Els conjunts no són un tipus seqüencial i NO tenen els operadors d'indexació ni slicing (llesques).

Aquests són els mètodes més habituals:

- mètodes bàsics: `add, clear, copy, pop, remove`
- mètodes per determinar disjunció i subconjunts: `isdisjoint, issubset, issuperset`
- mètodes per operacions booleanes NO modificadors: `union, intersection, difference, symmetric_difference`
- mètodes per operacions booleanes modificadors: `update, intersection_update, difference_u symmetric_difference_update`

Descripció dels mètodes generals:

- `add`. Modificador. Afegeix un element al conjunt.
- `clear`. Modificador. Elimina tots els elements del conjunt.
- `copy`. Retorna una còpia del conjunt sense efecte àlies.
- `pop`. Modificador. Esborra i retorna un element a l'atzar del conjunt
- `remove`. Modificador. Esborra l'element del conjunt. Cal que l'element pertanyi al conjunt.
- `isdisjoint`. Retorna `True` si els dos conjunts tenen la intersecció nul·la.

- `issubset`. Retorna `True` si un altre conjunt conté aquest conjunt.
- `issuperset`. Retorna `True` si aquest conjunt conté un altre conjunt.

Descripció dels mètodes NO modificadors relacionats amb operacions booleanes:

- `union`. Retorna la unió dels conjunts|
- `intersection`. Retorna la intersecció entre els dos conjunts.&
- `difference`. Retorna la diferència entre els conjunts.−), symmetric_difference
- update, intersection_update, difference_update, symmetric_difference_update

Descripció dels mètodes generals:

- `add`. Modificador. Afegeix un element al conjunt.
- `clear`. Modificador. Elimina tots els elements del conjunt.
- `copy`. Retorna una còpia del conjunt sense efecte àlies.
- `pop`. Modificador. Esborra i retorna un element a l'atzar del conjunt
- `remove`. Modificador. Esborra l'element del conjunt. Cal que l'element pertanyi al conjunt.
- `isdisjoint`. Retorna `True` si els dos conjunts tenen la in
- `symmetric_difference`. Retorna el conjunt d'elements que pertanyen només a un dels conjunts. [1]

Descripció dels mètodes modificadors relacionats amb operacions booleanes:

- `update`. Modifica el conjunt fent la unió entre ell i els altres.
- `intersection_update`. Modifica el conjunt fent la intersecció entre ell i els altres.
- `difference_update`. Modifica el conjunt esborrant tots els elements que pertanyen a l'altre conjunt.
- `symmetric_difference_update`. Modifica el conjunt fent la diferència simètrica entre ell i l'altre.

[1] La diferència simètrica entre A i B, $A \oplus B$ es pot expressar com:

1. $(A - B) \cup (B - A)$
2. $(A \cup B) - (A \cap B)$

Exemples:

```
>>> a = set([1, 2, 3, 4, 5, 6, 7])
>>> a.add(8)
>>> a == {1, 2, 3, 4, 5, 6, 7, 8}
True
>>> a.pop()  # traiem un element a l'atzar
1
>>> a == {2, 3, 4, 5, 6, 7, 8}
True
>>> a.remove(6)  # eliminem un element concret
>>> a == {2, 3, 4, 5, 7, 8}
True
>>> b = set()   # conjunt buit
>>> b.add(2)
>>> b.add(3)
>>> b.add(5)
>>> b == {5, 2, 3}
True
>>> a.isdisjoint(b)
False
>>> c = {4, 6, 8}
>>> c.isdisjoint(b)
True
>>> b.issubset(a)
True
>>> a.issuperset(b)
True
>>> aun = a.union(b)             # equival a a|b
>>> aun == {2, 3, 4, 5, 7, 8}
True
>>> aun1 = a | b
>>> aun1 == {2, 3, 4, 5, 7, 8}
True
>>> aint = a.intersection(b)     # equival a a&b
>>> aint == {5, 2, 3}
True
>>> a.update(b)
>>> a == {2, 3, 4, 5, 7, 8}
True
>>> d = a.copy()           # còpia de a sense efecte àlies
>>> b = {5, 2, 3}
>>> d.intersection_update(b)
>>> d == {5, 2, 3}
True
```

1.2 Cues

Una cua és una estructura de dades de tipus FIFO (first in first out): els elements (nodes) s'afegeixen al darrera i es treuen del davant.

Python disposa de la classe `queue.Queue` amb els mètodes següents:

- `put`. Afegeix un element al final

- `get`. Extreu i retorna l'element del davant
- `qsize`. Retorna el nombre d'elements de la cua.
- `empty`. Retorna un booleà que indica si la cua és buida.

Exemples:

```
>>> c = Queue()
>>> c.put('A') # elements de c: 'A'
>>> c.put('B') # elements de c: 'A', 'B'
>>> c.put('C') # elements de c: 'A', 'B', 'C'
>>> c.put('D') # elements de c: 'A', 'B', 'C', 'D'
>>> c.get()    # elements de c: 'B', 'C', 'D'
'A'
>>> c.get()    # elements de c: 'C', 'D'
'B'
>>> c.qsize()
2
```

També podríem dissenyar una classe `Cua` amb els mètodes `encuar` (`put`), `desencuar` (`get`)) i la funció `len`. La implementació es pot fer amb una llista:

```
class Cua(object):
    def __init__(self):
        self.elems = []
    def encuar(self, e):
        self.elems.append(e)
    def desencuar(self):
        return self.elems.pop(0)
    def __len__(self):
        return len(self.elems)
```

Exemples:

```
>>> c = Cua()
>>> c.encuar('A') # elements de c: 'A'
>>> c.encuar('B') # elements de c: 'A', 'B'
>>> c.encuar('C') # elements de c: 'A', 'B', 'C'
>>> c.encuar('D') # elements de c: 'A', 'B', 'C', 'D'
>>> c.desencuar() # elements de c: 'B', 'C', 'D'
'A'
>>> c.desencuar() # elements de c: 'C', 'D'
'B'
>>> len(c)
2
```

Un concepte derivat de les cues són les cues amb prioritat. En aquest cas, cada node a part del valor porta associada una prioritat. Aquesta estructura ja no segueix l'esquema FIFO: els elements es posen al darrera, però quan se'n treu un es té en compte la prioritat.

Python disposa de la classe `queue.PriorityQueue` amb els mètodes: `put`, `get` i `qsize`. La prioritat està relacionada amb el valor, en l'ordre de petit a gran. Si aquest no és el criteri de prioritat que es vol, els elements es poden representar per tuples de la forma (prioritat, valor).

Exemples:

```
>>> cp = PriorityQueue()
>>> cp.put((2, 'Laia'))  # afegim elements (prioritat, valor)
>>> cp.put((1, 'Joan'))
>>> cp.put((2, 'Anna'))
>>> cp.put((0, 'Pere'))
>>> cp.get()   # retorna el de prioritat 0
(0, 'Pere')
>>> cp.get()   # retorna el de prioritat 1
(1, 'Joan')

# pels de prioritat 2, té en compte l'ordre lexicogràfic del valor
# 'Anna' anirà abans que 'Laia'
>>> cp.get()
(2, 'Anna')
>>> cp.get()
(2, 'Laia')
```

1.3 Piles

Una pila és una estructura de dades de tipus LIFO (last in first out): els elements (nodes) s'afegeixen i es treuen pel mateix costat de l 'estructura, que s'anomena *cim*.

Python disposa de la classe `queue.LifoQueue` amb els mètodes: `put`, `get` i `qsize`.

Exemples:

```
>>> p = LifoQueue()
>>> p.put('A') # elements de p: 'A' (cim)
>>> p.put('B') # elements de p: 'A', 'B' (cim)
>>> p.put('C') # elements de p: 'A', 'B', 'C' (cim)
>>> p.put('D') # elements de p: 'A', 'B', 'C', 'D' (cim)
>>> p.get()    # elements de p: 'A', 'B', 'C' (cim)
'D'
>>> p.get()    # elements de p: 'A', 'B' (cim)
'C'
>>> p.qsize()
2
```

També podríem dissenyar una classe `Pila` amb els mètodes `empilar`, `desempilar` i la funció `len`. La implementació també es pot fer amb una llista:

```
class Pila(object):
    def __init__(self):
        self.elems = []
    def empilar(self, e):
        self.elems.append(e)
    def desempilar(self):
        return self.elems.pop(-1)
    def __len__(self):
        return len(self.elems)
```

Exemples:

```
>>> p = Pila()
>>> p.empilar('A') # elements de p: 'A' (cim)
>>> p.empilar('B') # elements de p: 'A', 'B' (cim)
>>> p.empilar('C') # elements de p: 'A', 'B', 'C' (cim)
>>> p.empilar('D') # elements de p: 'A', 'B', 'C', 'D' (cim)
>>> p.desempilar() # elements de p: 'A', 'B', 'C' (cim)
'D'
>>> p.desempilar() # elements de p: 'A', 'B' (cim)
'C'
>>> len(p)
2
```

2 Grafs

2.1 Introducció

Un graf $G = (N, A)$ és una estructura de dades formada per un conjunt de nodes (vèrtexs) N i un conjunt d'arestes (arcs) A.

Un graf està associat a una relació binària R entre nodes: hi ha una aresta entre dos nodes u i v si uRv.

Els nodes i arestes d'un graf poden dur informació associada amb etiquetes (o atributs); en anglès: *node attribute*, *edge attribute*.

Considerarem els següents tipus de grafs:

- Graf no dirigit. Les arestes no tenen orientació: $aRb \Leftrightarrow bRa$. Només hi pot haver una aresta entre dos nodes. Exemple: R representa "ser amic de"
- Graf dirigit. Les arestes tenen orientació: $aRb \not\Leftrightarrow bRa$. Només una aresta amb la mateixa orientació entre dos nodes. Exemple: R representa "estar enamorat de"
- Multigraf (no dirigit). Les arestes no tenen orientació. Hi pot haver més d'una aresta entre dos nodes. Exemple: R representa "quedar per dinar amb"
- Multigraf dirigit. Les arestes tenen orientació. Hi pot haver més d'una aresta entre dos nodes. Exemple: R representa "fer un regal a"

2.1.1 Definicions

Definicions generals

- **ordre** d'un graf: nombre de nodes
- **mida** d'un graf: nombre d'arestes
- **etiqueta** (atribut) d'un node o d'una aresta: informació complementària del node o aresta
- **veïnatge** (adjacència), per grafs no dirigits: dos nodes u i v són veïns si existeix l'aresta (u, v)
- **grau** d'un node, per grafs no dirigits: nombre d'arestes del node (nombre de veïns)

Definicions per grafs dirigits

- **predecessors** d'un node u: és el conjunt de nodes v tals que existeix l'aresta (v, u)
- **successors** d'un node u: és el conjunt de nodes v tals que existeix l'aresta (u, v)
- **grau d'entrada** d'un node u: nombre d'arestes que arriben a u
- **grau de sortida** d'un node u: nombre d'arestes que surten de u

2.1.2 Operacions amb grafs

A continuació s'enumeren algunes de les operacions que es poden dur a terme amb grafs, les quals es detallen en els apartats següents

Operacions bàsiques

- Crear un graf buit
- Afegir (treure) nodes
- Afegir (treure) arestes
- Consultar l'ordre i la mida d'un graf
- Consultar els nodes i les arestes d'un graf
- Consultar el grau i els veïns d'un node

Algorismes sobre grafs

- Components connexos (*connected components*) d'un graf (*connected components*)
- Camins simples (*simple paths*) i camins mínims (*shortest paths*) d'un graf
- Recorreguts (*traversal*) d'un graf

També es veurà com llegir i guardar grafs de fitxers i com representar-los gràficament.

2.1.3 Grafs i Python

Python ofereix la biblioteca NetworX, en la que es disposa dels tipus de grafs esmentats a través de les classes següents:

- `Graph`: classe per un graf no dirigit.
- `DiGraph`: classe per un graf dirigit.
- `MultiGraph`: classe per un multigraf no dirigit.
- `MultiDiGraph`: classe per un multigraf dirigit.

Totes aquestes classes tenen els corresponents *atributs* i *mètodes*.

Python permet crear grafs de 3 maneres:

- Afegint nodes i arestes directament: d'un en un, a partir de llistes, ... (apartat 2.2)
- usant generadors de grafs: mètodes estàndard (apartat 2.14)

- Important dades de fitxers o altres fonts (apartat 2.6)

A més, la biblioteca NetworkX proporciona una col·lecció de funcions (*algorithms*) que apliquen els mètodes més habituals per grafs, com components connexos, camins, i recorreguts. En aquest document es descriuen alguns d'aquests algorismes; però cal consultar la documentació de la biblioteca NetworkX tant per veure de quins algorismes disposem com de la seva especificació exacta i actualitzada.

Tant per a l'ús adequat com per a la sintaxi, caldrà parar atenció en si utilitzem un atribut o mètode d'alguna de les 4 classes de grafs o bé una funció de la biblioteca. En el primer cas, cal usar la notació del punt amb el paràmetre corresponent al graf davant del punt. El segon cas és una crida a una funció on un dels paràmetres, normalment el primer, és el graf al que se li aplica el mètode.

Els nodes i arestes poden tenir etiquetes associades. Aquestes tenen un nom i un valor. En definir-les s'indica com `nom=valor` on nom és l'identificatiu de l'etiqueta i valor una expressió.

L'estructura dels grafs s'implementa com un diccionari de diccionaris de diccionaris (3 nivells):

- el diccionari (1r nivell) de l'estructura té com a claus els nodes i els valors són
- diccionaris (2n nivell) on les claus són els nodes veïns (arestes) i els valors són
- diccionaris (3r nivell) on les claus són els noms de les etiquetes de les arestes i els valors són els valors d'aquestes etiquetes

En els grafs dirigits hi ha dues estructures com la que s'acaba de descriure: una pels nodes successors i una altra pels nodes predecessors.

En els multigrafs, les diverses arestes entre dos nodes s'identifiquen nombres enters correlatius. Per representar-los, l'anterior estructura de diccionaris s'amplia amb un altre diccionari que s'insereix al 3r nivell i les etiquetes passen al 4t nivell (apartat 2.8).

Per treballar amb la biblioteca NetworkX, cal importar-la amb la comanda:

```
import networkx as nx
```

Als apartats següents es descriu l'operativitat bàsica per grafs.

2.2 Creació de grafs

Per crear un graf no dirigit, cal instanciar un objecte de la classe `Graph`, que crea un graf buit. A continuació el podem editar amb els mètodes de la classe:

- `add_node()`
- `add_edge()`
- `add_nodes_from()`
- `add_edges_from()`

- `remove_node()`
- `remove_edge()`
- `remove_nodes_from()`
- `remove_edges_from()`

També hi ha funcions que enriqueixen l'entrada de nodes i arestes en un graf com `add_path()` o `add_cycle()`. A partir d'una llista de nodes afegeixen el corresponent camí o cicle al graf.

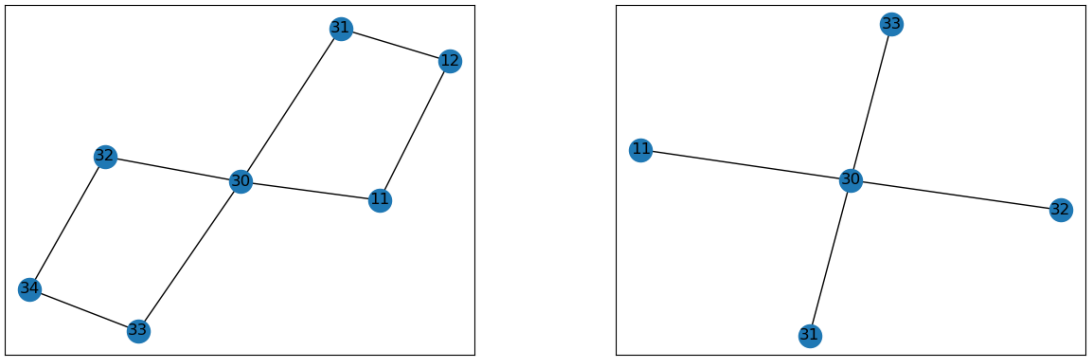

Figure 1: Graf `g` (esquerra) i subgraf `sg1` (dreta).

Exemples: (vegeu la figura 1)

```
###   creació d'un graf: afegim nodes i arestes
>>> g = nx.Graph()
>>> g.add_node(20)
>>> g.add_edge(20, 3)
>>> g.add_nodes_from([30, 31, 32, 33, 34])
>>> le = [(30, 31), (30, 32), (30, 33), (33, 34), (32, 34), (3, 30),
...       (3, 34), (20, 30)]
>>> g.add_edges_from(le)

###   actualització d'un graf: esborrem nodes i arestes
>>> g.remove_edge(3, 30)
>>> g.remove_edges_from([(20, 30), (3, 34)])
>>> g.remove_nodes_from([20, 3])
>>> list(g.nodes())
[30, 31, 32, 33, 34]
>>> list(g.edges())
[(30, 31), (30, 32), (30, 33), (32, 34), (33, 34)]

###  afegim un camí: és una funció, no un mètode
>>> nx.add_path(g, [30, 11, 12, 31])
>>> list(g.nodes())
[30, 31, 32, 33, 34, 11, 12]
>>> list(g.edges())
[(30, 31), (30, 32), (30, 33), (30, 11), (31, 12), (32, 34), (33, 34), (11,
   12)]
```

Subgrafs

Podem definir un subgraf a partir d'un graf seleccionant un subconjunt de nodes d'aquest: mètode `subgraph`. El subgraf estarà format pels nodes seleccionats i totes les arestes del graf inicial que connectin qualsevol parell de nodes del subgraf.

També es pot definir el subgraf seleccionant un subconjunt de les arestes del graf: mètode `edge_subgraph`. En aquest cas, el subgraf estarà format per les arestes seleccionades i tots els nodes connectats mitjançant aquestes arestes.

Exemples: (vegeu la figura 1)

```
>>> g = nx.Graph()
>>> g.add_nodes_from([30, 31, 32, 33, 34])
>>> g.add_edges_from([(30, 31), (30, 32), (30, 33), (33, 34), (32, 34)])
>>> nx.add_path(g, [30, 11, 12, 31])
>>> sg1 = g.subgraph([30, 31, 11, 32, 33]) # subgraf a partir de nodes de g
>>> sg1.nodes
NodeView((30, 31, 32, 33, 11))
>>> sg1.edges
EdgeView([(30, 31), (30, 32), (30, 33), (30, 11)])

# subgraf a partir d'arestes de g
>>> sg2 = g.edge_subgraph([(30, 31), (30, 32), (30, 11)])
>>> sg2.nodes
NodeView((30, 31, 32, 11))
```

2.3 Interrogació de grafs

Els nodes, arestes, veïns i graus es mostren usant *vistes* en lloc de contenidors. Una vista és un iterable i les vistes es modifiquen quan el graf es modifica. Els atributs `nodes` i `edges` així com els mètodes `nodes()` i `edges()` de la classe `Graph` (tenen el mateix nom) retornen una vista dels nodes i arestes del graf, respectivament.

Els mètodes més habituals de la classe `Graph` són:

- `nodes()`: tots els nodes
- `edges()`: totes les arestes
- `order()`, `number_of_nodes()`, `len()`: nombre de nodes
- `size()`, `number_of_edges()`: nombre d'arestes
- `degree(n)`: grau, nombre d'arestes veïnes de `n`
- `degree()`: diccionari amb el grau de tots els nodes
- `neighbors(n)`: iterador de nodes veïns a `n`.

Exemples:

```
>>> g.number_of_nodes()
7
>>> len(g), g.order()
(7, 7)
>>> g.number_of_edges()
8
>>> g.size()
8
>>> g.degree(30)
4
>>> g.degree()
DegreeView({30: 4, 31: 2, 32: 2, 33: 2, 34: 2, 11: 2, 12: 2})
>>> type(g.nodes)
<class 'networkx.classes.reportviews.NodeView'>
>>> g.nodes
NodeView((30, 31, 32, 33, 34, 11, 12))
>>> g.nodes()
NodeView((30, 31, 32, 33, 34, 11, 12))
>>> list(g.nodes())
[30, 31, 32, 33, 34, 11, 12]
>>> list(g.edges())
[(30, 31), (30, 32), (30, 33), (30, 11), (31, 12), (32, 34), (33, 34), (11,
    12)]

###   veïns
>>> a = g.neighbors(33)
>>> next(a)
30
>>> next(a)
34
>>> list(g.neighbors(33))
[30, 34]
```

Operadors de pertinença

Existeixen els mètodes `has_node()` i `has_edge()`. I es pot usar directament l'operador de pertinença: `node in g`. Recordem que els grafs estan implementats com diccionaris.

Exemples:

```
>>> g.has_node(30)
True
>>> g.has_node(10)
False
>>> g.has_edge(32, 34)
True
>>> g.has_edge(10, 11)
False
>>> 31 in g
True
>>> 10 in g
False
```

Recorreguts amb la sentència `for`

Els grafs són iterables: els nodes i arestes es poder recórrer amb la sentència `for`.

Exemples:

```
>>> for node in g:           # equival a for node in g.nodes():
...     print(node, end = ',')
...
30,31,32,33,34,11,12,

>>> for edge in g.edges():
...     print(edge, end= ',')
...
(30, 31),(30, 32),(30, 33),(30, 11),(31, 12),(32, 34),(33, 34),(11, 12),

>>> for node in g[30]:      # equival a   for node in g.neighbors(30):
...     print(node, end = ',')
...
31,32,33,11,
```

Operacions de conjunts en vistes

Les vistes es poden operar amb les operacions de conjunts i retornen un conjunt.

Exemples:

```
>>> g = nx.Graph()
>>> g.add_nodes_from([30, 31, 32, 33, 34])
>>> f = nx.Graph()
>>> f.add_nodes_from([27, 28, 29, 30, 31])
>>> a = g.nodes | f.nodes    # unió de conjunts
>>> a
{32, 33, 34, 27, 28, 29, 30, 31}
>>> type(a)
<class 'set'>
>>> g.nodes & f.nodes    # intersecció de conjunts
{30, 31}
>>> g.nodes - f.nodes    # diferència de conjunts
{32, 33, 34}
>>> g.nodes ^ f.nodes    # diferència exclusiva de conjunts (xor)
{32, 33, 34, 27, 28, 29}
```

2.4 Etiquetes

Els nodes i arestes d'un graf poden tenir etiquetes [2]. De les etiquetes n'hem de donar el nom i el valor. Les etiquetes es poden definir en el moment d'afegir el node o aresta corresponent o bé a posteriori.

[2] En lloc d'etiqueta també s'utilitza el terme atribut i a la documentació de NetworkX s'usa el terme attribute. En aquest text usem el terme etiqueta sobretot per no confondre aquest concepte amb el concepte d'atribut de les classes.

Exemples:

Suposem que `m` és un graf no dirigit:

```
>>> m = nx.Graph()
```

Afegim l'aresta `(3, 5)` amb una etiqueta de nom `pes` i valor `9` i un altra de nom `color` i valor `'blau'`. Observem que en la sintaxi de la crida al mètode `add_edge`, el nom de l'etiqueta no va entre cometes:

```
>>> m.add_edge(3, 5, pes = 9, color = 'blau')  # aresta amb dues etiquetes
```

Ara afegim l'aresta (10, 11) amb l'etiqueta de nom `color` amb valor `'verd'`, fent ús directe del diccionari de diccionaris de diccionaris amb el que es representa un graf, per indicar l'etiqueta. En la sintaxi que fa referència al diccionari de representació d'un graf, tant en aquest cas com quan es consulten les etiquetes i/o els seus valors, el nom de l'etiqueta, que és un string, ha d'anar entre cometes.

```
>>> m.add_edge(10, 11)
>>> m[10][11]['color'] = 'verd'    # afegim etiqueta a l'aresta (10, 11)
```

També podem afegir un camí amb una etiqueta per totes les arestes del camí:

```
>>> nx.add_path(m, [3, 4, 5, 6], pes = 8) # arestes d'un camí amb 1 etiqueta
```

Consultem els nodes i arestes del graf:

```
>>> m.nodes()
NodeView((3, 5, 10, 11, 4, 6))
>>> m.edges()
EdgeView([(3, 5), (3, 4), (5, 4), (5, 6), (10, 11)])
```

I ara observem l'estructura de diccionari de diccionaris de diccionaris que té aquest graf:

```
>>> for node in m:
...     print(node, m[node])
3 {5: {'pes': 9, 'color': 'blau'}, 4: {'pes': 8}}
5 {3: {'pes': 9, 'color': 'blau'}, 4: {'pes': 8}, 6: {'pes': 8}}
10 {11: {'color': 'verd'}}
11 {10: {'color': 'verd'}}
4 {3: {'pes': 8}, 5: {'pes': 8}}
6 {5: {'pes': 8}}
```

Consulta de les etiquetes d'una aresta:

```
>>> m[3][5]
{'pes': 9, 'color': 'blau'}
```

Les etiquetes de les arestes també es poden obtenir a partir de l'atribut `edges` de la classe `Graph` i usant directament l'estructura de diccionari dels grafs:

```
>>> m.edges[3, 4] # consulta de les etiquetes d'una aresta
{'pes': 8}
>>> m.edges[(3, 5)]['pes'] # consulta del valor d'una etiqueta d'una aresta
9
```

A la sintaxi anterior, `edges` és l'atribut, no el mètode (tenen el mateix nom): per això va seguit de claudàtors i no de parèntesis. En el segon exemple, l'aresta `(3, 5)` s'ha posat entre parèntesi per fer notar que està representada com un tuple.

Podem usar el mètode `edges()` amb el paràmetre `data = True` que retorna una vista de totes les arestes representades com 3-tuples `(node_inicial, node_final, dict_etiquetes)`:

```
>>> m.edges(data = True)
EdgeDataView([(3, 5, {'pes': 9, 'color': 'blau'}), (3, 4, {'pes': 8}),
(5, 4, {'pes': 8}), (5, 6, {'pes': 8}), (10, 11, {'color': 'verd'})])
```

Observem que a la sintaxi anterior, `edges` és el mètode.

Els nodes també poden portar etiquetes associades. En el següent exemple es mostra com afegir més nodes amb etiquetes i com posar etiquetes als nodes existents:

```
>>> m.add_node(20, tipus = 'gran', longitud = 2)
>>> m.add_nodes_from([21, 22], tipus = 'gran', longitud = 4)
>>> m.nodes[3]['tipus'] = 'petit'
>>> m.nodes[4]['tipus'] = 'petit'
>>> m.nodes[10]['longitud'] = 3
>>> m.nodes[11]['longitud'] = 5
```

Per afegir una etiqueta a un node (com hem vist als anteriors exemples) i també per consultar-la cal usar l'atribut `nodes` de la classe que representa el graf:

```
>>> m.nodes[21]
{'tipus': 'gran', 'longitud': 4}
>>> m.nodes[4]
{'tipus': 'petit'}
>>> m.nodes[11]
{'longitud': 5}
```

A continuació podem veure exemples de consulta d'etiquetes de nodes usant el paràmetre `data` del mètode `nodes`.

```
>>> m.nodes(data = 'tipus')
NodeDataView({3: 'petit', 5: None, 10: None, 11: None, 4: 'petit',
6: None, 20: 'gran', 21: 'gran', 22: 'gran'}, data='tipus')

>>> m.nodes(data = 'longitud')
NodeDataView({3: None, 5: None, 10: 3, 11: 5, 4: None, 6: None, 20: 2,
21: 4, 22: 4}, data='longitud')

>>> m.nodes(data = True)
NodeDataView({3: {'tipus': 'petit'}, 5: {}, 10: {'longitud': 3},
11: {'longitud': 5}, 4: {'tipus': 'petit'}, 6: {},
20: {'tipus': 'gran', 'longitud': 2}, 21: {'tipus': 'gran', 'longitud': 4},
22: {'tipus': 'gran', 'longitud': 4}})
```

Vegem finalment una altra forma d'entrar etiquetes amb el mètode `add_weighted_edges_from`:

```
>>> h = nx.Graph()
>>> h.add_weighted_edges_from([(1, 2, 5), (2, 3, 6)], weight = 'num')
>>> h[2][3]
{'num': 6}

>>> f = nx.Graph()
>>> l1 = [(1, 2, 'rosa'), (2, 3, 'blanc'), (3, 4, 'gris')]
>>> f.add_weighted_edges_from(l1, weight = 'color')
>>> f[2][3]
{'color': 'blanc'}
```

2.5 Dibuix de grafs

Per representar gràficament un graf, hem de crear primer la figura amb les funcions `nx.draw` o `nx.draw_networkx` entre altres. Després cal usar la biblioteca `matplotlib.pyplot`, que s'ha d'importar prèviament.

Les funcions de dibuix es troben a la documentació de `NetworkX` dins l'apartat **Drawing**. El següent exemple genera el gràfic que es mostra a la figura 2).

Exemple:

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> g = nx.Graph()
>>> g.add_nodes_from([30, 31, 32, 33, 34])
>>> g.add_edges_from([(30, 31), (30, 32), (30, 33), (33, 34), (32, 34)])
>>> nx.draw_networkx(g, with_labels = True) #
>>> plt.show()
```

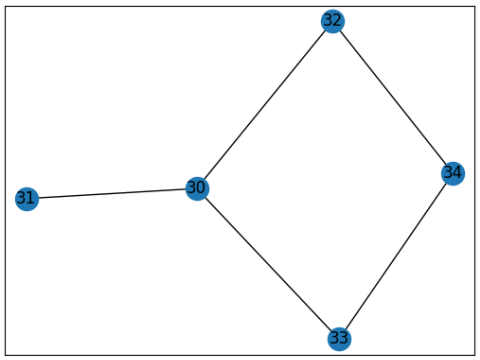

Figure 2: Graf corresponent a l'exemple de dibuix d'un graf.

2.6 Entrada/sortida de fitxers

GML (Graph Modelling Language) és el format proposat per grafs. Les comandes bàsiques per llegir i escriure usant aquest format són les següents:

```
g = nx.read_gml(nom_fitxer)
nx.write_gml(g, nom_fitxer)
```

Les funcions d'entrada/sortida es troben a la documentació de la biblioteca NetworkX dins l'apartat **Reading and writing graphs**.

2.7 Grafs dirigits: classe `DiGraph`

Un **graf dirigit** és aquell en que les arestes tenen orientació. Per crear un graf dirigit, podem usar els mateixos mètodes que pels grafs no dirigits.

Per interrogar grafs dirigits, també podem usar alguns mètodes dels grafs no dirigits i d'altres específics pels grafs dirigits. Com que les arestes tenen sentit, es pot distingir entre les que entren al node i les que en surten. Mètodes més habituals:

- `edges()`: totes les arestes
- `in_edges()`: totes les arestes que entren a algun node
- `out_edges()`: totes les arestes que surten d'algun node
- `degree(n)`: nombre d'arestes veïnes del node `n`
- `degree()`: diccionari per tots els nodes
- `out_degree(n)`: nombre d'arestes que surten del node `n`
- `in_degree(n)`: nombre d'arestes que arriben al node `n`
- `neighbors(n)`: iterador de nodes veïns a `n` (arestes que surten)
- `successors(n)`: el mateix que `neighbors(n)` (arestes que surten)
- `predecessors(n)`: iterador de nodes veïns a `n` (arestes que entren)

Exemples: (vegeu la figura 3)

```
>>> d = nx.DiGraph()
>>> d.add_nodes_from([1, 2, 3, 11, 12, 10])
>>> d.add_edges_from ([(1, 2), (2, 3), (3, 2), (3, 12), (12, 3), (11, 12),
... (10, 11), (1, 10), (1, 3), (3, 1), (2, 10), (10, 2), (12, 11), (2, 1)])
>>> d.order()
6
>>> d.size()
14
>>> d.degree(1)
5
>>> d.out_degree(1) # mètode de la classe
3
>>> d.out_degree[1] # atribut de la classe
3
>>> d.in_degree(1)
2
>>> list(d.neighbors(1))
[2, 10, 3]
>>> list(d.successors(1))
[2, 10, 3]
>>> d.out_edges(1)
OutEdgeDataView([(1, 2), (1, 10), (1, 3)])
>>> list(d.predecessors(1))
[3, 2]
>>> d.in_edges(1)
InEdgeDataView([(3, 1), (2, 1)])
```

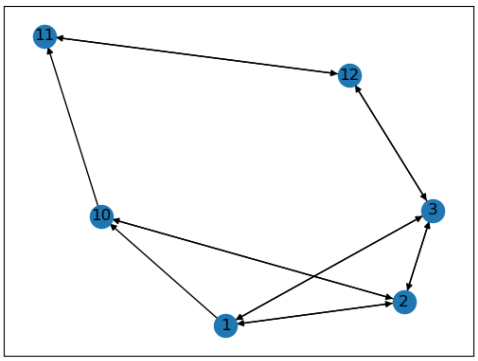

Figure 3: Graf dirigit `d`.

2.8 Multigrafs i multigrafs dirigits: classes `MultiGraph` i `MultiDiGraph`

Un **multigraf** és un graf no dirigit amb multiarestes, és a dir, més d'una aresta entre els mateixos dos nodes. Cadascuna d'aquestes arestes pot dur etiquetes associades.

Les múltiples arestes entre dos nodes s'identifiquen amb un enter. En els multigrafs l'estructura de dades és un diccionari de diccionaris de diccionaris de diccionaris (4 nivells):

L'estructura dels grafs s'implementa com un diccionari de diccionaris de diccionaris (3 nivells):

- el diccionari (1r nivell) de l'estructura té com a claus els nodes i els valors són
- diccionaris (2n nivell) on les claus són els nodes veïns (arestes) i els valors són
- diccionaris (3r nivell) on les claus són l'identificador d'aquella aresta (enter) i els valors són
- diccionaris (4t nivell) on les claus són els noms de les etiquetes de les arestes i els valors són els valors d'aquestes etiquetes

La notació per indexar una aresta és: `g[node1][node2][idx], idx in g[node1][node2]`.

I per accedir a una etiqueta: `g[node1][node2][idx]['nom_etiqueta']`.

Exemple: (vegeu la figura 4)

El següent graf representa un conjunt de persones (nodes) i la relació 'reunió': cada aresta indica un dia en què les dues persones s'han reunit en un mes determinat i té una etiqueta de nom 'dia' i amb un valor enter que indica el dia de la reunió.

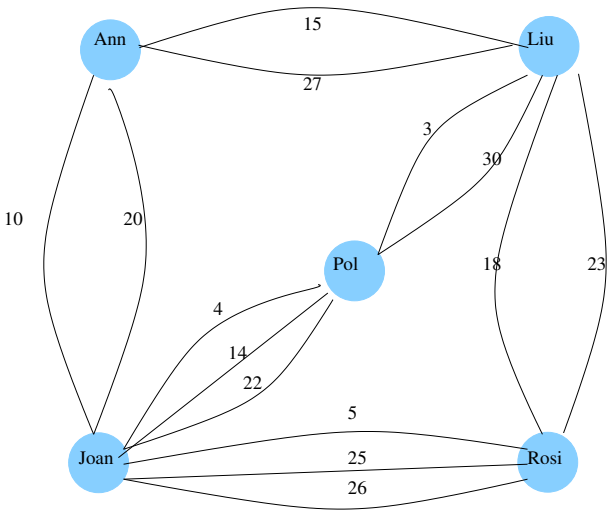

Figure 4: Multigraf `g`.

```
>>> g = nx.MultiGraph()
>>> nodes= ['Pol', 'Joan', 'Rosi', 'Liu', 'Ann']
>>> arestes=[('Pol', 'Joan', 4), ('Pol', 'Joan', 14),('Pol', 'Joan', 22),
...         ('Pol', 'Liu', 3),('Pol', 'Liu', 30),
...         ('Joan', 'Ann', 10),('Joan', 'Ann', 20),
...         ('Joan', 'Rosi', 5),('Joan', 'Rosi', 25),('Joan', 'Rosi', 26),
...         ('Rosi', 'Liu', 18),('Rosi', 'Liu', 23),
...         ('Liu', 'Ann', 15),('Liu', 'Ann', 27)]
>>> g.add_nodes_from(nodes)
>>> g.add_weighted_edges_from (arestes, weight = 'dia')
>>> g.degree()
MultiDegreeView({'Pol': 5, 'Joan': 8, 'Rosi': 5, 'Liu': 6, 'Ann': 4})
>>> g['Joan']
AdjacencyView({'Pol': {0: {'dia': 4}, 1: {'dia': 14}, 2: {'dia': 22}}, 'Ann':
    {0: {'dia': 10}, 1: {'dia': 20}}, 'Rosi': {0: {'dia': 5}, 1: {'dia':
   25}, 2: {'dia': 26}}})

>>> g['Joan']['Pol'] # dicc amb les arestes, (0, 1, 2), entre dos nodes
AtlasView({0: {'dia': 4}, 1: {'dia': 14}, 2: {'dia': 22}})
>>> g['Joan']['Pol'][0] # dicc d'etiquetes per l'aresta 0 entre dos nodes
{'dia': 4}
>>> g['Joan']['Pol'][1]
{'dia': 14}

# valor de l'etiqueta 'dia' per l'aresta 2 entre els nodes 'Joan' i 'Pol'
>>> g['Joan']['Pol'][2]['dia']
22
>>> for vei in g['Joan']:
...     for i in sorted(g['Joan'][vei]):
...         print (vei, i, g['Joan'][vei][i]['dia'])
...
Pol 0 4
Pol 1 14
Pol 2 22
Ann 0 10
Ann 1 20
Rosi 0 5
Rosi 1 25
Rosi 2 26
```

2.9 Algorismes per grafs

En els apartats següents es mostra una selecció dels algorismes més usuals aplicats a grafs. Aquests algorismes estan implementats a la biblioteca NetworkX com a funcions que, com a mínim, tenen un paràmetre de tipus graf. Els tipus d'algorismes que es veuen són:

- Components connexos
- Camins simples i camins mínims
- Recorreguts

2.10 Algorismes per components connexos

Un **component connex** d'un graf no dirigit g és un subgraf de g en el que hi ha un camí entre cada parell de nodes. Un component connex és una classe d'equivalència definida per la relació d'equivalència sobre els nodes: aRb = *"hi ha camí entre el node a i el node b"*. Un graf no dirigit és connex quan només té un component connex.

Les funcions per components connexos de grafs no dirigits són a la documentació de NetworkX dins l'apartat **Algorithms: Components**. Les següents funcions són per grafs NO dirigits:

```
is_connected(g)
     Retorna True si el graf és connectat, False altrament.

number_connected_components(g)
     Retorna el nombre de components connexos.

connected_components(g)
     genera components connexos: és un generador

node_connected_component(g, n)
     Retorna el conjunt de nodes en el component del graf que conté el node n
```

Exemples:(vegeu la figura 5)

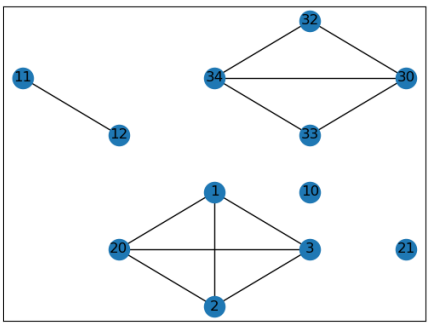

Figure 5: Graf g.

```
>>> g = nx.Graph()
>>> g.add_nodes_from([1, 2, 3, 20, 11, 12, 10, 21, 30, 32, 33, 34])
>>> le = [(1, 2), (2, 3), (3, 20), (33, 34), (32, 34), (32, 30), (11, 12),
...       (30, 33), (30, 34), (1, 3), (2, 20), (1, 20)]
>>> g.add_edges_from (le)
>>> nx.is_connected(g)
False
>>> nx.number_connected_components(g)
5
>>> type(nx.connected_components(g))
<class 'generator'>
>>> a = nx.connected_components(g)  # generador de components connexos (sets)
>>> next(a)       # 1r component connex (un conjunt)
{1, 2, 3, 20}
>>> next(a)       # 2n component connex
{11, 12}
>>> next(a)       # 3r component connex
{10}
>>> next(a)       # ...
{21}
>>> list(next(a)) # darrer component connex convertit a llista
[32, 33, 34, 30]
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list(nx.connected_components(g)) # llista amb els components connexos
[{1, 2, 3, 20}, {11, 12}, {10}, {21}, {32, 33, 34, 30}]

# llista amb longituds dels components connexos en ordre descendent
>>> longituds_components = []
>>> for c in sorted(nx.connected_components(g), key=len, reverse=True):
...   longituds_components.append(len(c))
...
>>> longituds_components
[4, 4, 2, 1, 1]

>>> nx.node_connected_component(g, 2)  # component connex amb el node 2
{1, 2, 3, 20}
>>> nx.node_connected_component(g, 10) # component connex amb el node 10
{10}
>>> nx.node_connected_component(g, 11) # component connex amb el node 11
{11, 12}
```

La definició de connex és més complexa en grafs dirigits, en els quals es distingeix entre feblement connex i fortament connex. Un graf dirigit és **feblement connex** si canviant totes les arestes dirigides per arestes no dirigides, el resultat és un graf connex. Un graf dirigit és **fortament connex** si per cada parell de nodes u i v existeix un camí dirigit entre u i v i un camí dirigit entre v i u .

Funcions equivalents per connectivitat forta i feble en grafs dirigits es troben a la documentació de NetworkX dins l'apartat **Algorithms: Components**.

2.11 Algorismes per nodes aïllats

Un node **aïllat** és un node que no té cap veí. Ell sol constitueix un component connex.

Les funcions per components aïllades es troben a la documentació de NetworkX dins l'apartat **Algorithms: Isolates**.

Disposem de les funcions següents:

```
is_isolate(g, n)
      Determina si un node és aïllat o no.

isolates(g)
      Iterador sobre els nodes aïllats del graf.

number_of_isolates()
      Retorna el nombre de nodes aïllats del graf.
```

Exemples:(vegeu la figura 5)

```
>>> type(nx.isolates(g))  # components aïllades
<class 'generator'>
>>> list(nx.isolates(g))
[10, 21]
```

2.12 Algorismes per camins simples i camins mínims

Un **camí** (*path*) entre dos nodes u i v és una successió d'arestes que uneixen una successió de nodes que porten de u a v.

Un **camí simple** (*simple path*) entre dos nodes u i v és un camí entre u i v on cada node només hi apareix una vegada.

La **longitud d'un camí** és el nombre d'arestes del camí. Si al camí hi ha n nodes, la longitud és $n-1$. Un camí amb un node té una longitud de 0.

Un **cicle** o bucle és un camí que comença i acaba en el mateix node.

El **camí mínim** (*shortest path*) entre dos nodes u i v és, de tots els camins de u a v, el que té menys arestes (o que minimitza algun altre criteri associat a les etiquetes de les arestes).

Les funcions per camins simples es troben a la documentació de NetworkX dins l'apartat **Algorithms: Simple Paths**. Disposem de les següents funcions per camins simples:

```
all_simple_paths(g, origen, destí)
     Genera tots els camins simples del graf g desde origen a destí.

is_simple_path(g, llista_nodes)
     Retorna True si i només si els nodes donats formen un camí simple a g.

shortest_simple_paths(g, origen, destí)
     Genera tots els camins simples en el graf g desde origen a destí
     Comença pels més curts
```

Exemples: (vegeu la figura 6)

```
>>> cs = nx.Graph()
>>> cs.add_nodes_from([1, 2, 3, 20, 11, 12, 10, 21, 30, 32, 33, 34])
>>> le = [(1, 2), (2, 3), (3, 20), (33, 34), (32, 34), (32, 30), (11, 12),
...       (10, 11), (1, 10), (2, 10), (3, 12), (30, 33)]
>>> cs.add_edges_from (le)
>>> list(nx.all_simple_paths(cs, 1, 3))
[[1, 2, 3], [1, 2, 10, 11, 12, 3], [1, 10, 11, 12, 3], [1, 10, 2, 3]]
>>> list(nx.all_simple_paths(cs, 1, 3))[1]  # 1r element de la llista
[1, 2, 10, 11, 12, 3]
>>> nx.is_simple_path(cs, [1, 2, 10, 11, 12, 3])
True
>>> nx.is_simple_path(cs, [1, 10, 12])
False
```

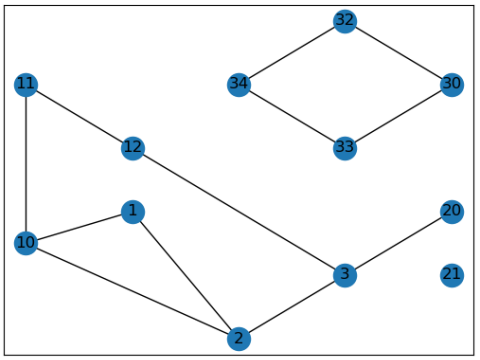

Figure 6: Graf cs.

Les funcions per camins mínims es troben a la documentació de NetworkX dins l'apartat **Algorithms: Shortest Paths**. Disposem de les següents funcions per camins mínims:

```
shortest_path(g[, origen, destí, weight, ...)
      Calcula tots els possibles camins mínims en el graf g.
      Si origen i destí s'especifiquen, calcula només un camí mínim
      que té aquest origen i destí i retorna una llista de nodes.
      Si origen i/o destí no s'especifiquen, retorna un diccionari

all_shortest_paths(g, origen, destí[, ...)
      Com l'anterior.
      És un generador de tots els camins mínims entre origen i destí.

shortest_path_length(g[, origen, destí, ...)
      Calcula la longitud dels camins mínims en el graf g.

average_shortest_path_length(g)
      Retorna la longitud mitjana dels camins mínims de g.

has_path(g, origen, destí)
      Retorna True si g té un camí que va d'origen a destí.
```

Exemples: (vegeu la figura 6)

```
>>> cs = nx.Graph()
>>> cs.add_nodes_from([1, 2, 3, 20, 11, 12, 10, 21, 30, 32, 33, 34])
>>> le = [(1, 2), (2, 3), (3, 20), (33, 34), (32, 34), (32, 30), (11, 12),
...       (10, 11), (1, 10), (2, 10), (3, 12), (30, 33)]
>>> cs.add_edges_from (le)

>>> nx.shortest_path (cs, 1, 3)
[1, 2, 3]
>>> nx.shortest_path (cs, 1, 12)
[1, 2, 3, 12]
>>> list(nx.all_shortest_paths (cs, 1, 3))
[[1, 2, 3]]
>>> list(nx.all_shortest_paths (cs, 1, 12))
[[1, 2, 3, 12], [1, 10, 11, 12]]
>>> nx.shortest_path_length (cs, 1, 3)
2
>>> nx.shortest_path_length (cs, 1, 12)
3
>>> nx.has_path (cs, 1, 3)
True
>>> nx.has_path (cs, 1, 12)
True
>>> nx.has_path (cs, 1, 30)
False
```

2.13 Algorismes per recorreguts en profunditat i en amplada

Permeten recórrer els nodes d'un component connex d'un graf, partint d'un node donat o, el que és el mateix, per obtenir tots els nodes connectats amb un node donat, es pot aplicar un d'aquests dos esquemes:

- recorregut en profunditat (*depth-first search, dfs*)
- recorregut en amplada (*breath-first search, bfs*)

Tots dos esquemes es basen en recórrer recursivament els veïns del node donat i obtenen el mateix conjunt de nodes.

El **recorregut en profunditat** consisteix, a partir del node origen, a expandir-lo de forma recursiva: primer el primer veí (fill), després el primer fill d'aquest, i així successivament. Quan s'han visitat tots els fills d'un node, es puja un nivell i es continua amb el següent germà i els seus fills. És a dir, donat un node, es visiten primer els seus veïns (fills) abans que els seus germans.

El **recorregut en amplada** consisteix, a partir del node origen, a visitar els nodes d'una generació de veïns; a continuació els de la següent generació i així successivament. És a dir, donat un node, es visiten primer els seus germans abans que els seus fills.

Disposem de les següents funcions, entre altres, per recorreguts en profunditat:

```
dfs_edges(g, origen)
      Fa un recorregut en profunditat pels nodes de g començant per origen
      i retorna un iterador de les arestes d'aquest recorregut
      en l'ordre en què s'han obtingut.

dfs_tree(g, origen)
      Retorna un arbre (classe DiGraph) obtingut a partir del
      recorregut en profunditat pels nodes de g començant per origen
```

Disposem de funcions similars per recorreguts en amplada.

Les funcions per recorreguts es troben a la documentació de NetworkX dins l'apartat **Algorithms: Traversal**.

Aquests recorreguts interpreten el graf com un arbre (que es representa com un graf dirigit). Els nodes d'un arbre es poden mostrar diversos ordres: **preordre** indica que primer es mostra un node abans que els seus fills i **postordre** és l'invers.

Exemples:

```
>>> r = nx.Graph()
>>> r.add_nodes_from([1, 2, 3, 20, 11, 12, 10, 21, 30, 32, 33, 34])
>>> larestes = [(1, 2), (2, 3), (3, 20), (33, 34), (32, 34), (32, 30),
...           (11, 12), (10, 11), (1, 10), (2, 10), (3, 12), (30, 33)]
>>> r.add_edges_from (larestes)

## recorregut en profunditat (depth first search, dfs)
>>> list(nx.dfs_edges(r, 1))
[(1, 2), (2, 3), (3, 20), (3, 12), (12, 11), (11, 10)]

## nodes en preordre i postordre
>>> list(nx.dfs_preorder_nodes(r, 1))
[1, 2, 3, 20, 12, 11, 10]
>>> list(nx.dfs_postorder_nodes(r, 1))
[20, 10, 11, 12, 3, 2, 1]

## predecessors i successors
>>> nx.dfs_predecessors(r, 1) # dicc dels predecessors en dfs del node 1
{2: 1, 3: 2, 20: 3, 12: 3, 11: 12, 10: 11}
>>> nx.dfs_successors(r, 1)   # dicc dels successors en dfs del node 1
{1: [2], 2: [3], 3: [20, 12], 12: [11], 11: [10]}

## arbre dfs representat com un DiGraph
>>> trd = nx.dfs_tree(r, 1)
>>> trd.nodes()
NodeView((1, 2, 3, 20, 12, 11, 10))
>>> trd.edges()
OutEdgeView([(1, 2), (2, 3), (3, 20), (3, 12), (12, 11), (11, 10)])

## recorregut en amplada (breath first search, bfs)
>>> list(nx.bfs_edges(r, 1))
[(1, 2), (1, 10), (2, 3), (10, 11), (3, 20), (3, 12)]

## predecessors i successors en bfs del node 1: retorna un iterador
>>> list(nx.bfs_predecessors(r, 1))
[(2, 1), (10, 1), (3, 2), (11, 10), (20, 3), (12, 3)]
```

```
>>> list(nx.bfs_successors(r, 1))
[(1, [2, 10]), (2, [3]), (10, [11]), (3, [20, 12])]

# arbre bfs representat com un DiGraph
>>> trb = nx.bfs_tree(r, 1)
>>> trb.nodes()
NodeView((1, 2, 10, 3, 11, 20, 12))
>>> trb.edges()
OutEdgeView([(1, 2), (1, 10), (2, 3), (10, 11), (3, 20), (3, 12)])
```

El graf `r` d'aquest exemple és el mateix que el graf `cs` de la figura 6). A l'exemple es calculen els arbres obtinguts a partir d'un recorregut en profunditat (`trd`) i d'un recorregut en amplada (`trb`), partint del node `1`, que es mostren a la figura 7.

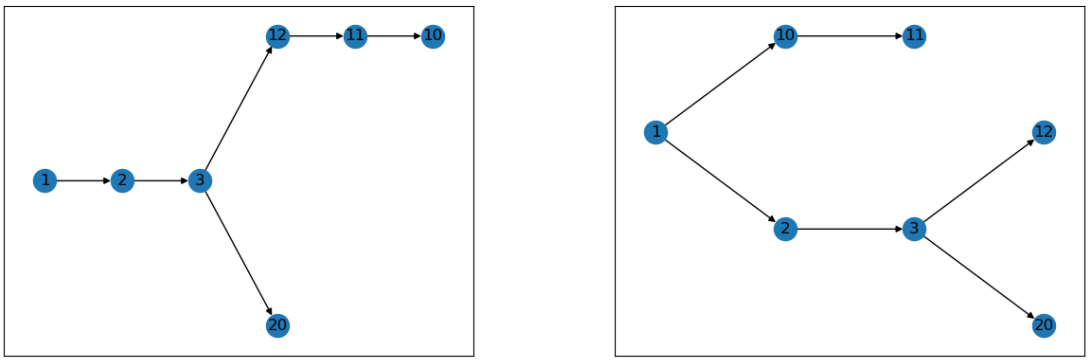

Figure 7: Arbres corresponents al recorregut en profunditat, dfs (esquerra) i al recorregut en amplada, bfs (dreta) del graf `r` mostrat a la figura 6, partint del node `1`.

2.14 Generadors de grafs

NetworkX disposa de funcions que permeten generar diferents tipus de grafs. Als exemples que es mostren a continuació es generen els grafs:

- *complet*: a partir d'un enter n, retorna un graf amb n nodes, amb valors $i \in [0, n-1]$ i totes les possibles arestes entre ells. El nombre d'arestes és: $C_{n,2} = n(n-1)/2$. Vegeu la figura 8 (esquerra).
- *hipercub*: és un graf que representa un hipercub de dimensió n. El graf retornat té 2^n nodes amb valors que són 3-tuples indicant les coordenades dels nodes suposant que l'hipercub està situat amb el vèrtex inferior a l'origen de coordenades i de mida unitària. El nombre d'arestes (cubs de dimensió 1) és $2^{n-1}n$. L'hipercub amb $n = 2$ és un quadrat, amb 4 nodes i 4 arestes, i el de $n = 3$ és un cub, amb 8 nodes i 12 arestes. L'hipercub de dimensió 4 té 16 nodes i 32 arestes. Vegeu la figura 8 (dreta).

Les funcions per generar grafs es troben a la documentació de NetworkX dins l'apartat **Graph generators**.

Exemples: (vegeu figura 8)

```
>>> complet4 = nx.complete_graph(4)
>>> complet4.nodes()
NodeView((0, 1, 2, 3))
>>> complet4.edges()
EdgeView([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)])

>>> hipercub = nx.hypercube_graph(3)
>>> lnodes = [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
...           (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> le=[((0, 0, 0), (1, 0, 0)),((0, 0, 0), (0, 1, 0)),((0, 0, 0), (0, 0, 1)),
...     ((0, 0, 1), (1, 0, 1)),((0, 0, 1), (0, 1, 1)),((0, 1, 0), (1, 1, 0)),
...     ((0, 1, 0), (0, 1, 1)),((0, 1, 1), (1, 1, 1)),((1, 0, 0), (1, 1, 0)),
...     ((1, 0, 0), (1, 0, 1)),((1, 0, 1), (1, 1, 1)),((1, 1, 0), (1, 1, 1))]
>>> list(hipercub.nodes()) == lnodes
True
>>> list(hipercub.edges()) == le
True
```

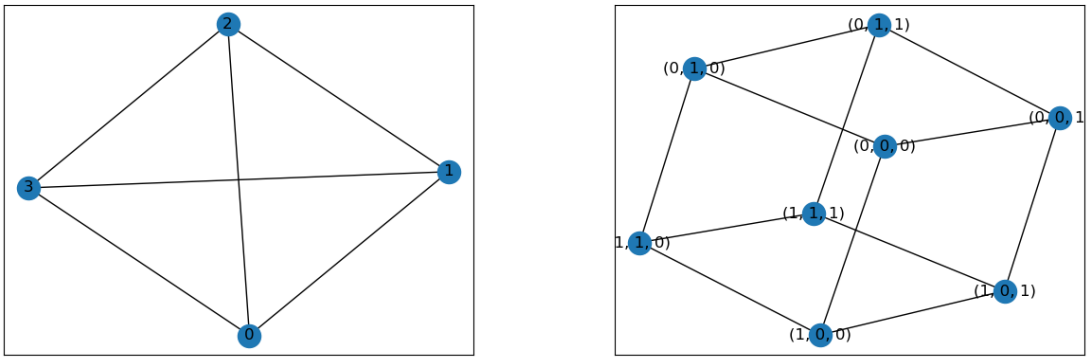

Figure 8: Graf complet per n=4 (esquerra) i graf d'un hipercub de dimensió 3 (dreta).