UNIVERSITÀ di VERONA

Linguaggio di Programmazione Python

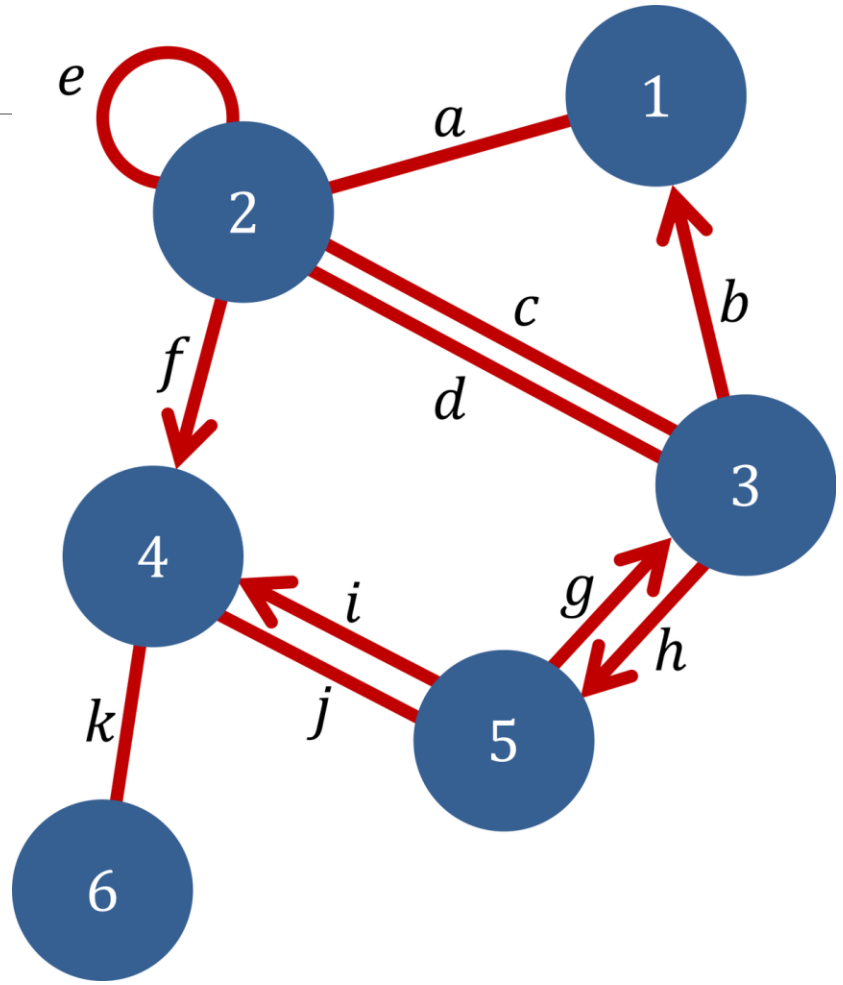
23/05/2022

Giulio Mazzi

giulio.mazzi@univr.it

Grafi - Cenni

- I Grafi sono strutture dati estremamente utili, usati per rappresentare relazioni tra oggetti
- Un **grafo** è composto da una collezione di **nodi** e **archi**.
- I nodi possono rappresentare oggetti di ogni tipo
- Gli archi possono essere direzionali e avere peso, connettono tra loro due (o più) oggetti
- Usati in moltissimi ambiti: analisi di social network, contact tracing, finanza, biologia, ecc...



Networkx

Python offre più di una libreria per lavorare con I grafi, oggi vedremo NetworkX.

NetworkX è un pacchetto python che serve a creare, manipolare e studiare grafi anche molto complessi. Offre moltissimi algoritmi standard, generatori per grafi noti, strumenti di analisi e misura, e molto altro.

Supporta grafi di diverso tipo. I suoi grafi sono estremamente flessibili: "tutto" può essere un nodo. Posso associare attributi a piacere a nodi e archi!

Documentazione: https://networkx.org/documentation/latest/reference/glossary.html

Costruire un grafo

Possiamo creare un grafo usando la classe Graph (o DiGraph, per grafi diretti).

Possiamo esplicitare I nodi, oppure aggiungere direttamente gli archi (questo crea in automatico un nodo, se non esiste!)

```python
import networkx as nx
```

```python
G = nx.Graph()
```

```python
G = nx.DiGraph()
```

```python
G = nx.Graph()
G.add_node(1)
```

```python
# non serve introdurre il nodo esplicitamente
G.add_edge(1, 2)
```

```python
# posso specificare paramentri, tra cui il peso
G.add_edge(1, 2, weight = 0.9)
```

Nodi

Molto spesso usiamo numeri o stringhe per identificare I nodi... ma possiamo fare molto di più.

In NetworkX, ogni oggetto hashable può essere un nodo. Nello specifico, non esiste un tipo nodo, ma il grafo contiene semplicemente delle collezioni di oggetti che fungono da nodi.

```
# uso stringhe
G = nx.Graph()
elist = [('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
G.add_weighted_edges_from(elist)

print(G.nodes)
```

```
['a', 'b', 'c', 'd']
```

```
# in questo caso, i nodi sono numeri, ma con attributi
G = nx.Graph()
G.add_nodes_from([
    (4, {"color": "red"}),
    (5, {"color": "green"}),

])

print(G.nodes)
```

```
[4, 5]
```

Graph Reporting

Alcuni dei parametri più importanti sono I seguenti: **nodes** (la lista dei nodi), **edges** (lista archi), **adj** (dizionario che mappa ogni nodo ai successori), **degree** (numero di archi uscenti da ogni nodo).

NetworkX usa liste di adiacenza per rappresentare i grafi. Di fatto, possiamo pensarli come a "dizionari di dizionari".

```
G.clear()
G.add_edges_from([('A', 'B'), (['A', 'C']), (['C', 'D'])])
```

```
# funzioni utili per interrogare il grafo:
print(G.nodes)
print(G.edges)
print(G.adj)
print(G.degree)
```

```
['A', 'B', 'C', 'D']
[('A', 'B'), ('A', 'C'), ('C', 'D')]
{'A': {'B': {}, 'C': {}}, 'B': {'A': {}}, 'C': {'A': {}, 'D': {}}, 'D': {'C': {}}}
[('A', 2), ('B', 1), ('C', 2), ('D', 1)]
```

```
# iterare tutti gli archi
for u, succ in G.adj.items():
    for v in succ:
        print(f"{u} -> {v}")
```

```
A -> B
A -> C
B -> A
C -> A
C -> D
D -> C
```

Attributi

Un feature che rende NetworkX particolarmente utile è quella di aggiungere attributi ad archi e nodi.

Alcuni attributi sono definiti di default, in particolare weight, che deve sempre essere di tipo numerico.

I nostri algoritmi possono sfruttare queste etichette in vari modi.

```
import math
G.add_edge('y', 'x', function=math.cos)
```

```
G = nx.Graph([(1, 2, {"color": "yellow"})])
G[1]
```

```
AtlasView({2: {'color': 'yellow'}})
```

```
G.add_edge(1, 3)
G[1][3]['color'] = "blue"
G.edges[1, 2]['color'] = "red"
G.edges[1, 2]
```

```
{'color': 'red'}
```

Attributi - Esempio

```
FG = nx.Graph()
FG.add_weighted_edges_from([(1, 2, 0.125), (1, 3, 0.75), (2, 4, 1.2), (3, 4, 0.375)])
# Posso scorrere tutti gli adiacenti con adj.items()
for n, nbrs in FG.adj.items():
    # in questo caso, uso gli adiacenti come un altro dizionario
    for nbr, eattr in nbrs.items():
        wt = eattr['weight']
        if wt < 0.5:
            print(f"({n}, {nbr}, {wt:.3})")
```

```
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

```
for (u, v, col) in G.edges.data('color'):
    if col == "red":
        print(f"({u}, {v}, {wt:.3})")
```

```
(1, 2, red)
```

Graph generators

NetworkX offre moltissimi generatori di default. Questi possono essere usati per costruire grafi che seguono specifiche regole.

Questi metodi spesso supportano parametri e/o randomicità. Potete trovare la lista completa qui: https://networkx.org/documentation/latest/reference/generators.html

```
G = nx.sedgewick_maze_graph()
print(G.edges)
```

```
[(0, 2), (0, 7), (0, 5), (1, 7), (2, 6), (3, 4), (3, 5), (4, 5), (4, 7), (4, 6)]
```

```
G = nx.complete_bipartite_graph([1, 2, 3], ['A', 'B'])
print(G.edges)
```

```
[(1, 'A'), (1, 'B'), (2, 'A'), (2, 'B'), (3, 'A'), (3, 'B')]
```

```
G = nx.florentine_families_graph()
print(G.edges)
```

```
[('Acciaiuoli', 'Medici'), ('Medici', 'Barbadori'), ('Medici', 'Ridolfi'), ('Medici', 'Tornabuoni'), ('Medici', 'Albiz
zi'), ('Medici', 'Salviati'), ('Castellani', 'Peruzzi'), ('Castellani', 'Strozzi'), ('Castellani', 'Barbadori'), ('Per
uzzi', 'Strozzi'), ('Peruzzi', 'Bischeri'), ('Strozzi', 'Ridolfi'), ('Strozzi', 'Bischeri'), ('Ridolfi', 'Tornabuoni
'), ('Tornabuoni', 'Guadagni'), ('Albizzi', 'Ginori'), ('Albizzi', 'Guadagni'), ('Salviati', 'Pazzi'), ('Bischeri', 'G
uadagni'), ('Guadagni', 'Lamberteschi')]
```

Algoritmi

Il cuore di NetworkX è la sua collezione di algoritmi pre-implementati. Il pacchetto copre moltissimi algoritmi noti, tra cui: clustering, colorabilità, clique, centralità, flusso, isomorfismo, cammini minimi, copertura, …

Ci sono anche algoritmi pensati per operare su grafi specifici, ad esempio DAG, grafi bipartiti, alberi, ...

Potete trovare una lista completa qui:
https://networkx.org/documentation/latest/reference/algorithms/index.html

```
G = nx.sedgewick_maze_graph()
print(G.edges)
```

```
[(0, 2), (0, 7), (0, 5), (1, 7), (2, 6), (3, 4), (3, 5), (4, 5), (4, 7), (4, 6)]
```

```
print(nx.shortest_path(G,3,7))
```

```
[3, 4, 7]
```

Rappresentazione

Possiamo stampare la rappresentazione grafica dei nostri grafi. NetworkX sfrutta matplotlib, dot e graphviz per creare disegni.

```
import matplotlib.pyplot as plt
```

```
G = nx.florentine_families_graph()
nx.draw(G, with_labels=True, font_weight='bold')
```

Grazie a dot (https://www.graphviz.org/doc/info/lang.html), possiamo personalizzare nel dettaglio le nostre stampe.