

RICERCHE SU GRAFI:

!! PRIMA DI CERCARE LA COMPONENTE CONNESSA ASSICURATI CHE IL GRAFO POSSEGGA QUEL NODO

Nx.MultiGrap() se non ho grafo semplice

Nx.MultiDiGrap() se ho un multigrafo diretto

<https://networkx.org/documentation/stable/reference/classes/index.html>

funzioni predefinite

PRENDO ARCHI INCIDENTI A UN NODO

(caso grafo non orientato)

- **archi = self._graph.edges(node, data=True)** serve a **ottenere tutti gli archi (edges)** connessi a un nodo specifico in un grafo, **includendo anche i dati associati a quegli archi**.

(caso grafo orientato)

- **out = self._graph.out_edges(node, data=True)** prendo gli archi uscenti da un grafo orientato
- **en = self._graph.in_edges(node, data=True)** prendo gli archi entranti da un grafo orientato

ORDINAMENTI

- **volumi.sort(key=lambda x:x[1], reverse=True)** ordino una lista di tuple in maniera decrescente sulla base del secondo valore

FUNZIONI "RICERCA"

- **tree = nx.dfs_tree(self._graph, p)** restituisce un **albero orientato** costruito seguendo una **visita in profondità** partendo da p. Il grafo di partenza può essere orientato o meno, ma il ritorno sarà sempre orientato.

CAMMINO MINIMO

(caso grafo non orientato)

- **path = nx.shortest_path(G, source=source, target=target)**: Se vuoi il cammino più breve IN TERMINI DI NUMERO DI ARCHI **da un nodo source a un nodo target specifico**. E' una lista contenente i nodi nell'ordine in cui devi visitarli per andare da source a target lungo il cammino più breve.
- **path = nx.shortest_path(G, source=source, target=target, weight='weight')**: Se vuoi il cammino più breve USANDO I PESI **da un nodo source a un nodo target specifico**. E' una lista contenente i nodi nell'ordine in cui devi visitarli per andare da source a target lungo il cammino più breve.
- **paths = nx.single_source_shortest_path(G, source)**: Se vuoi i cammini più brevi IN TERMINI DI NUMERO DI ARCHI da source a **tutti i nodi raggiungibili**. paths sarà un dizionario: chiave = nodo raggiunto, valore = lista di nodi che costituisce il percorso più breve da source a quel nodo
- **paths = nx.single_source_shortest_path(G, source, weight='weight')**: Se vuoi i cammini più brevi IN TERMINI DI PESI da source a **tutti i nodi raggiungibili**. paths sarà un dizionario: chiave = nodo raggiunto, valore = lista di nodi che costituisce il percorso più breve da source a quel nodo
-

- **BFS utile per ricerca cammino che minimizza num mi archi (non cammino minimo!) e componenti connesse**

Camm minimo da tutti i nodi a tutti i nodi: dijkstra ripetuto

(Dijkstra ripetuto)

```
distanze = {}

for nodo in G.nodes:
    distanze[nodo] = nx.single_source_dijkstra_path_length(G, nodo)

# Stampa le distanze minime da ogni nodo
for sorgente, mappa in distanze.items():
    print(f"Da {sorgente}: {mappa}")
```

(caso grafo non orientato)

RENDI IL GRAFO EULERIANO AGGIUNGENDO IL MINIMO NUMERO DI ARCHI: `eulerize(graph)`

algoritmi utili

TROVA COMPONENTE CONNESSA:

(caso grafo non orientato)

```
def getNumCompConnesse(self):
    # S = [self._graph.subgraph(c).copy() for c in nx.connected_components(self._graph)]
    #this is to get all the components

    return nx.number_connected_components(self._graph)
```

- **`nx.connected_components`** restituisce tutte le componenti connesse di un grafo non orientato, ogni componente è un **insieme di nodi** che sono **mutuamente raggiungibili** tra loro ma **non sono connessi** a nodi di altre componenti

Operazione	Risultato	Cosa contiene	Utilità
<code>list(nx.connected_components(G))</code>	Lista di <code>set</code>	Solo nodi	Leggero, ma poco usabile
<code>[G.subgraph(c).copy() for c in ...]</code>	Lista di <code>Graph</code>	Nodi + archi + attributi	Pronto per analisi, disegno, modifica

-

(caso grafo orientato)

- **`nx.number_weakly_connected_components(self._graph)`** Numero di componenti **debolmente connesse** (ignora direzione archi)

- `nx.number_strongly_connected_components(self._graph)` Numero di componenti **fortemente connesse** (seguendo la direzione)
- `nx.weakly_connected_components(G)` Restituisce un **generatore** di insiemi di nodi, dove ogni insieme rappresenta una **componente connessa** del grafo.
- `nx.strongly_connected_components(G)`

TROVA LA COMPONENTE CONNESSA PIU' GRANDE

- `max_component = max(nx.connected_components(G), key=len)` PER TROVARE TUTTI I NODI DELLA COMPONENTE CONNESSA PIU' GRANDE
- `S = [self._graph.subgraph(c).copy() for c in nx.connected_components(self._graph)]`
`max_subgraph = max(S, key=lambda g: g.number_of_nodes())` TROVARE TUTTO IL SOTTOGRAFO DELLA COMPONENTE CONNESSA PIU' GRANDE

SOMMA DEI PESI DI TUTTI GLI ARCHI INCIDENTI A UN NODO – ordinato per valore decrescente

```
volumi = []

for node in self._graph.nodes():
    volume = 0
    vicini = list(nx.neighbors(self._graph, node))

    for vicino in vicini:
        volume += self._graph[node][vicino]['weight']

    volumi.append((node, volume))

volumi.sort(key=lambda x:x[1], reverse=True)
return volumi
```

CAMINO PIU' LUNGO PARTENDO DA UN NODO

```
def cercaPercorso(self, partenza):
    p = self._idMap[(int(partenza))]
    tree = nx.dfs_tree(self._graph, p)
```

```

nodes = list(tree.nodes())

sol = []

for node in nodes:

    temp = [node]

    while (temp[0] != p):

        pred = nx.predecessor(tree, p, temp[0])
        temp.insert(0, pred[0])

    if len(temp) > len(sol):

        sol = copy.deepcopy(temp)

return sol

```

uso tutti i vertici -> ciclo hamiltoniano (travelling salesman problem)

uso tutti i nodi -> ciclo euleriano, uso l'algoritmo di Hierholzer SE TUTTI I NODI HANNO UN NUMERO PARI DI ARCHI

hamiltonian_path

`hamiltonian_path(G)` [\[source\]](#)

Returns a Hamiltonian path in the given tournament graph.

Each tournament has a Hamiltonian path. If furthermore, the tournament is strongly connected, then the returned Hamiltonian path is a Hamiltonian cycle (by joining the endpoints of the path).

Parameters:

- G** : NetworkX graph
- A directed graph representing a **tournament**.

Returns:

- path** : list
- A list of nodes which form a Hamiltonian path in **G**.

Notes

This is a recursive implementation with an asymptotic running time of $O(n^2)$, ignoring multiplicative polylogarithmic factors, where n is the number of nodes in the graph.

Examples

```

>>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)])
>>> nx.is_tournament(G)
True
>>> nx.tournament.hamiltonian_path(G)
[0, 1, 2, 3]

```

Eulerian

Eulerian circuits and graphs.

<code>is_eulerian</code> (G)	Returns True if and only if G is Eulerian.
<code>eulerian_circuit</code> (G[, source, keys])	Returns an iterator over the edges of an Eulerian circuit in G .
<code>eulerize</code> (G)	Transforms a graph into an Eulerian graph.
<code>is_semieulerian</code> (G)	Return True iff G is semi-Eulerian.
<code>has_eulerian_path</code> (G[, source])	Return True iff G has an Eulerian path.
<code>eulerian_path</code> (G[, source, keys])	Return an iterator over the edges of an Eulerian path in G .