Progetto in Python

Gestione di un grafo

Il progetto consta dei file functions.py e graph.py, che descriveremo in questo documento, oltre che un file test.py che abbiamo creato come esempio di programma che utilizzi i metodi le classi e la funzione creati, per testarne il funzionamento. Sono presenti anche un file copy.py perchè è un modulo, che abbiamo importato ed un file __init__.py per ovvi motivi

File "`graph.py`"

Classe "`DirGraphNode`"

La classe ha i seguenti attributi eventualmente inizializzati di default:

```
self.id                 //numero intero identificativo del  nodo.
self.labels             //varie ed eventuali informazioni aggiuntive
self.neighbours_out     //lista contenente tuple aventi per primi elementi i possibili nodi
                        //raggiungibili e per secondi elementi i labels degli archi tramite
                        //i quali lo fanno
self.neighbours_in      //lista contenente i nodi da cui da cui arrivano gli archi che lo
                        //raggiungono
```

La classe contiene i seguenti metodi:

```
__init__(self, id=None, **labels)      //costruttore

add_neighbours_in(self, *new_neighbours_in)
                                //aggiunta di un nodo a neighbours_in

 add_neighbours_out(self, *new_neighbours_out, **edge_labels)
                                 //aggiunta di una tupla (nodo, labels) a neghbours_out
degrees(self)
                                //restituisce una tupla contenente il grado del nodo
                                //rispetto agli archi in uscita e il grado del nodo rispetto
                                //agli archi in entrata

get_edge_labels(self, elenco)   //data una lista di nodi, se questi sono
                                //neighbours_out
                                //del nodo, restituisce una lista di edge labels relativa
                                //agli archi che collegano gli elementi della lista con il
                                //nodo

get_neighbours(self)            //restituisce una tupla (lista nodi di neighbours_out,lista
                                //nodi di neighbours_in)
```

`rmv_neighbours_in(self, elenco)` //rimuove uno o più neighbours_in

`rmv_neighbours_out(self, elenco)` //rimuove uno o più neighbours_out

Classe "`DirectedGraph:`"

La classe ha i seguenti attributi eventualmente inizializzati di default:

```
self.name             //eventuale nodo del grafo
self.default_weight   //il peso dato agli archi se non diversamente specificato
self.nodes            //un dizionario avente per valori degli oggetti DirGraphNode e
                      //per chiavi gli id degli stessi
```

La classe contiene i seguenti metodi:

```
def __init__(self, name='noname_graph',
                default_weight=1.0,
                nodes=(),
                edges=(),
                node_labels={},
                edge_labels={}):      //costruttore
```

`add_edges(self, edge_list, **edge_labels)`
//aggiunge una lista di archi con le stesse etichette

`add_from_adjacency(self, matrice)` //aggiunge dei nodi e gli archi che li collegano partendo
//da una matrice di adiacenza

`add_from_files(self, percorso)` //aggiunge tutti gli elementi di un altro grafo salvato su
//file

`add_graph(self, grafo)` //aggiunge tutti gli elementi di un altro grafo

`add_nodes(self, id_list, **node_labels)`
//aggiunge una lista di nodi con le stesse etichette
//assegnando ad ogni nodo uno degli id specificati in
//lista

`auto_add_nodes(self, num, **node_labels)`
//aggiunge num nodi con le stesse etichette

`compute_adjacency(self, tipo='D')` //restituisce la matrice di adiacenza del grafo, in forma
//densa o sparsa a discrezione dell'utente

`copy(self)` //restituisce una copia del grafo

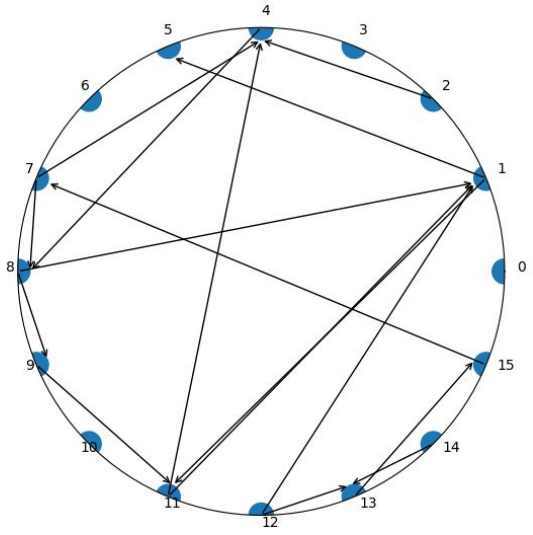
`get_edges(self)` //restituisce la lista di tutti gli archi

`get_edges_labels(self, edge_list)` //data una lista di archi ne restituisce le etichette

`minpath_dijkstra(self, id_start, id_end)`
//dati gli id di due nodi, calcola(se esiste) il cammino
//minimo, calcolato con l'algoritmo di Dijkstra.
//restituisce quindi una tupla con l'elenco dei nodi per
//cui si passa e il costo di ogni passaggio

`plot(self, etichette_nodi=False, etichette_archi=False)`
//genera un grafico del grafo. A discrezione dell'utente si
//può scegliere di visualizzare anche le etichette degli
//archi e/o dei nodi

esempio di esecuzione comando `grafo.plot(False, True)`




```
{(1, 5): {'weight': 4.0}}
{(1, 11): {'weight': 0.5}}
{(2, 4): {'weight': 1.0}}
{(4, 8): {'weight': 1.0}}
{(7, 4): {'weight': 1.0}}
{(7, 8): {'weight': 4.0}}
{(8, 9): {'weight': 4.0}}
{(8, 1): {'weight': 0.5}}
{(9, 11): {'weight': 1.0}}
{(11, 4): {'weight': 1.0}}
{(11, 1): {'lunghezza': '20', 'colore': 'arancione', 'weight': 1.0}}
{(12, 13): {'lunghezza': '10', 'colore': 'blu', 'weight': 2}}
{(12, 1): {'weight': 1.5}}
{(13, 15): {'weight': 2}}
{(14, 13): {'weight': 2}}
{(15, 7): {'weight': 1.5}}
```

rmv_edges(self, edge_list) //rimuove uno o più archi

rmv_nodes(self, lista_id) //rimuove uno o più nodi

save(self, **inputo)	//genera una cartella in cui salvare alcuni file contenenti
	//i dati relativi al grafo. A discrezione dell'utente si può
	//scegliere il nome della cartella e il path in cui crearla

size(self)	//ritorna una tupla con due interi rappresentanti il //numero di nodi ed il numero degli archi

File "functions.py"

Funzione "load_graph"

load_graph(percorso)	//crea un nuovo DirectedGraph utilizzando dati salvati //su file