UNIVERSITÀ DEGLI STUDI DI VERONA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in Informatica

Tesi di Laurea

Un approccio temporale per determinare la proprietà di notizie diffuse su Twitter

Relatore
prof. ssa Elisa Quintarelli

Laureando
Nicola Agresti
VR407685

Anno accademico 2019 - 2020

Indice

Dataset

Descrizione

Il dataset di partenza scelto (si veda [1]) contiene i Tweet degli utenti che hanno usato i seguenti hashtags: #coronavirus, #coronavirusoutbreak, #coronavirusPandemic, #covid19, #covid_19, #epitwitter, #ihavecorona, #StayHomeStaySafe, #TestTraceIsolate, a partire dal giorno 30/04/2020 alle ore 00:00:00 fino al giorno 01/05/2020 ore 01:59:00. Esso è composto da un unico file in formato CSV che contiene 22 colonne e 355387 righe. In dettaglio le 22 colonne sono:

1. status_id: un id univoco che identifica un tweet
2. user_id: un id univoco che identifica un utente
3. created_at: un timestamp di quando è stato creato un tweet
4. screen_name: il nome utente collegato allo user_id
5. text: il testo del tweet
6. source: la piattaforma usata per creare il tweet
7. reply_to_status_id: l'id del tweet a cui il tweet è in risposta
8. reply_to_user_id: l'id dell'utente a cui il tweet è in risposta
9. reply_to_screen_name: il nome utente a cui il tweet è in risposta
10. is_quote: un booleano per descrivere se il tweet è di tipo quotes
11. is_retweet: un booleano per descrivere se il tweet è di tipo retweet
12. favorites_count: il numero di mi piace ricevuti
13. retweet_count: il numero di retweet ricevuti
14. country_code: il codice dello stato da dove è stato creato il tweet
15. place_full_name: il nome della città da dove è stato creato il tweet
16. place_type: una descrizione del tipo di città da dove è stato creato il tweet

17. followers_count: il numero dei followers dell'account che ha creato il tweet
18. friends_count: il numero di amici dell'account che ha creato il tweet
19. account_lang: la lingua usata dall'account che ha creato il tweet
20. verified: un booleano per descrivere se è un account verificato
21. lang: la lingua usata nel testo del tweet

Alcune di queste informazioni non sono utili per quanto riguarda l'obiettivo di questa tesi, di conseguenza sono state eseguite delle scelte progettuali per capire quali attributi sono da mantenere nel dataset.

Progettazione Logica

Prima di andare a personalizzare il dataset è stato ideato un opportuno schema ER che potesse descrivere al meglio il problema, lo schema ER risultante è: Mentre lo schema relazionale risulta:

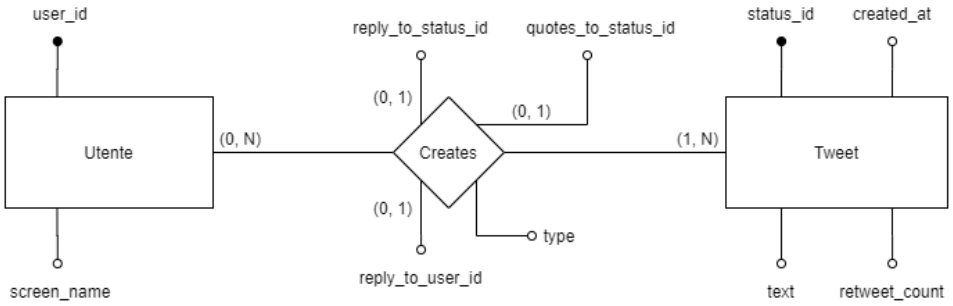

Figura 1. Schema ER

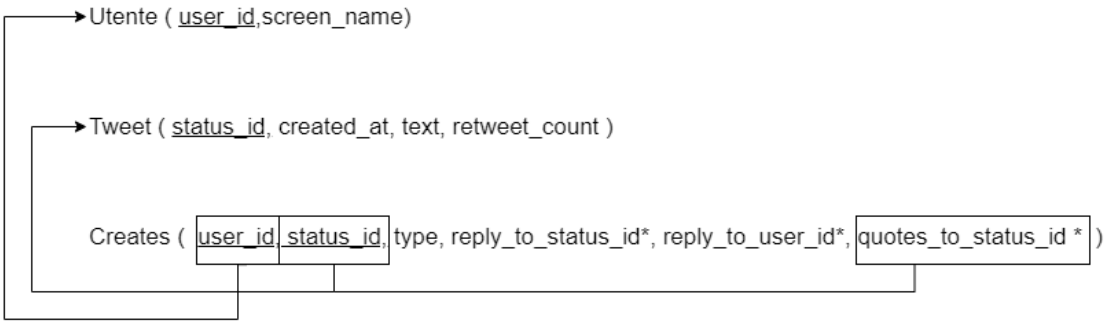

Figura 2. Schema Relazionale

Entrambe le figure sono state realizzate con ER.drawio [3].

Personalizzazione

In base alla progettazione logica ho creato 3 file csv, per prima cosa ho aperto con Excel il dataset originale e quindi ho copiato l'intera colonna user_id e l'intera colonna screen_name in un nuovo file denominato "Utente.csv", poi ho controllato tramite un' apposita funzione di Excel che non ci fossero duplicati nella chiave primaria "user_id".

	A	B
1	user_id	screen_name
2	3171712086	RahulGandhi
3	93794912	seoulmania
4	705694814612938752	NCDCgov
5	102071743	GovWhitmer
6	16989178	JamesOKeefeIII
7	531041640	KKMPutrajaya
8	1030099481533014017	MazzoleniJulio
9	147994804	rishibagree
10	21059255	tedlieu

Figura 3. Esempio preso dal file utente.csv

Ho poi proseguito copiando le colonne status_id, created_at, text e retweet_count in un altro file di nome "Tweet.csv", ho controllato nuovamente la presenza di eventuali duplicati nella chiave primaria status_id.

	A	B	C	D
1	status_id	created_at	text	retweet_count
2	1255701270901387264	30/04/2020 05:31	A conversation with Dr Raghuram Rajan, former RBI Governor, on dealing with the #Co	10146
3	1255852107673907202	30/04/2020 15:30	ì,¬íšŒì ê±°ë¦¬ëŠ" ë'ì§€ë§Œâ†"ëŠ^ìŒì€ ê°€êª'Œì› ìŠµë‹ˆ ë«¤ðŸ'—#ìš°ë¦¬_ëª¨ë'ê°€_ì˜ì›…ìž	9521
4	1255993387603329026	01/05/2020 00:52	204 new cases of #COVID19 reported;80-Kano45-Lagos12-Gombe9-Bauchi9-Sokoto7-Bc	8852
5	1255689432738349059	30/04/2020 04:44	At 9 AM today, you can watch my conversation with Dr Raghuram Rajan, former RBI Go	8507
6	1255978307641905154	30/04/2020 23:52	Iâ€™ve said it before, and Iâ€™ll say it again â€" Michigan is an extraordinary place to l	6961
7	1255976688594419718	30/04/2020 23:45	UNBELIEVABLE! Our team just caught @Twitter red handed stealing retweets off our ne	5504
8	1255778934668398593	30/04/2020 10:40	Terkini 30 April #COVID19 ðŸ‡²ðŸ‡¾Kes baharu 5âƒ£7âƒ£ (25 kes import)Kematian	4629
9	1255682896020541440	30/04/2020 04:18	Informe #COVID19: hoy procesamos 563 muestras, 10 dieron positivo: 5 del exterior, 1	4400
10	1255857575783919617	30/04/2020 15:52	Didi Stays in a Gated Community at 41st floor.Now tell me which beggar/ Unknown per	4239

Figura 4. Esempio preso dal file tweet.csv

Per quanto riguarda il file creates.csv ho copiato le colonne user_id, status_id, reply_to_status_id, reply_to_user_id e successivamente ho aggiunto una colonna nuova di nome "type" che conterrà una delle seguenti tipologie di tweet:

- TW: un tweet originale creato da un utente

- RE: il tweet creato è in risposta ad un altro tweet
- QT: il tweet creato è un retweet con un commento
- REQT: il tweet creato è un retweet con un commento in risposta ad un altro tweet
- RT: un retweet di un tweet di uno dei 4 tipi precedenti

	user_id	status_id	type	reply_to_status_id	reply_to_user_id
28	921525775169458176	1255968042787524609	TW		
29	22703339	1255922986726707200	TW		
30	109263459	1255909313316601858	RE	1255905597976969216	109263459
31	16910746	1255860128231895044	TW		
32	11347122	1255949158751571968	TW		
33	705694814612938752	1255997250544812035	RE	1255993387603329026	705694814612938752
34	77372998	1255852155967332352	TW		
35	12272322	1255930519298224134	TW		
36	42407972	1255981948872413184	TW		
37	1154180084078501889	1255947666866843648	QT		
38	928677206	1255894000827011072	TW		

Figura 5. Esempio preso dal file creates.csv

La colonna type nel file creates l'ho popolata nel seguente modo:

- Assegno RE ad ogni riga in cui è presente un valore in reply_to_status_id o reply_to_user_id
- Assegno QT ad ogni riga in cui is_quote è true
- Assegno REQT ad ogni riga in cui è presente un valore in reply_to_status_id o reply_to_user_id e is_quote è true
- Assegno TW ai restanti tweet

Come si può notare il tipo RT non è presente, questo perché nel dataset è salvato solamente il numero dei retweet e non chi e quando li ha fatti. Per ovviare a questa mancanza ho provveduto a generare N tuple da inserire in creates tante quante la somma di tutti i retweet_count della tabella tweet. Per fare ciò ho creato un file creates.csv dove per convenienza ho aggiunto la colonna dei retweet_count. Poi grazie a un convertitore online, si veda [2], ho trasformato il file creates_rtcount.csv in creates_rtcount.json, quindi

	user_id	status_id	type	reply_to_status_id	reply_to_user_id	retweet_count
28	921525775169458176	1255968042787524609	TW			1947
29	22703339	1255922986726707200	TW			1691
30	109263459	1255909313316601858	RE	1255905597976969216	109263459	1617
31	16910746	1255860128231895044	TW			1616
32	11347122	1255949158751571968	TW			1598
33	705694814612938752	1255997250544812035	RE	1255993387603329026	705694814612938752	1591
34	77372998	1255852155967332352	TW			1563
35	12272322	1255930519298224134	TW			1443
36	42407972	1255981948872413184	TW			1429
37	1154180084078501889	1255947666866843648	QT			1428
38	928677206	1255894000827011072	TW			1428

Figura 6. Esempio preso dal file creates_rtcount.csv

ho proceduto a programmare in python questo "generatore" di retweet, la porzione di codice più importante è:

```
for row in reader:
        if row['retweet_count'] > 0:
            for tupla in range(row['retweet_count']):
                stringa = {"user_id": user[posUtente],
                           "status_id": row['status_id'],
                           "type": "RT",
                           "reply_to_status_id": None,
                           "reply_to_user_id": None}
                retweet.append(stringa)
                if posUtente != len(user) - 1:
                    posUtente = posUtente + 1
                else:
                    posUtente = 0
```

Ogni tupla che rappresenta un retweet di un tweet avrà un user_id casuale e lo status_id uguale a quello del tweet originale, le ho inserite in una lista che poi ho salvato in un file JSON. Sempre grazie a [2] ho proseguito a convertire gli altri file CSV in modo da prepararmi a popolare un eventuale database.

Database

Creazione

Grazie al lavoro fatto in precedenza, vedi Progettazione Logica, ho creato un Database locale sul mio PC tramite PostgreSQL 12 [4], successivamente ho creato le tre tabelle previste dalla Progettazione Logica con una semplice query:

```
CREATE TABLE utente(
    user_id VARCHAR PRIMARY KEY CHECK (user_id <> ''),
    screen_name VARCHAR NOT NULL CHECK (screen_name <> '')
);
CREATE TABLE tweet(
    status_id VARCHAR PRIMARY KEY CHECK (status_id <> ''),
    text VARCHAR NOT NULL CHECK (text <> ''),
    created_at TIMESTAMP NOT NULL CHECK (created_at <=
       CURRENT_TIMESTAMP),
    retweet_count INTEGER CHECK (retweet_count >= 0)
);
CREATE DOMAIN TYPETWEET VARCHAR(4) CHECK(
    VALUE IN ('TW', 'RE', 'RT', 'QT', 'REQT')
);
CREATE TABLE creates(
    user_id VARCHAR REFERENCES utente,
    status_id VARCHAR REFERENCES tweet,
    type TYPETWEET NOT NULL,
    reply_to_status_id VARCHAR DEFAULT NULL,
    reply_to_user_id VARCHAR DEFAULT NULL,
    quotes_to_status_id VARCHAR DEFAULT NULL REFERENCES
       tweet,
    PRIMARY KEY (user_id, status_id)
);
```

Upload Dati

Durante alcune prove per popolare le tabelle del database ho riscontrato problemi con il testo dei tweet, questo perché erano presenti degli apici e delle emoticon non codificate. Per quanto riguarda gli apici, SQL li interpreta solo se sono doppi, mentre le emoticon dovevano essere tradotte nella loro codifica esadecimale. Di conseguenza ho creato in python (si veda [5] e [6]) un programma che va a sostituire un apice con un doppio apice e le emoticon con la loro codifica, in generale va a correggere un intero file JSON dato in input. Il codice è:

```
import json

parser = json.JSONDecoder()
parsed = []
with open("tweet.json", encoding='utf8') as f:
    data = f.read()
    data = data.replace("'", "'"+"'")

head = 0
for row in data:
    head = (data.find('{', head) + 1 or data.find('[', head)
       + 1) - 1
    try:
        struct, head = parser.raw_decode(data, head)
        parsed.append(struct)
    except (ValueError, json.JSONDecodeError):
        break

with open("tweet_output.json", "w") as outfile:
    json.dump(parsed, outfile, separators = (',', ': '),
       indent = 1)

print("\nFile JSON formattato correttamente!")
```

Fatto ciò ho tutti gli ingredienti per andare a popolare il database, ho creato (si veda [7]) un uploader in python che va a creare una connessione e fare un INSERT per ogni oggetto del file JSON dato in input. Il codice è:

```
import json
import psycopg2
from myAppConfig import myHost, myDatabase, myUser, myPsw

connessione = psycopg2.connect (host = myHost, database =
   myDatabase, user = myUser, password = myPsw)

with connessione:
    with connessione.cursor() as cursore:

        with open("createsRT.json", encoding='utf8') as f:
            data = json.load(f)

        keys = []
        for row in data:
            for key in row.keys():
                if key not in keys:
                    keys.append(key)

        for row in data:
            query = "INSERT INTO creates (user_id,
               status_id, type, reply_to_status_id,
               reply_to_user_id)
               VALUES({0});".format(",".join(map(lambda key:
               "'{0}'".format(row[key]) if key in row else
               "NULL", keys)))
            cursore.execute(query)

        print("Caricamento completato, tuple inserite!")

connessione.close()
print("Connessione chiusa con successo!")
```

Modificando il file in input con quello desiderato e la stringa query ho eseguito in modo corretto l'upload di tutti i file JSON che ho creato in precedenza.

Grafo

Struttura

Prima di poter sviluppare un programma che generi un determinato grafo c'è la necessità di capire come andarlo a strutturare. Dopo alcuni colloqui abbiamo scelto di implementare un grafo che avesse i seguenti nodi

- DB, il nodo radice del grafo
- Tweet, un nodo che rappresenta un'istanza di un tweet
- Retweet, un nodo che rappresenta un'istanza di un retweet

Ogni nodo può essere connesso ad un altro tramite un arco, in particolare nel grafo troveremo questi archi:

- DB → Tweet
- DB → Retweet
- Tweet → Retweet
- Tweet → Tweet successivo (stesso creatore)
- Retweet → Tweet successivo (se presente)
- Tweet precedente → Retweet (se presente)

Ogni arco è descritto da un'etichetta del tipo <Start, End, Author, Weight> dove Weight rappresenta un valore che descrive quanto un autore ha collaborato nella creazione del tweet o retweet, in particolare può assumere questi valori:

- 1 se l'arco è entrante in un Tweet
- 0.1 se l'arco è entrante in un Retweet
- 0.5 se l'arco è entrante in un Tweet di tipo quotes

Start ed End rappresentano un intervallo temporale in minuti per descrivere la creazione del nodo a cui punta l'arco.
Author rappresenta il proprietario del Tweet o Retweet.

I nodi con degli intervalli di tempo indeterminati vengono descritti con un'etichetta qualitativa con il valore 0 in Start e il valore ∞ in End. Per distiguere meglio i nodi e archi ho scelto di assegnare alcuni colori:

- Per il nodo radice DB
- Per il nodo che rappresenta un tweet di tipo TW, il suo arco collegato a DB e l'arco collegato al tweet successivo
- Per il nodo che rappresenta un tweet di tipo RE, il suo arco collegato a DB e l'arco collegato al tweet successivo
- Per il nodo che rappresenta un tweet di tipo QT, il suo arco collegato a DB e l'arco collegato al tweet successivo
- Per il nodo che rappresenta un tweet di tipo REQT, il suo arco collegato a DB e l'arco collegato al tweet successivo
- Per il nodo che rappresenta un retweet e il suo arco collegato a un nodo tweet

Creazione

Per la creazione del grafo ho scelto di usare nuovamente Python, in particolare la libreria NetworkX [8] mi è venuta in aiuto grazie alle possibilità di creazione e personalizzazione di nodi, archi ed etichette. Ho scelto di inizializzare un grafo di tipo DiGraph, questo perché dà la possibilità di dare una direzione agli archi e fornisce alcuni metodi aggiuntivi che ho usato nel mio programma. Ad ogni avvio la prima cosa che fa il programma è il creare la connessione con il database e successivamente inizializzare il cursore che mi permetterà di eseguire le query necessarie a prelevare determinate informazioni. La prima query che andrà a fare e la seguente:

```
DROP VIEW IF EXISTS user_id_quotes;
    CREATE VIEW user_id_quotes AS (
        SELECT U.screen_name, U.user_id, T.status_id
        FROM utente U
             JOIN creates C ON U.user_id = C.user_id
             JOIN tweet T ON C.status_id = T.status_id
        WHERE C.type = 'TW' AND
              T.status_id IN (SELECT Ci.quotes_to_status_id
                              FROM creates Ci
                              WHERE Ci.quotes_to_status_id
                                 IS NOT NULL)
);
SELECT T.status_id, T.created_at, U.screen_name, U.user_id,
   C.type,
       C.quotes_to_status_id, V.screen_name
FROM tweet T
     JOIN creates C ON T.status_id = C.status_id
     JOIN utente U ON C.user_id = U.user_id
     LEFT JOIN user_id_quotes V ON C.quotes_to_status_id =
        V.status_id
WHERE C.type <> 'RT' AND
      T.status_id IN ('1255877065758388224',
                      '1255779231776206849',
                      '1255751113053265920',
                      '1255930357691453442',
                      '1255744610523021312')
ORDER BY U.user_id, T.created_at;
```

Successivamente avrò come risultato una tabella contenente lo status_id, created_at, screen_name, user_id e type di ogni tweet specificato nella clausola WHERE, inoltre se il tweet è di tipo QT avrò il quotes_to_status_id e lo screen_name di chi ha creato il tweet a cui fa riferimento il quotes, negli altri tipi di tweet queste due colonne sono popolate da NULL. La fase successiva è la creazione della prima parte di grafo, il codice è il seguente:

```
tweetIdOriginale = cursore.fetchall()
status_id_tweet = [tupla[0] for tupla in tweetIdOriginale]
created_at_tweet = [tupla[1] for tupla in tweetIdOriginale]
screen_name_tweet = [tupla[2] for tupla in tweetIdOriginale]
user_id_tweet = [tupla[3] for tupla in tweetIdOriginale]
type_tweet = [tupla[4] for tupla in tweetIdOriginale]
original_tw_quoted = [tupla[5] for tupla in tweetIdOriginale]
author_tw_quoted = [tupla[6] for tupla in tweetIdOriginale]

startDB = datetime.strptime('2020-04-30 00:00:00', '%Y-%m-%d
   %H:%M:%S')

    for x in range(len(tweetIdOriginale)):
        node_created_at =
           datetime.strptime(str(created_at_tweet[x]),
           '%Y-%m-%d %H:%M:%S')
        interval = node_created_at - startDB
        minuteInterval = int(interval.total_seconds()/60)
        attr_dict = {}
        time_list = []
        if type_tweet[x] != 'QT':
            attr_dict["Start"] = minuteInterval
            attr_dict["End"] = minuteInterval
            attr_dict["Authors"] = [(screen_name_tweet[x],
               1)]
            stringa = tuple(x for x in attr_dict.values())
            time_list.append(stringa)
            graph.add_edge('DB', status_id_tweet[x],
               attr_dict = attr_dict, time = stringa,
               time_list = time_list)
        elif type_tweet[x] == 'QT':
            attr_dict["Start"] = minuteInterval
            attr_dict["End"] = minuteInterval
            attr_dict["Authors"] = [(screen_name_tweet[x],
               0.5)]
            stringa = tuple(x for x in attr_dict.values())
            time_list.append(stringa)
```

```
31              graph.add_edge('DB', status_id_tweet[x],
                    attr_dict = attr_dict, time = stringa,
                    time_list = time_list)
32              attr_dict["Start"] = 0
33              attr_dict["End"] = math.inf
34              attr_dict["Authors"] = [(screen_name_tweet[x],
                    0.5)]
35              stringa = tuple(x for x in attr_dict.values())
36              time_list = []
37              time_list.append(stringa)
38              graph.add_edge(original_tw_quoted[x],
                    status_id_tweet[x], attr_dict = attr_dict,
                    time = stringa, time_list = time_list)
39          if x < len(tweetIdOriginale) - 1:
40              if user_id_tweet[x] == user_id_tweet[x+1]:
41                  attr_dict = {}
42                  attr_dict["Start"] = 0
43                  attr_dict["End"] = math.inf
44                  attr_dict["Authors"] =
                        [(screen_name_tweet[x], 1)]
45                  stringa = tuple(x for x in
                        attr_dict.values())
46                  time_list = []
47                  time_list.append(stringa)
48                  graph.add_edge(status_id_tweet[x],
                        status_id_tweet[x+1], attr_dict =
                        attr_dict, time = stringa, time_list =
                        time_list)
```

Questa porzione di codice salva ogni colonna della tabella ottenuta tramite la query in una lista, successivamente procede a scorrere ogni elemento, controlla se è un quotes o un altro tipo di tweet e di conseguenza crea l'etichetta adatta in base alle scelte fatte in precedenza, vedi Struttura, lo collega al nodo radice DB e se sono creati dallo stesso utente connette i tweet tra di loro.

Dopo questa fase il programma procede con la seconda query:

```
DROP VIEW IF EXISTS User_Id_Originale;
CREATE VIEW User_Id_Originale AS (
    SELECT U.user_id, T.status_id, U.screen_name
    FROM tweet T
         JOIN creates C ON T.status_id = C.status_id
         JOIN utente U ON C.user_id = U.user_id
    WHERE C.type <> 'RT' AND
          T.status_id IN ('1255877065758388224',
                          '1255779231776206849',
                          '1255751113053265920',
                          '1255930357691453442',
                          '1255744610523021312')
    ORDER BY U.user_id
);
SELECT U.user_id, T.status_id, T.created_at, U.screen_name,
       V.user_id, V.screen_name
FROM utente U
     JOIN creates C ON U.user_id = C.user_id
     JOIN tweet T ON C.status_id = T.status_id
     JOIN User_Id_Originale V ON T.status_id = V.status_id
WHERE C.type = 'RT' AND
      T.status_id IN ('1255877065758388224',
                      '1255779231776206849',
                      '1255751113053265920',
                      '1255930357691453442',
                      '1255744610523021312')
ORDER BY V.user_id, T.created_at;
```

La quale restituirà una tabella contenente user_id, status_id, created_at, screen_name di chi ha creato il retweet e user_id, screen_name del creatore del tweet che è stato retweetato, il tutto in base agli status_id inseriti nella clausola WHERE.

Per creare la seconda parte del grafo ho scritto il seguente codice:

```
utenti_tweet_RT = cursore.fetchall()
user_id = [tupla[0] for tupla in utenti_tweet_RT]
status_id = [tupla[1] for tupla in utenti_tweet_RT]
status_id_no_duplicates =
   list(OrderedDict.fromkeys(status_id))
created_at = [tupla[2] for tupla in utenti_tweet_RT]
screen_name = [tupla[3] for tupla in utenti_tweet_RT]
creator_user_id = [tupla[4] for tupla in utenti_tweet_RT]
creator_screen_name = [tupla[5] for tupla in utenti_tweet_RT]

user_tweet_duplicates = defaultdict(list)
    for i, j in zip(creator_screen_name, status_id):
        user_tweet_duplicates[i].append(j)
    user_tweet = {a:list(OrderedSet(b)) for a, b in
       user_tweet_duplicates.items()}

    for i in range(len(utenti_tweet_RT)):
        rt_id = status_id[i]+'-'+str(i)
        retweet_id.append(rt_id)
        attr_dict = {}
        attr_dict["Start"] = 0
        attr_dict["End"] = math.inf
        attr_dict["Authors"] = [(screen_name[i], 0.1)]
        stringa = tuple(x for x in attr_dict.values())
        time_list = []
        time_list.append(stringa)
        graph.add_edge(status_id[i], retweet_id[i],
           attr_dict = attr_dict, time = stringa, time_list
           = time_list)
        graph.add_edge('DB', retweet_id[i], attr_dict =
           attr_dict, time = stringa, time_list = time_list)

    posRT = 0
    rt_id_precedenti = []
    for user, tweet in user_tweet.items():
        posTW = 0
        for tw in tweet:
            nodes = []
            nodes = nx.dfs_preorder_nodes(graph, tw,
               depth_limit = 1)
            for n in nodes:
                if n == 'DB':
                    continue
                elif n in status_id_no_duplicates:
                    continue
```

```
40                  elif n in rt_id_precedenti:
41                      continue
42                  else:
43                      for x in range(0, posRT):
44                          if retweet_id[x] not in
                               rt_id_precedenti:
45                              rt_id_precedenti.append(retweet_id[x])
46                      posRT = posRT + 1
47                      if len(tweet) > 1 and posTW < len(tweet)
                           - 1:
48                          attr_dict = {}
49                          attr_dict["Start"] = 0
50                          attr_dict["End"] = math.inf
51                          attr_dict["Authors"] =
                               [(creator_screen_name[posTW+1],
                               1)]
52                          stringa = tuple(x for x in
                               attr_dict.values())
53                          time_list = []
54                          time_list.append(stringa)
55                          graph.add_edge(n, tweet[posTW+1],
                               attr_dict = attr_dict, time =
                               stringa, time_list = time_list)
56                      if len(tweet) > 1 and posTW > 0:
57                          attr_dict = {}
58                          attr_dict["Start"] = 0
59                          attr_dict["End"] = math.inf
60                          attr_dict["Authors"] =
                               [(screen_name[posTW], 0.1)]
61                          stringa = tuple(x for x in
                               attr_dict.values())
62                          time_list = []
63                          time_list.append(stringa)
64                          graph.add_edge(tweet[posTW-1], n,
                               attr_dict = attr_dict, time =
                               stringa, time_list = time_list)
65              posTW = posTW + 1
```

Come in precedenza salvo ogni colonna della tabella ottenuta in liste, in aggiunta ho creato la lista status_id_no_duplicates che corrisponde alla lista status_id ma senza duplicati, in pratica conterrà tutti gli id univoci dei tweet originali che sono stati retweetati. Il programma procede con la creazione di un dizionario che avrà come chiavi tutti gli user_id dei creatori dei tweet originali e come valori ogni status_id, senza duplicati (si veda [9], [10], [11]), che corrisponde ad un tweet creato dall'utente. Vengono poi

creati gli archi che dal tweet originale si collegano al nodo di ogni retweet e poi da ogni retweet un arco verso DB, il tutto con le etichette decise nella Struttura. Poi il programma inizia a scorrere ogni utente ed analizzare ogni tweet creato dallo stesso, in particolare va ad estrarre i nodi successivi al tweet fino alla profondità massima di 1 arco (vedi [12]), salta i nodi che non servono per creare nuovi archi ed infine se l'utente ha creato più di 1 tweet verranno connessi i retweet precedenti/successivi secondo quanto stabilito nella Struttura.
Dopo queste fasi la struttura del grafo è completa, la fase seguente è la colorazione dei nodi e archi in base alla loro tipologia e la loro stampa sotto forma di immagine, per fare ciò ho creato delle liste per poter capire di quale tipologia fosse il nodo e arco, ho usato queste query:

```
SELECT DISTINCT C.user_id
FROM creates C
WHERE C.type = 'RT';

SELECT C.status_id
FROM creates C
WHERE C.type = 'TW';

SELECT C.status_id
FROM creates C
WHERE C.type = 'QT';

SELECT C.status_id
FROM creates C
WHERE C.type = 'RE';

SELECT C.status_id
FROM creates C
WHERE C.type = 'REQT';
```

Le loro liste verranno usate in questa funzione creata appositamente:

```
def mapping_colors_nodes_edges(G):
    edge_colors.clear()
    node_colors.clear()
    for node in G:
        if node in check_tweetTW:
            node_colors.append('#269AED')
        elif node in check_tweetQT:
            node_colors.append('#EB8D00')
        elif node in check_tweetRE:
            node_colors.append('#FFE100')
        elif node in check_tweetREQT:
            node_colors.append('#AA00FF')
        elif node == 'DB':
            node_colors.append('red')
        else:
            node_colors.append('#0DA342')

    for edge in G.edges:
        if any(ele in edge for ele in retweet_id) == True:
            edge_colors.append('#0DA342')
        elif any(ele in edge for ele in check_tweetQT) ==
           True:
            edge_colors.append('#EB8D00')
        elif any(ele in edge for ele in check_tweetRE) ==
           True:
            edge_colors.append('#FFE100')
        elif any(ele in edge for ele in check_tweetREQT) ==
           True:
            edge_colors.append('#AA00FF')
        elif any(ele in edge for ele in check_tweetTW) ==
           True:
            edge_colors.append('#269AED')
```

In dettaglio questa funzione va a salvare in due liste, una per i nodi e una per gli archi, il codice del colore corrispondente alla posizione del nodo/arco.

Questa verrà usata in un'altra funzione per la stampa del grafo:

```
def print_graph (sub_G):
    posSub_G = nx.shell_layout(sub_G)
    mapping_colors_nodes_edges(sub_G)
    nx.draw_networkx(sub_G, pos = posSub_G, with_labels =
       True, node_color = node_colors, edge_color =
       edge_colors, font_size = 9)
    nx.draw_networkx_edge_labels(sub_G, pos = posSub_G,
       edge_labels = nx.get_edge_attributes(sub_G, 'time'),
       font_color = 'red', font_size = 7)
    plt.plot(1)
    plt.show(block = False)
```

La funzione di libreria nx.shell_layout(sub_G), si veda [13], mappa la posizione dei nodi in cerchi concentrici. In generale verrà creata l'immagine completa del grafo e la si potrà visionare grazie alla libreria matplotlib.pyplot (si veda [14]).

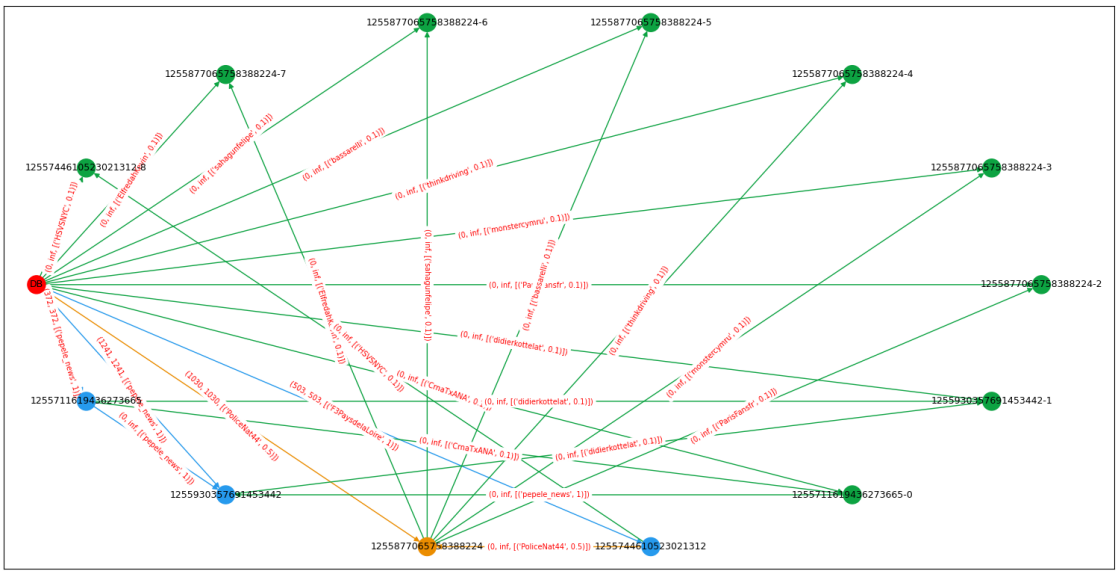

Figura 7. Grafo di esempio

Salvataggio

Il salvataggio del grafo può avvenire in due modi, tramite il plot del grafo avremo la possibilità di salvare un file PNG del grafo, oppure tramite una apposita funzione che ho creato si potrà salvare un file che conterrà la lista completa dei nodi e archi del grafo, la funzione è:

```
def save_file_edgelist(G):
    if not nx.is_empty(G):
        nome = input("Inserire un nome per il salvataggio
           del grafo: ")
        nome = nome + '.edgelist'
        f = open(nome, 'wb')
        nx.write_edgelist(G, f)
        print("Sotto-grafo salvato correttamente.")
        f.close()
    else:
        print("Non c'e' nessun sotto-grafo da salvare.")
```

La funzione is_empty(G) (si veda [16]) controlla che il grafo non sia vuoto, quindi si procede con la creazione del file tramite write_edgelist(G, f) che riceve in input il grafo da salvare e il file (si veda [15]).

Caricamento

Un grafo salvato può essere caricato dal programma tramite la seguente funzione:

```
def read_file_edgelist():
    nome = input("Inserire il nome del grafo da caricare: ")
    nome = nome + '.edgelist'
    try:
        f = open(nome, 'rb')
        G = nx.read_edgelist(f, create_using = nx.DiGraph())
        f.close()
        print("Sotto-grafo letto correttamente.")
        return G
    except:
        print("Il nome inserito non esiste!")
```

La funzione read_edgelist (si veda [17]) riceve in input il file.edgelist e l'indicazione della tipologia di grafo, creerà e copierà il grafo creato nella variabile G la quale verrà inserita nel return della funzione principale.

Algoritmo Path-Consistency

Prima di dare la definizione dell'algoritmo è necessario dare la definizione di altri tre concetti che ci serviranno per capirlo meglio.

Definizione 1 (Inversione) *Data una tupla* $T_k =< Start, End >$ *il suo inverso è* $T_k^{-1} =< -End, -Start >$

Definizione 2 (Composizione $\circ$**)** *Date due tuple* $T_{k1} =< Start_1, End_1 >$ *e* $T_{k2} =< Start_2, End_2 >$ *la composizione è*
$T_{k12} =< Start_1 + Start_2, End_1 + End_2 >$*, in pratica la somma di due tuple.*

Definizione 3 (Congiunzione $\otimes$**)** *Date due tuple* $T_{k1} =< Start_1, End_1 >$ *e* $T_{k2} =< Start_2, End_2 >$ *la congiunzione è*
$T_{k12} =< MAX(Start_1, Start_2), MIN(End_1, End_2) >$*, in pratica crea una tupla con il valore Start massimo e il valore End minimo tra le due tuple.*

Ora possiamo dare la definizione dell'algoritmo

Definizione 4 (Path-Consistency) *Date tre tuple* $R_{ij} =< Start, End >$, $R_{ik} =< Start, End >$ *e* $R_{kj} =< Start, End >$ *l'algoritmo esegue quanto segue* $R_{ij}^{'} = R_{ij} \otimes (R_{ik} \circ R_{kj})$

Può accadere che in alcuni casi di studio serva applicare l'inversione a una tupla per ottenere un risultato coerente con il grafo.

Definizione 5 (Intersezione Intervalli $\cap$**)** *Dati due intervalli* $T_1 =< Start_1, End_1 >$ *e* $T_2 =< Start_2, End_2 >$ *l'intersezione* $T_{12} = T_1 \cap T_2$ *è data dal valore minimo di Start* $min(Start_1, Start_2)$ *e quello massimo di End* $max(End_1, End_2)$*, nel caso End sia maggiore o uguale di Start il risultato è la sottrazione* $End - Start$ *altrimenti 0.*

Definizione 6 (Unione Intervalli $\cup$**)** *Dati due intervalli* $T_1 =< Start_1, End_1 >$ *e* $T_2 =< Start_2, End_2 >$ *l'unione* $T_{12} = T_1 \cup T_2$ *è data dal valore massimo di Start* $max(Start_1, Start_2)$ *e quello minimo di End* $min(End_1, End_2)$*, nel caso End sia maggiore o uguale di Start il risultato è la sottrazione* $End - Start$ *altrimenti 0.*

Definizione 7 (Similarità) *Dati due intervalli* $T_1 =< Start_1, End_1 >$ *e* $T_2 =< Start_2, End_2 >$ *la similarità è data da* $sim(T_1, T_2) = \frac{T_1 \cap T_2}{T_1 \cup T_2}$*, nel caso l'unione risulti infinita il risultato è 0.1 (poca similarità), nel caso sia diversa da 0 il risultato è dato dal rapporto tra l'intersezione e l'unione, altrimenti risulta 0.*

Al grafo viene applicato l'algoritmo con il seguente codice:

```
def path_consistency(G):
    changed = True
    count = 0
    while(changed == True):
        count = count + 1

        G_edgelist = open('G_file.txt', 'wb')
        nx.write_edgelist(G, G_edgelist)
        G_edgelist.close()

        G_new = copy.deepcopy(path_consistency_step(G))

        G_new_edgelist = open('G_new_file.txt', 'wb')
        nx.write_edgelist(G_new, G_new_edgelist)
        G_new_edgelist.close()

        G_edgelist = open('G_file.txt')
        G_new_edgelist = open('G_new_file.txt')

        if filecmp.cmp('G_file.txt', 'G_new_file.txt',
           shallow=False) == True:
            G_edgelist.close()
            G_new_edgelist.close()
            print("La "+str(count)+"   esecuzione non ha
               apportato modifiche.")
            changed = False
        else:
            G_edgelist.close()
            G_new_edgelist.close()
            G = copy.deepcopy(G_new)
            print(str(count) + "   esecuzione
               dell'algoritmo.")

    return G_new
```

Dove una porzione significativa della funzione path_consistency_step è:

```
def path_consistency_step(G):
    G_new = nx.DiGraph()
    G_new.add_nodes_from(copy.deepcopy(G.nodes(data =
       False)))

    for edge in G.edges():
        triangles = []
        triangles = get_triangles(G, edge[0], edge[1])
        for t in triangles:
            if (G.has_edge(t[0], t[1]) and G.has_edge(t[1],
               t[2])):

                if G_new.has_edge(t[0], t[2]) == True:
                    Rij_label = {}
                    Rij_label =
                       copy.deepcopy(G_new.get_edge_data(t[0],
                       t[2]))
                    Rij = (Rij_label['attr_dict']['Start'],
                       Rij_label['attr_dict']['End'])
                else:
                    Rij_label = {}
                    Rij_label =
                       copy.deepcopy(G.get_edge_data(t[0],
                       t[2]))
                    Rij = (Rij_label['attr_dict']['Start'],
                       Rij_label['attr_dict']['End'])

                Rik_label = {}
                Rik_label =
                   copy.deepcopy(G.get_edge_data(t[0], t[1]))
                Rik = (Rik_label['attr_dict']['Start'],
                   Rik_label['attr_dict']['End'])

                Rkj_label = {}
                Rkj_label =
                   copy.deepcopy(G.get_edge_data(t[1], t[2]))
                Rkj = (Rkj_label['attr_dict']['Start'],
                   Rkj_label['attr_dict']['End'])

                autori = []
                for autore1 in
                   Rik_label['attr_dict']['Authors']:
                    for autore2 in
                       Rkj_label['attr_dict']['Authors']:
                        if autore1[0] == autore2[0]:
```

```
32                          autori.append((autore1[0],
                               max(autore1[1], autore2[1])))
33                      else:
34                          autori.append(autore1)
35                          autori.append(autore2)
36
37                  Rikj = (Rik[0]+Rkj[0], Rik[1]+Rkj[1])
38
39                  if Rij[0] >= Rikj[0]:
40                      if Rij[1] >= Rikj[1]:
41                          Rij_new = [Rij[0], Rikj[1],
                               Rij_label['attr_dict']['Authors']]
42                      else:
43                          Rij_new = [Rij[0], Rij[1],
                               Rij_label['attr_dict']['Authors']]
44                  elif Rij[0] < Rikj[0]:
45                      if Rij[1] >= Rikj[1]:
46                          Rij_new = [Rikj[0], Rikj[1],
                               Rij_label['attr_dict']['Authors']]
47                      else:
48                          Rij_new = [Rikj[0], Rij[1],
                               Rij_label['attr_dict']['Authors']]
49
50                  for index, autore in enumerate(autori):
51                      y = list(autore)
52                      y[1] = similarity(Rikj[0], Rikj[1],
                           Rij_new[0], Rij_new[1])
53                      autori[index] = y
54
55                  for index, autore in enumerate(Rij_new[2]):
56                      y = list(autore)
57                      y[1] =
                           similarity(Rij_label['attr_dict']['Start'],
                           Rij_label['attr_dict']['End'],
                           Rij_new[0], Rij_new[1])
58                      Rij_new[2][index] = y
59
60                  Rij_new[2] = Rij_new[2]+autori
61
62                  for index1, autore1 in enumerate(Rij_new[2]):
63                      for index2, autore2 in
                           enumerate(Rij_new[2]):
64                          if autore1 == autore2:
65                              continue
66                          elif autore1[0] == autore2[0]:
```

```
                                Rij_new[2][index1] =
                                    [autore1[0], max(autore1[1],
                                    autore2[1])]
                                Rij_new[2][index2] =
                                    [autore2[0], max(autore1[1],
                                    autore2[1])]


                    Rij_new[2] = [t for t in (set(tuple(i) for i
                        in Rij_new[2]))]

                    Rij_label['attr_dict']['Start'] = Rij_new[0]
                    Rij_label['attr_dict']['End'] = Rij_new[1]
                    Rij_label['attr_dict']['Authors'] =
                        Rij_new[2]

                    new_time = tuple(x for x in
                        Rij_label['attr_dict'].values())

                    G_new.add_edge(t[0], t[2], attr_dict =
                        Rij_label['attr_dict'], time = new_time)

    return G_new
```

La funzione triangles è:

```
def get_triangles(G, i, j):
    result = []
    for nodo in G.nodes():
        if (G.has_edge(i, nodo) and G.has_edge(nodo, j)) or \
           (G.has_edge(i, nodo) and G.has_edge(j, nodo)) or \
           (G.has_edge(nodo, i) and G.has_edge(nodo, j)) or \
           (G.has_edge(nodo, i) and G.has_edge(j, nodo)):

            triangle = (i, nodo, j)
            result.append(triangle)

    return result
```

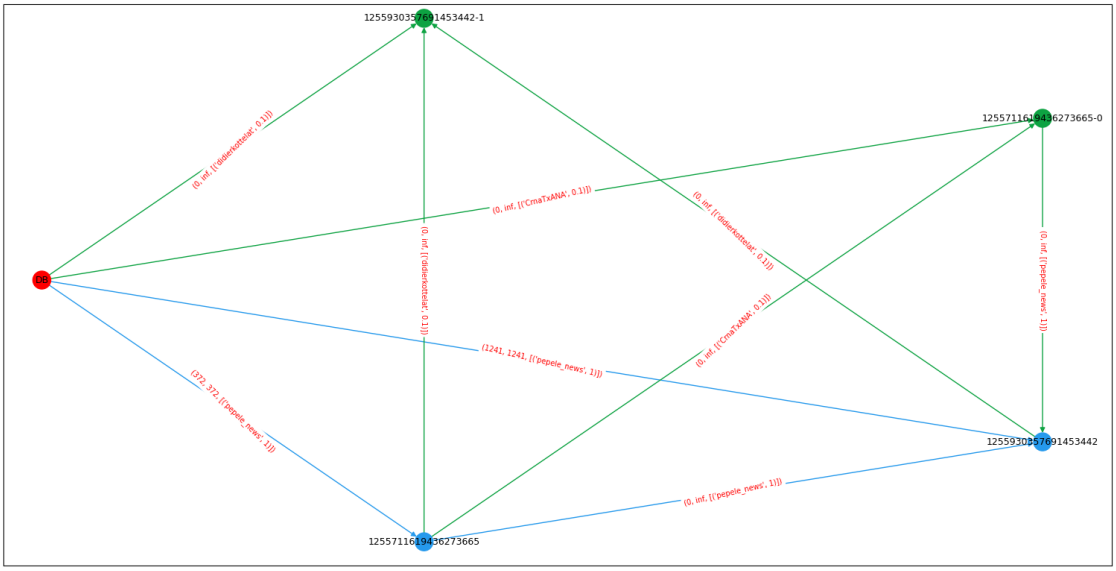

Figura 8. Grafo di esempio senza algoritmi applicati

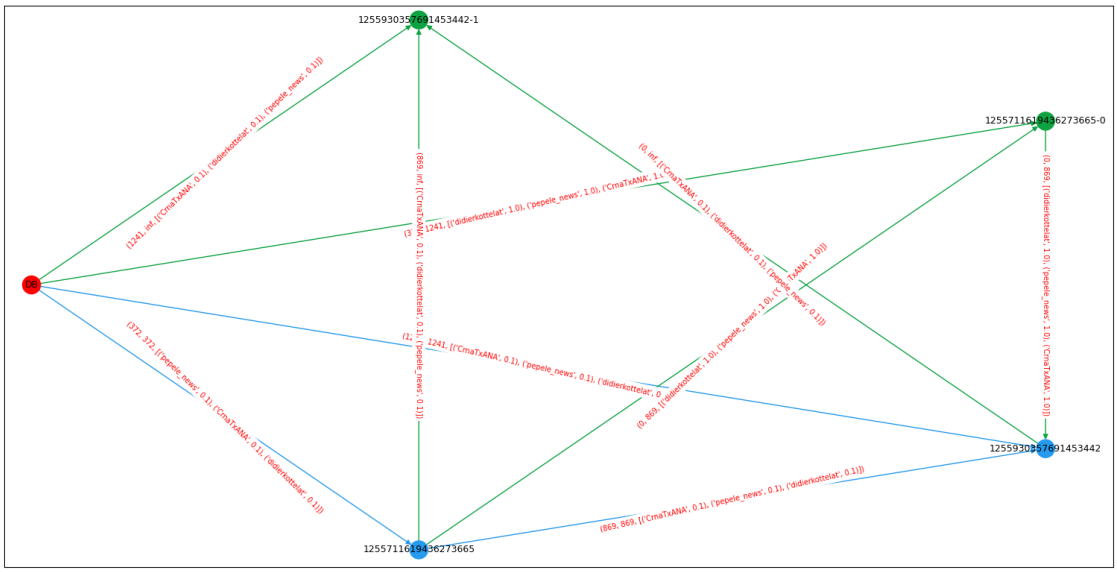

Figura 9. Grafo di esempio con Path-Consistency e Similarità degli intervalli

Gestione

Per fornire una gestione minimale delle funzioni descritte in precedenza ho creato un semplice menù, il codice è il seguente:

```
def menu():
    print('*' * 55)
    comando = input("Scegliere uno dei seguenti comandi:\n"\
                    "1) Applica l'algoritmo Path Consistency
                       al grafo.\n"\
                    "2) Salvare il grafo creato in un
                       file.\n"\
                    "3) Caricare un grafo da un file.\n"\
                    "4) Visualizzare il grafo
                       creato/modificato o caricato dal
                       file.\n"\
                    "0) Pulire la console.\n"\
                    "q) exit()\n\n")

    if comando == 'q':
        print('*' * 55)
        return comando
    else:
        print('*' * 55)
        return int(comando)

def loop(G):
    try:
        x = menu()
        while x in {0, 1, 2, 3, 4, 'q'}:
            if x == 'q':
                print("Sto chiudendo il programma...")
                break
            if x == 0:
                os.system("cls")
                x = menu()
            if x == 1:
                G = path_consistency(G)
                print("Algoritmo applicato.")
                x = menu()
            if x == 2:
                save_file_edgelist(G)
                x = menu()
            if x == 3:
                G = read_file_edgelist()
                x = menu()
            if x == 4:
```

```
39                  print_graph(G)
40                  x = menu()
41          if x not in {0, 1, 2, 3, 4, 'q'}:
42              raise ValueError
43      except ValueError:
44          print("Hai inserito una scelta del men  non valida,
               ritenta!\n")
45          loop(G)
```

Questo permette di far eseguire un'operazione alla volta tramite l'inserimento di un numero da 0 a 4, nel caso venga inserito il carattere 'q' il programma terminerà.

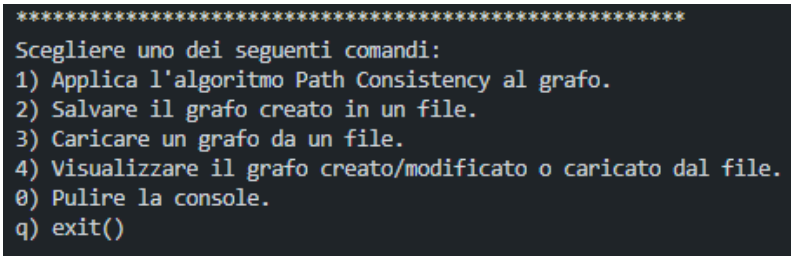

Figura 10. Screenshot del menù

Sitografia

[1] https://www.kaggle.com/smid80/coronavirus-covid19-tweets-late-april/data?select=2020-04-30+Coronavirus+Tweets.CSV

[2] http://www.convertcsv.com/csv-to-json.htm

[3] https://app.diagrams.net/

[4] https://www.postgresql.org/download/

[5] https://stackoverflow.com/questions/55555095/how-to-add-commas-in-between-json-objects-present-in-a-txt-file-and-then-conver

[6] https://www.geeksforgeeks.org/reading-and-writing-json-to-a-file-in-python/

[7] https://stackoverflow.com/questions/48604563/creating-a-data-structure-from-json-using-python

[8] https://networkx.github.io/documentation/stable/index.html#

[9] https://stackoverflow.com/questions/49268016/creating-a-dictionary-from-2-lists-with-duplicate-keys

[10] https://stackoverflow.com/questions/44395560/how-to-remove-duplicate-values-from-dict

[11] https://stackoverflow.com/questions/1653970/does-python-have-an-ordered-set

[12] https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.traversal.depth_first_search.dfs_preorder_nodes.html#networkx.algorithms.traversal.depth_first_search.dfs_preorder_nodes

[13] https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.layout.shell_layout.html?highlight=shell_layout#networkx.drawing.layout.shell_layout

[14] https://matplotlib.org/index.html

[15] https://networkx.github.io/documentation/stable/reference/readwrite/generated/networkx.readwrite.edgelist.write_edgel

ist.html?highlight=write_edgelist#networkx.readwrite.edgelist.write_edgelist
[16] https://networkx.github.io/documentation/stable/reference/generated/networkx.classes.function.is_empty.html?highlight=is_empty#networkx.classes.function.is_empty
[17] https://networkx.github.io/documentation/stable/reference/readwrite/generated/networkx.readwrite.edgelist.read_edgelist.html?highlight=read_edgelist#networkx.readwrite.edgelist.read_edgelist
[18] https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.traversal.breadth_first_search.bfs_edges.html#networkx.algorithms.traversal.breadth_first_search.bfs_edges
[19] https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.DiGraph.get_edge_data.html?highlight=get_edge_data#networkx.DiGraph.get_edge_data