IMDB graph exploration

Project for the exam of *advanced algorithms and graph mining*

Candidate: Edoardo Wijaya Grappolini

Lecturers:
Andrea Marino *and* Massimo Nocentini

Chosen questions

- **1.G)** Considering only the movies up to year x with x in {1930,1940,1950,1960,1970,1980,1990,2000,2010,2020}, write a function which, given x, computes the average number of movies per actor up to year x.
- **2.3)** Considering only the movies up to year x with x in {1930,1940,1950,1960,1970,1980,1990,2000,2010,2020} and restricting to the largest connected component of the graph. Approximate the closeness centrality for each node. Who are the top-10 actors?
- **3.III)** Which is the pair of movies that share the largest number of actors?
- **4.-)** Which is the pair of actors who collaborated the most among themselves?

Graph construction

Graph construction

.tsv read by pandas, structure achieved ⇒

13509	Abuda, Rob	Blind Eyes (2003) (V)	2003
13510	Abude, Leonardo Cesare	La passione di Giosu? l'Ebreo (2005)	2005
13511	Abudi, Ilan	Shitat HaShakshuka (2008)	2008
13512	Abudin, Mohammed	Go-Con! Japanese Love Culture (2000)	2000
13513	Abudlkar, Tahira	Belles & Whistles (2002)	2002
13514	Abudo, Abijiang	Guangzhou laile Xinjiang wa (1995)	1995

Year separated by movie title by a simple regex applied through pandas

```
df[2] = df[1].str.extract(r'(\(\d{4})', expand=True)
```

Graph construction

Having the dataframe, I generate the actor and movies nodes.

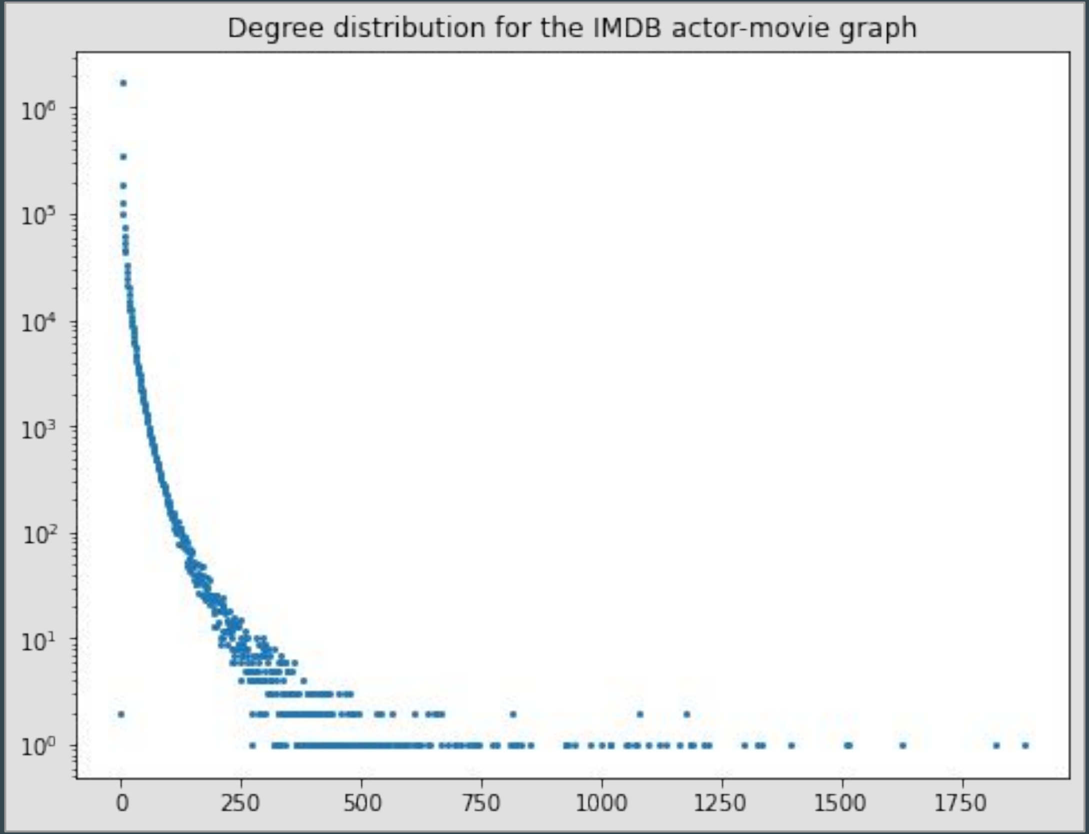
Nodes

```
# get all that will be the actor nodes
actors = df.actor.unique() #*
# find all unique movies records, create a list of tuples of type
# (‘Movie title’, {year: xxxx} which allows graph population through networkx method :)
movies_dict = df.drop(columns='actor').drop_duplicates().set_index('movie').to_dict('index')
movies_tuples_list = [(k, v) for k, v in movies_dict.items()]

oriGinal = nx.Graph()
oriGinal.add_nodes_from(actors, bipartite = 0) # 0 is actors, 1 is movies
print(f"Number of nodes after adding actors is {oriGinal.number_of_nodes()}")
oriGinal.add_nodes_from(movies_tuples_list, bipartite = 1)
edges = df.to_records(index=False)
oriGinal.add_edges_from(edges)
G = nx.convert_node_labels_to_integers(oriGinal, label_attribute='original_name')
```

Edges

Graph construction



Degree distribution

Q.1.G. Average number of movies per actor up to year *x*

Strategy

Given the graph:

1. Create subgraph composed of all actors, and the movies up to year *x*
2. Get the cardinalities of the neighborhoods of each actor node of subgraph
3. Calculate the average (two flavors):
    a. Considering all actors, regardless of the year
    b. Considering only actors that made at least one movie

Cycle for all the years of interest

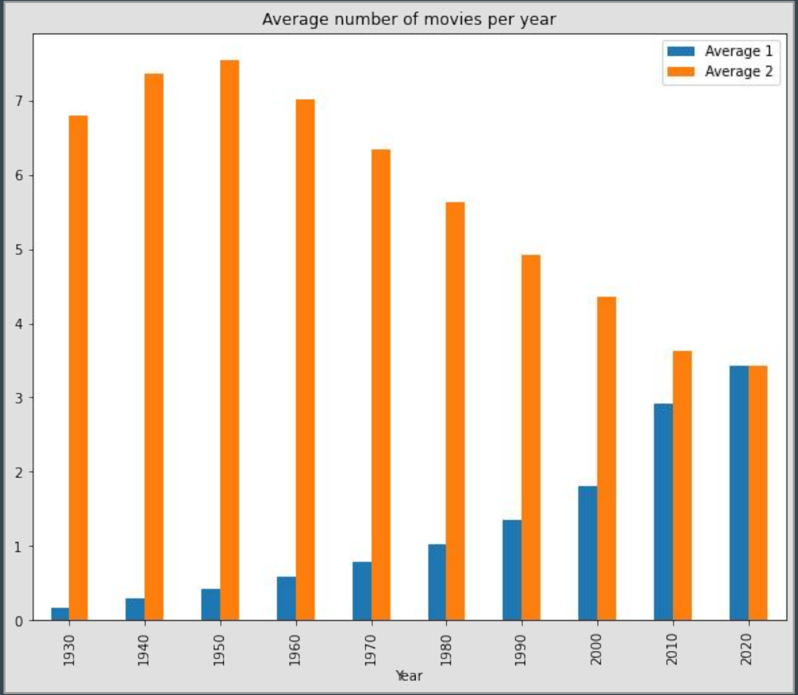
Q.1.G. Average number of movies per actor up to year *x*

Pythonized version

```
def avgMoviesPerActorUpToYear(graph, act_nodes, mv_nodes, year):
    # get movies nodes up to a certain year
    movies_up_to_year = {x for x,y in graph.nodes(data=True) if y['bipartite'] == 1 and y['year'] <= year}
    # define the nodes of interest for the year (i.e. both movie and actor nodes)
    nodes_subset = movies_up_to_year.union(act_nodes)
    subgraph = graph.subgraph(nodes_subset)
    assert subgraph.number_of_nodes() == len(nodes_subset)
    subgraph_actor_nodes = {n for n, d in subgraph.nodes(data=True) if d["bipartite"] == 0}
    degrees = subgraph.degree(nbunch = subgraph_actor_nodes)
    deg_data = pd.DataFrame(degrees)
    sol = (year, deg_data[1].mean(), deg_data[1].replace(0, np.NaN).mean()) #convenient for output later

    return sol
```

Q.1.G. Average number of movies per actor up to year *x* - Results



	Avg_1	Avg_2
1930	0.163	6.790
1940	0.300	7.355
1950	0.427	7.533
1960	0.583	7.018
1970	0.782	6.345
1980	1.030	5.627
1990	1.350	4.925
2000	1.813	4.356
2010	2.912	3.624
2020	3.427	3.427

Q.2.3 Approximate the closeness centrality for each node.

Q.2.3 Approximate the closeness centrality for each node.

➪ Up to a certain year x, with x in {1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020}
➪ Restricting to the largest connected component

In this case, literal implementation of the pseudocode in the original article [*] was implemented

1. Let k be the number of iterations needed to obtain the desired error bound.

2. In iteration i, pick vertex v_i uniformly at random from G and solve the SSSP problem with v_i as the source.

3. Let

$$\hat{c}_u = 1 / \sum_{i=1}^{k} \frac{n\, d(v_i, u)}{k(n-1)}$$

be the centrality estimator for vertex u.

[*][Eppstein, Wang] Fast approximation of centrality

Q.2.3 Approximate the closeness centrality for each node.

For year selection, same strategy as previous (subset and union); in this case, also largest CC through nx method

```
def closenessCentralityUpToYear(graph, act_nodes, year, k = None, epsilon = None):
    # fundamental lines for the methods are shown. Before: subselection of nodes up to a certain year and largest CC
      [...]
    # calculate sample size, if an epsilon is specified
    if epsilon is not None:
        import math
        k = math.ceil(math.log(cc_subgraph.number_of_nodes())/math.pow(epsilon, 2))

    # sample k nodes
    starting_nodes = random.sample(list(largest_cc), k)
    # find shortest paths starting from sample nodes
    sssp_s = list(map(lambda x: nx.single_source_shortest_path_length(cc_subgraph, x), tqdm(starting_nodes)))
    sssp_s_dict = {}
    for starting_node in tqdm(starting_nodes):
        sssp_s_dict[starting_node] = nx.single_source_shortest_path_length(cc_subgraph, starting_node)
    n = len(largest_cc)
    distances_df = pd.DataFrame(sssp_s).T
    distances_df['centrality'] = distances_df.mean(numeric_only=True, axis=1).apply(lambda x: 1/(x*(n/(n-1))))
    return distances_df
```

Q.2.3 Approximate the closeness centrality for each node.

For year selection, same strategy as previous (subset and union); in this case, also largest CC through nx method

```
def closenessCentralityUpToYear(graph, act_nodes, year, k = None, epsilon = None):
    # fundamental lines for the methods are shown. Before: subselection of nodes up to a certain year and largest CC
      [...]
    starting_nodes = random.sample(list(largest_cc), k)
    sssp_s = list(map(lambda x: nx.single_source_shortest_path_length(cc_subgraph, x), tqdm(starting_nodes)))
    sssp_s_dict = {}
    for starting_node in tqdm(starting_nodes):
        sssp_s_dict[starting_node] = nx.single_source_shortest_path_length(cc_subgraph, starting_node)
    n = len(largest_cc)
    distances_df = pd.DataFrame(sssp_s).T
    distances_df['centrality'] = distances_df.mean(numeric_only=True, axis=1).apply(lambda x: 1/(x*(n/(n-1))))
    return distances_df
```

	v_1	v_2	v_3
s_1	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$
s_2	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$
s_3	$d_{3,1}$	$d_{3,2}$	$d_{3,3}$

s_i - i-th source
v_j - j-th node
$d_{i,j}$ - $d(s_i, v_j)$

Q.2.3 Approximate the closeness centrality for each node.

For year selection, same strategy as previous (subset and union); in this case, also largest CC through nx method

```
def closenessCentralityUpToYear(graph, act_nodes, year, k = None, epsilon = None):
    # fundamental lines for the methods. Before: subselection of nodes up to a certain year and largest CC
      [...]
    starting_nodes = random.sample(list(largest_cc), k)
    sssp_s = list(map(lambda x: nx.single_source_shortest_path_length(cc_subgraph, x), tqdm(starting_nodes)))
    sssp_s_dict = {}
    for starting_node in tqdm(starting_nodes):
        sssp_s_dict[starting_node] = nx.single_source_shortest_path_length(cc_subgraph, starting_node)
    n = len(largest_cc)
    distances_df = pd.DataFrame(sssp_s).T
    distances_df['centrality'] = distances_df.mean(numeric_only=True, axis=1).apply(lambda x: 1/(x*(n/(n-1))))
    return distances_df
```

	v_1	v_2	v_3
s_1	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$
s_2	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$
s_3	$d_{3,1}$	$d_{3,2}$	$d_{3,3}$

T

	s_1	s_2	s_3
v_1	$d_{1,1}$	$d_{2,1}$	$d_{3,1}$
v_2	$d_{1,2}$	$d_{2,2}$	$d_{3,2}$
v_3	$d_{1,3}$	$d_{2,3}$	$d_{3,3}$

s_i - i-th source
v_j - j-th node
$d_{i,j}$ - $d(s_i, v_j)$

Q.2.3 Approximate the closeness centrality for each node.

For year selection, same strategy as previous (subset and union); in this case, also largest CC through nx method

```
def closenessCentralityUpToYear(graph, act_nodes, year, k = None, epsilon = None):
    # fundamental lines for the methods. Before: subselection of nodes up to a certain year and largest CC
      [...]
    starting_nodes = random.sample(list(largest_cc), k)
    sssp_s = list(map(lambda x: nx.single_source_shortest_path_length(cc_subgraph, x), tqdm(starting_nodes)))
    sssp_s_dict = {}
    for starting_node in tqdm(starting_nodes):
        sssp_s_dict[starting_node] = nx.single_source_shortest_path_length(cc_subgraph, starting_node)
    n = len(largest_cc)
    distances_df = pd.DataFrame(sssp_s).T
    distances_df['centrality'] = distances_df.mean(numeric_only=True, axis=1).apply(lambda x: 1/(x*(n/(n-1))))
    return distances_df
```

s_i - i-th source
v_j - j-th node
$d_{i,j}$ - $d(s_i, v_j)$

	v_1	v_2	v_3
s_1	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$
s_2	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$
s_3	$d_{3,1}$	$d_{3,2}$	$d_{3,3}$

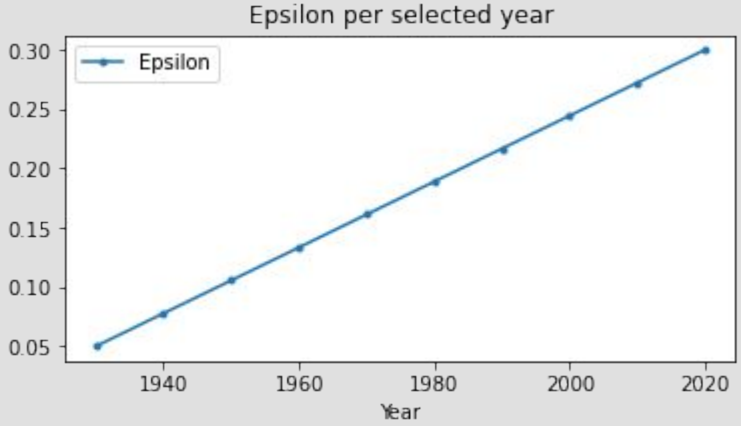
T

	s_1	s_2	s_3
v_1	$d_{1,1}$	$d_{2,1}$	$d_{3,1}$
v_2	$d_{1,2}$	$d_{2,2}$	$d_{3,2}$
v_3	$d_{1,3}$	$d_{2,3}$	$d_{3,3}$

...

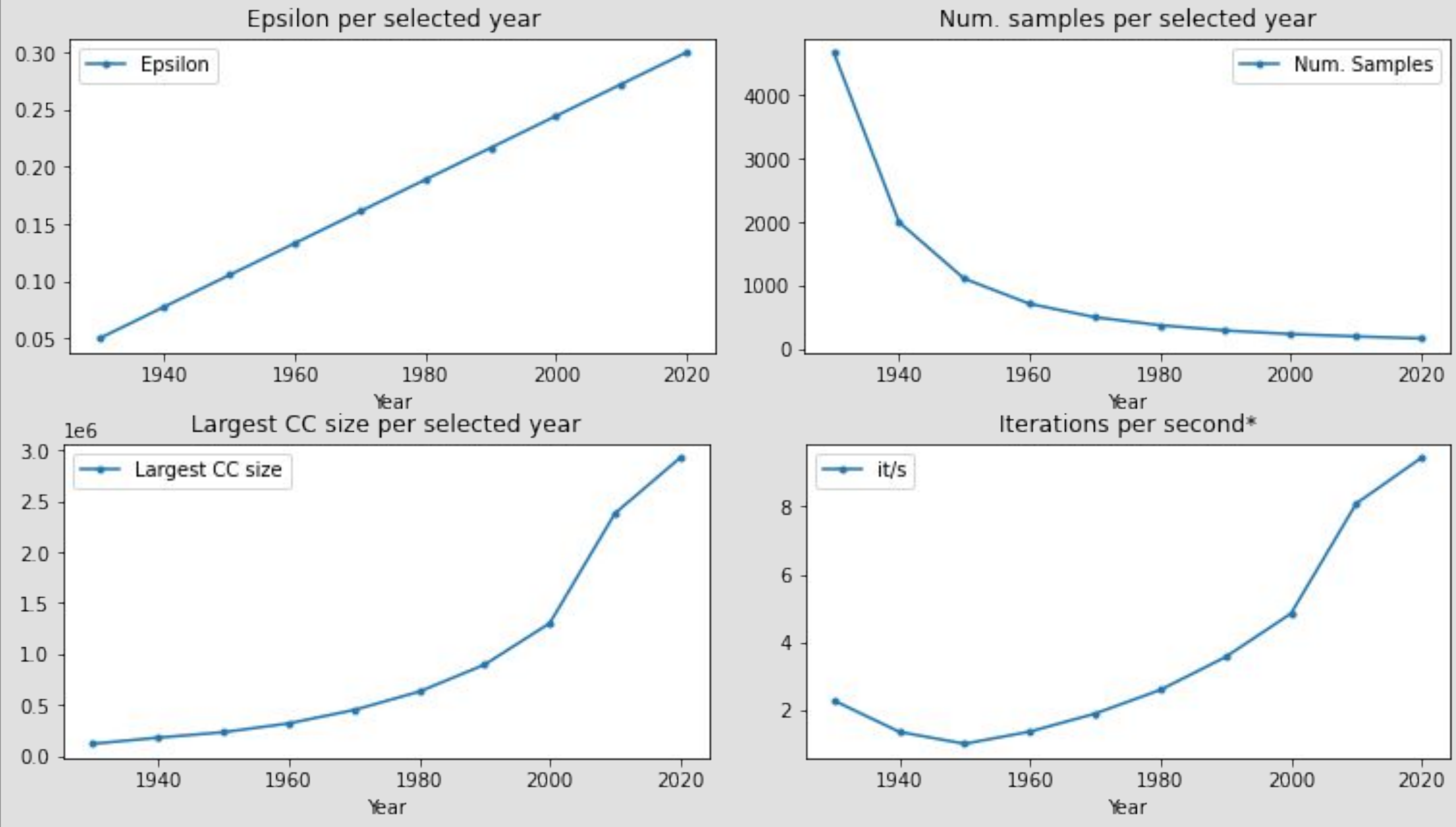
avg	centr
$đ(s_1)$	$\hat{c}(s_1)$
$đ(s_2)$	$\hat{c}(s_2)$
$đ(s_3)$	$\hat{c}(s_3)$

Q.2.3 Approximate the closeness centrality for each node.
- Preliminary results



Choice for epsilon followed a linear progression from 0.05 to 0.3 (unfortunately quite large), for each of the input years.
The choice was heuristically made, the objective was to have "*reasonable results*" (in terms of errors) in reasonable times.

Q.2.3 Approximate the closeness centrality for each node.
- Preliminary results



☹️ Worse epsilon for larger CCs had to be specified
One of many manifestation of 'the curse of dimensionality'?

it/s was not a scientific measurement due to too many variabilities.
Still nice to see qualitatively.

Q.2.3 Approximate the closeness centrality for each node.

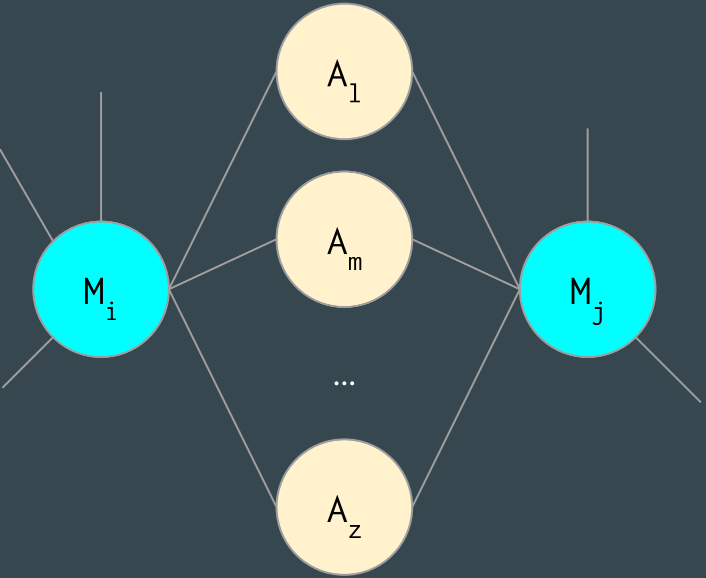
- Results

*colorations were made by hand, therefore coloration-completeness is not guaranteed

1930	['Bracey, Sidney', 'Fawcett, George', 'Beery, Noah (I)', 'Siegmann, George', 'Marshall, Tully', 'De Brulier, Nigel', 'Holmes, Stuart', 'Swickard, Josef', 'McDowell, Claire', 'Pitts, Zasu']
1940	['Steers, Larry', 'Bracey, Sidney', 'White, Leo (I)', 'Lucas, Wilfred', 'Semels, Harry (I)', 'Corrado, Gino', 'Mulhall, Jack', 'Brady, Ed (III)', 'Hoyt, Arthur', "O'Malley, Pat (I)"]
1950	['Steers, Larry', 'Corrado, Gino', 'Flowers, Bess', 'Semels, Harry (I)', 'Harris, Sam (II)', 'White, Leo (I)', 'Holmes, Stuart', 'Blue, Monte', "O'Malley, Pat (I)", 'Hagney, Frank']
1960	['Flowers, Bess', 'Harris, Sam (II)', 'Steers, Larry', 'Corrado, Gino', 'Farnum, Franklyn', 'Chefe, Jack', 'Auer, Mischa', 'Miller, Harold (I)', "O'Brien, William H.", 'Holmes, Stuart']
1970	['Harris, Sam (II)', 'Flowers, Bess', 'Tamiroff, Akim', 'Welles, Orson', 'Farnum, Franklyn', 'Miller, Harold (I)', 'Frees, Paul', 'Sayre, Jeffrey', 'Stevens, Bert (I)', 'Quinn, Anthony (I)']
1980	['Flowers, Bess', 'Harris, Sam (II)', 'Welles, Orson', 'Miller, Harold (I)', 'Tamiroff, Akim', 'Farnum, Franklyn', 'Sayre, Jeffrey', 'Niven, David (I)', 'Quinn, Anthony (I)', 'Holmes, Stuart']
1990	['Welles, Orson', 'Hitler, Adolf', 'Carradine, John', 'Quinn, Anthony (I)', 'Flowers, Bess', 'Douglas, Kirk (I)', 'Sinatra, Frank', 'Fonda, Henry', 'Wayne, John (I)', 'Harris, Sam (II)']
2000	['Steiger, Rod', 'Sutherland, Donald (I)', 'Lee, Christopher (I)', 'Hitler, Adolf', 'Welles, Orson', 'Caine, Michael (I)', 'Loren, Sophia', 'Schell, Maximilian', 'York, Michael (I)', 'Hopper, Dennis']
2010	['Reagan, Ronald (I)', 'Hitler, Adolf', 'Madonna', 'De Niro, Robert', 'Schwarzenegger, Arnold', 'Kidman, Nicole', 'Pitt, Brad', 'Hanks, Tom', 'Caine, Michael (I)', 'Hopper, Dennis']
2020	['Jackson, Samuel L.', 'Hitler, Adolf', 'Depp, Johnny', 'Sheen, Martin', 'De Niro, Robert', 'Willis, Bruce', 'Madonna', 'Lee, Christopher (I)', 'Clooney, George', 'Hanks, Tom']

Q.3.III Which is the pair of movies that share the largest number of actors?

Q.3.III Which is the pair of movies that share the largest number of actors?



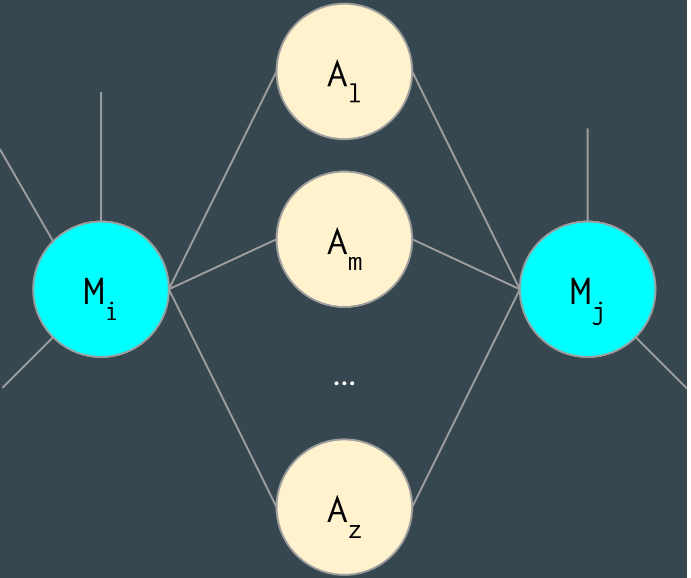
Reasoning:

Given two movie nodes M_i and $M_j \in G$, the common actors are represented by $\mathbf{N}(M_i) \cap \mathbf{N}(M_j)$

$\Rightarrow$ find maximum intersection

Q.3.III Which is the pair of movies that share the largest number of actors?

Basic strategy: for each two different nodes, intersect the adjacency lists to find common neighbors.



Main idea to solve this would be to do an intersection of the edges of each of the movies.

Doing the intersections of all sets can become very expensive timewise.

Given an unordered set of sets $\hat{S} = \{S_1, .., S_N\}$ for any $N \in \mathbb{N}$ s.t. $|S_i| \leq M$ for any $i = 1, .., N$ and $M \in \mathbb{N}$; finding the max intersection would cost $\mathcal{O}(N)$, and $\mathcal{O}(\min\{|U_i|, |U_j|\})$ (python documentation), reaching $\mathcal{O}(N * M)$.

In this case I use the following simple observations:

- For any two given sets $S_i \neq S_j$ (for $i \neq j = 1, .., N$) it is true that $|S_i \cap S_j| \leq \min\{|S_i|, |S_j|\}$
- Let m be the maximum intersection found until a certain iteration. Then if U_i (or U_j) is s.t. $|U_i| < m$ (or $|U_j| < m$) then necessarily $|U_i \cap U_j| < m$, i.e. it is not necessary to do the intersection to infer that the cardinality of that intersection would now surpass the current max. Therefore, it's possible to only check the cardinality and skip the calculation of the intersections.

With this heuristic, although the formal complexity would be essentially the same, in practice a lot of the intersections are skipped.

Q.3.III Which is the pair of movies that share the largest number of actors?

```
def moviesWithMaxCommonNumActors(graph, mv_nodes):
    mv_act = nx.to_dict_of_lists(graph)
    print(mv_act[1])
    current_solution = (None, None)
    current_max = 0
    for movie in tqdm(mv_nodes):
        if len(mv_act[movie]) >= current_max:
            for second_movie in mv_nodes:
                if len(mv_act[second_movie]) >= current_max and movie != second_movie:
                    temp = len(set(mv_act[movie]).intersection(set(mv_act[second_movie])))
                    if current_max < temp:
                        current_solution = (movie, second_movie)
                        current_max = temp
                        print(f"Current max: {current_max}")
    return current_solution
```

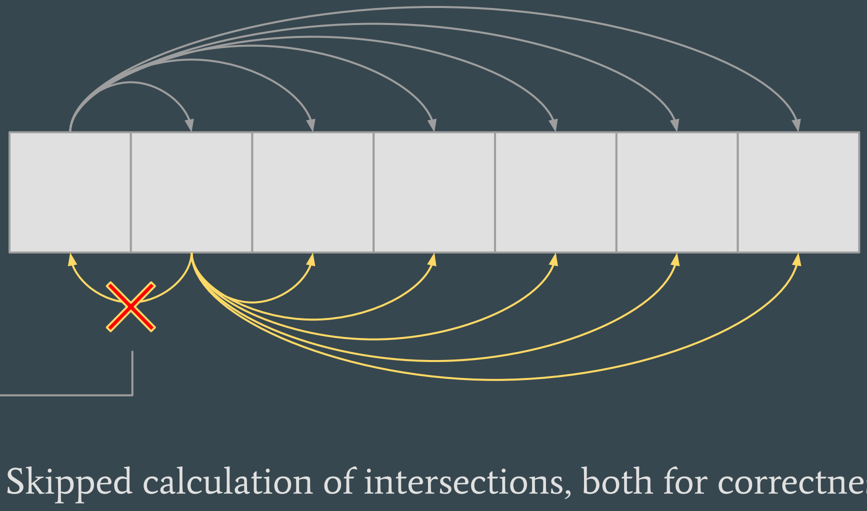
Q.3.III Which is the pair of movies that share the largest number of actors? - Results

'Kingdom Hearts II (2005) (VG)' ↔ 'Kingdom Hearts II: Final Mix+ (2007) (VG)'
194 common actors (mainly, voice actors)

Q4. Building the actor graph: which is the pair of actors who collaborated the most among themselves?

Q4. Building the actor graph: which is the pair of actors who collaborated the most among themselves?

➪ Graph constructed from the given dataset (not from previous graph due to memory limitations)
For the graph generation:



Through pandas, achieved strutture as dict of lists, of structure:

```
{‘movie 1’: [‘actor 1’, …, ‘actor n’],
   …
}
```

Skipped calculation of intersections, both for correctness (to avoid double counting an edge) and performance reasons.

Q4. Building the actor graph: which is the pair of actors who collaborated the most among themselves?

```
def constructGraphAndFindMaxCollaborationGivenActorsGraph(imdb_df_f):
    actor_graph_dict = imdb_df_f.groupby('movie')['actor'].apply(list).to_dict()
    mass = 0
    sol = (None, None)
    for movie in tqdm(actor_graph_dict):
        current_actors_list = actor_graph_dict[movie]
        for i in range(len(current_actors_list)):
            for j in range(i+1, len(current_actors_list)):
                if current_actors_list[i] != current_actors_list[j]:
                    if not actor_graph.has_edge(current_actors_list[i], current_actors_list[j]):
                        actor_graph.add_edge(current_actors_list[i], current_actors_list[j], weight=1)
                    else:
                        actor_graph[current_actors_list[i]][current_actors_list[j]]['weight'] += 1
                        if actor_graph[current_actors_list[i]][current_actors_list[j]]['weight'] > mass:
                            mass = actor_graph[current_actors_list[i]][current_actors_list[j]]['weight']
                            sol = (current_actors_list[i], current_actors_list[j])
    return (actor_graph, mass, sol)
```

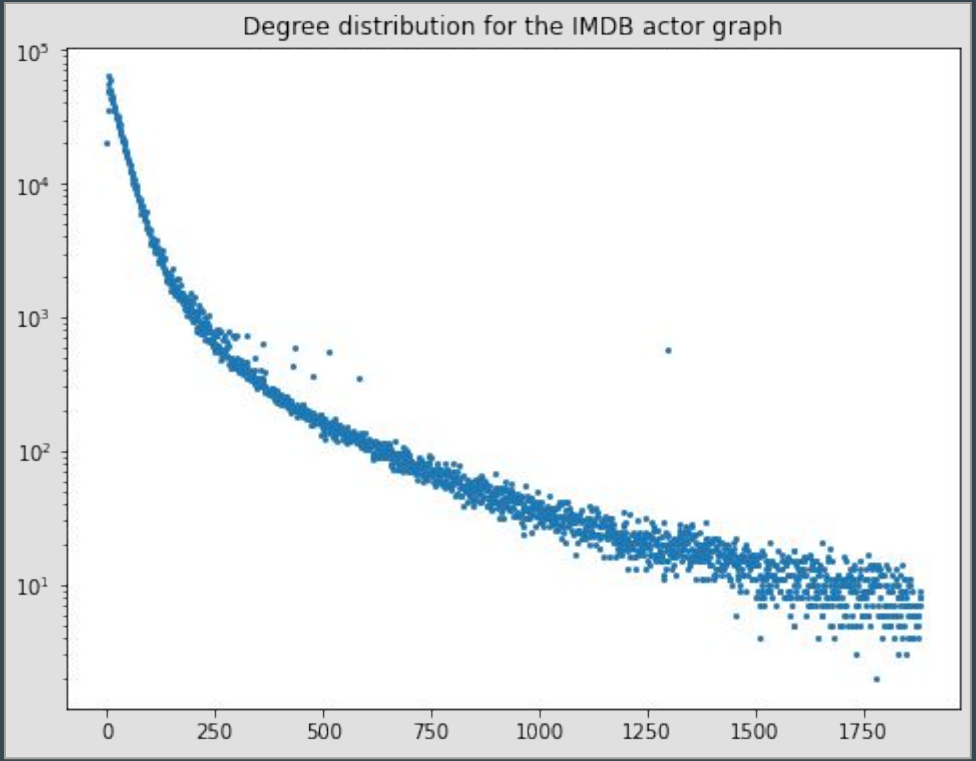
Q4. Building the actor graph: which is the pair of actors who collaborated the most among themselves?

```
archi = gr.edges(data=True)
massimo = 0
sol = (None, None)
hist = [0]*500
i = 0
sol = (None, None)
for arco in archi:
    curr_weight = arco[2]['weight']
    #print(curr_weight)
    if massimo < curr_weight:
        massimo =  curr_weight
        sol = (arco[0], arco[1])

    hist[curr_weight] += 1
    i+=1
```

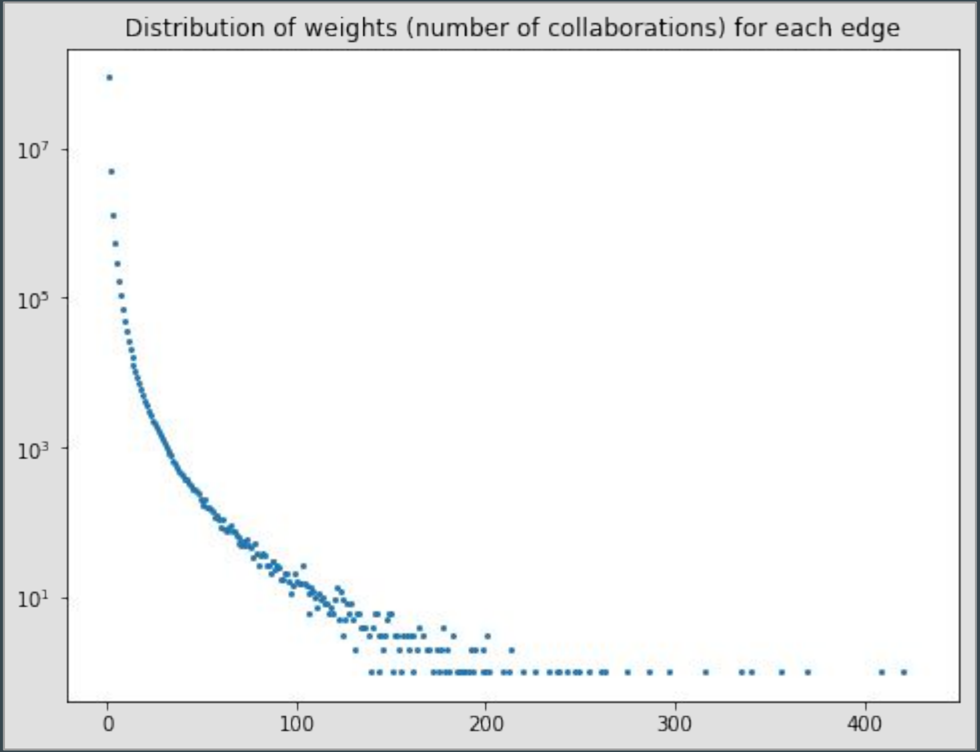
As an alternative.. classic algorithm for finding max, given the constructed graph.

Q4. Building the actor graph: which is the pair of actors who collaborated the most among themselves?



Degree distribution for the actor graph
x axis: degree
y axis: number of occurrences of the degree

Q4. Building the actor graph: which is the pair of actors who collaborated the most among themselves?



Horizontal axis: edge weight
Vertical axis: number of edges with said weight

Thank you