An exact and fast algorithm for computing top-k closeness centrality

Luca Lombardo

Univeristy of Pisa - Department of Mathematics

Abstract

Understanding and investigating social structures is essential in the modern world. Through the use of networks and graph theory we can find the most central elements in a community. In particolar, given a connected graph $G = (V, E)$, the closeness centrality of a vertex v is defined as $\frac{n-1}{\sum_{w \in V} d(v,w)}$. This measure can be seen as the efficiency of a node to pass information through all the other nodes in the graph. In this paper we will discuss and algorithm and its result for finding the top-k most central elements in web-scale graphs. As a case study, we are going to use the IMDB collaboration network, building two completely different graphs and analyzing their proprieties.

Contents

1 Introduction

A graph $G = (V, E)$ is a pair of a sets. Where $V = \{v_1, ..., v_n\}$ is the set *nodes*, and $E \subseteq V \times V,\ E = \{(v_i, v_j), ...\}$ is the set of *edges* (with $|E| = m \leq n^2$).

In this paper we discuss the problem of identifying the most central nodes in a network using the measure of *closeness centrality.* Given a connected graph, the closeness centrality of a node $v \in V$ is defined as the reciprocal of the sum of the length of the shortest paths between the node and all other nodes in the graph. Normalizing we obtain the following formula:

$$c(v) = \frac{n-1}{\displaystyle\sum_{w \in V} d(v, w)} \tag{1}$$

where n is the cardinality of V and $d(v, w)$ is the distance between $v, w \in V$. This is a very powerful tool for the analysis of a network: it ranks each node telling us the most efficient ones in spreading information through all the other nodes in the graph. As mentioned before, the denominator of this definition give us the length of the shortest path between two nodes. This means that for a node to be central, the average number of links needed to reach another node has to be low. The goal of this paper is to computer the k vertices with the higher closeness centrality.

As case study we are using the collaboration network in the *Internet Movie Database* (IMDB). We are going to consider two different graphs. For the first one we define an undirected graph $G = (V, E)$ where

- the vertex V are the actor and the actress
- the non oriented edges in E links the actors and the actresses if they played together in a movie.

For the second one we do the opposite thing: we define an undirected graph $G = (V, E)$ where

- the vertices V are the movies
- the non oriented edges in E links two movies if they have an actor or actress in common

1.1 The Problem

Since we are dealing with a web-scale network any brute force algorithm would require years to end. The main difficulty here is caused by the computation of distance $d(v, w)$ in (1). This is a well know problem known as *All Pairs Shortest Paths or APSP problem.*

We can solve the APSP problem either using the fast matrix multiplication or, as made in this paper, implementing a breath-first-search (BFS) method. There are several reason to prefer this second approach over the first one in this type of problems.

A graph is a data structure and we can describe it in different ways. Choosing one over another can have an enormous impact on performance. In this case, we need to remember the type of graph that we are dealing with: a very big and sparse one. The fast matrix multiplication implement the graph as an $n \times n$ matrix where the position (i, j) is zero if the nodes i, j are not linked, 1 (or a generic number if weighted) otherwise. This method requires $O(n^2)$ space in memory, that is an enormous quantity on a web-scale graph. Furthermore the time complexity is $O(n^{2.373} \log n)\}$ `[Zwick 2002; Williams 2012]`

Using the BFS method the space complexity is $O(n + m)$, which is a very lower value compared to the previous method. In terms of time, the complexity is $O(nm)$. Unfortunately, this is not enough to compute all the distances in a reasonable time. It is also been proven that this method can not be improved. In this paper I propose an exact algorithm to compute the top-k nodes with the higher closeness centrality.

2 The algorithm

In a connected graph, given a node $v \in V$, we can define the its farness as

$$f(v) = \frac{1}{c(v)} = \frac{1}{n-1} \sum_{w \in V} d(v,w) \tag{2}$$

where $c(v)$ is the closeness centrality defined in (1). Since we are working with a disconnected graph, a natural generalization of this formula is

$$f(v) = \frac{1}{c(v)} = \frac{1}{r(v)-1} \sum_{w \in V} d(v,w) \tag{3}$$

where $r(v) = |R(v)|$ is the cardinality of the set of reachable nodes from v. To avoid any problem during the computation, this formula still needs to be modified. Let's assume that the node v that we are considering has just one link at distance 1 with another node w with *out-degree* 0. If we consider the formula (3) we will get a false result: v would appear to be very central, even if it's obviously very peripheral. To avoid this problem, we can generalize the formula (3) normalizing as suggested in [Lin 1976; Wasserman and Faust 1994; Boldi and Vigna 2013; 2014; Olsen et al. 2014]

$$f(v) = \frac{n-1}{(r(v)-1)^2} \sum_{w \in R(v)} d(v,w) \tag{4}$$

With the convention that in a case of $\frac{0}{0}$ we set the closeness of v to 0

2.1 The lower bound technique

During the computation of the farness, for each node, we have to compute the distance from that node and all the other ones reachable from it. Since we are dealing with millions of nodes, it's not possibile in a reasonable time. In order to compute only the top-k most central node we need to find a way to avoid computing BFS for nodes that won't be in the top-k.

The idea is to keep track of a lower bound on the farness for each node that we will compute. This will allow us to kill the BFS operation before reaches the end if the lower bound tell us that the node will not be in the top-k. More precisely:

- The algorithm will compute the farness of the first k nodes, saving them in a vector `top-actors`. From now on, this vector will be full.
- Then, for all the next vertices, it defines a lower bound

$$\frac{n-1}{(n-1)^2}(\sigma_{d-1} + n_d \cdot d) \tag{5}$$

where σ_d is the partial sum in (4) at the level of exploration d. The lower bound (5) is updated every time that we change level of exploration during the BFS. In this way, if at a change of level the lower bound of the vertex that we are considering is bigger than the $k-th$ element of `top-actors`, we can kill the BFS. The reason behind that is very simple: the vector `top-actors` is populated with the top-k nodes in order and the farness is inversely proportional to the closeness centrality. So if at that level the lower bound is already bigger than the last element of the vector, there is no need to compute the other level of the BFS since it will not be added in `top-actors` anyway.

The (5) it's a worst case scenario, and that makes it perfect for a lower bound. If we are at the level d of exploration, we have already computed the sum in (4) up to the level $d-1$. Then we need consider in our computation of the sum the current level of exploration: the worst case gives us that it's linked to all the nodes at distance d. We also put $r(v) = n$, in the case that our graph is strongly connected and all vertices are reachable form v.

Scrivere pseudocodice

3 The IMDB Case Study

The algorithm shown before can be applied to any dataset on which is possibile to build a graph on. In this case we are considering tha data taken from the *Internet Movie Database* (IMDB).

3.1 Data Structure

All the data used can be downloaded here: `https://datasets.imdbws.com/`

In particolar we're interest in 3 files

- `title.basics.tsv`
- `title.principals.tsv`
- `name.basics.tsv`
- `title.ratings.tsv`

Let's have a closer look to this 4 files:

title.basics.tsv

Contains the following information for titles:

- `tconst` (string) - alphanumeric unique identifier of the title
- `titleType` (string) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc)
- `primaryTitle` (string) – the more popular title / the title used by the filmmakers on promotional materials at the point of release
- `originalTitle` (string) - original title, in the original language
- `isAdult` (boolean) - 0: non-adult title; 1: adult title
- `startYear` (YYYY) – represents the release year of a title. In the case of TV Series, it is the series start year
- `endYear` (YYYY) – TV Series end year.
- `runtimeMinutes` – primary runtime of the title, in minutes
- `genres` (string array) – includes up to three genres associated with the title

title.principals.tsv

Contains the principal cast/crew for titles:

- `tconst` (string) - alphanumeric unique identifier of the title
- `ordering` (integer) – a number to uniquely identify rows for a given titleId
- `nconst` (string) - alphanumeric unique identifier of the name/person
- `category` (string) - the category of job that person was in
- `job` (string) - the specific job title if applicable
- `characters` (string) - the name of the character played if applicable

name.basics.tsv

Contains the following information for names:

- `nconst` (string) - alphanumeric unique identifier of the name/person
- `primaryName` (string)– name by which the person is most often credited
- `birthYear` – in YYYY format
- `deathYear` – in YYYY format if applicable
- `primaryProfession` (array of strings)– the top-3 professions of the person
- `knownForTitles` (array of tconsts) – titles the person is known for

title.ratings.tsv

Contains the following information for titles:

- `tconst` (string) - alphanumeric unique identifier of the title
- `averageRating` – weighted average of all the individual user ratings
- `numVotes` – number of votes the title has received

3.2 Filtering

This is a crucial section for the algorithm in this particolar case study. This raw data contains a huge amount of un-useful information that will just have a negative impact on the performance during the computation. We are going to see in detail all the modification made for each file. All this operation have been implemented using `python` and the `pandas` library.

Since we want to build two different graph, some consideration will have to be considered for the specific case. If nothing is told it means that the filtering of that file is the same for both graphs.

3.2.1 name.basics.tsv

For this file we only need the following columns

- `nconst`
- `primaryTitle`
- `primaryProfession`

Since all the actors starts with the string `nm0` we can remove it to clean the output. Furthermore a lot of actors/actresses do more than one job (director etc..). To avoid excluding important actors we consider all the ones that have the string `actor/actress` in their profession. In this way, both someone who is classified as `actor` or as `actor, director` is taken into consideration.

Then we can generate the final filtered file `Attori.txt` that has only two columns: `nconst` and `primaryName`

3.2.2 title.basics.tsv

For this file we only need the following columns

- `tconst`
- `primaryTitle`
- `isAdult`
- `titleType`

Since all the movies starts with the string `t0` we can remove it to clean the output. In this case, we also want to remove all the movies for adults. This part can be optional if we are interest only in the closeness and harmonic centrality. Even if the actors and actresses of the adult industry use to make a lot of movies together, this won't alter the centrality result. As we know, an higher closeness centrality can be seen as the ability of a node to spread efficiently information in the network. Including the adult industry would lead to the creation of a very dense and isolated neighborhood. But none of those nodes will have an higher closeness centrality because they only spread information in their community. This phenomenon will be discussed more deeply in the analysis of the graph visualized.

We can also notice that there is a lot of *junk* in IMDb. To avoid dealing with un-useful data, we are considering all the non-adult movies in this whitelist

- `movie`
- `tvSeries`
- `tvMovie`
- `tvMiniSeries`

The reason to only consider this categories is purely to optimize the performance during the computation. On IMDb each episode is listed as a single element: to remove them without loosing the most important relations, we only consider the category `tvSeries`. This category list a TV-Series as a single element, not divided in multiple episodes. In this way we will loose some of the relations with minor actors that may appear in just a few episodes. But we will have preserved the relations between the protagonists of the show.

Then we can generate the final filtered file `FilmFiltrati.txt` that has only two columns: `tconst` and `primaryTitle`

3.2.3 title.principals.tsv

This file is needed for the analysis of both graphs, but there some different observation between them. For the both we only need the following columns

- `tconst`
- `nconst`
- `category`

As before, we clean the output removing unnecessary strings.

Actors Graph

Using the data obtained before we create an array of unique actor ids (`nconst`) and an array of how may times they appear (`counts`). This will give us the number of movies they appear in. And here it comes the core of the optimization for this graph. Let's define a constant `MINMOVIES`. This integer is the minimum number of movies that an actor needs to have made in his carrier to be considered in this graph. The reason to do that it's purely computational.

If an actor/actress has less then a reasonable number of movies made in his carrier, there is an high probability that he/she has an important role in our graph during the computation of the centralities.

Movies Graph

For this graph we don't need any optimization on this file. We just clean clean the output and leave the rest as it is

At the end, for both graph, we can finally generate the file `Relazioni.txt` containing the columns `tconst` and `nconst`

3.2.4 title.ratings.tsv

This file is necessary just in the analysis of the movie graph, it won't be even downloaded for the analysis of the actors graph. We will only need the following columns

- `tconst`
- `numVotes`

The idea behind the optimization made in this file is the same that we have used before with the `MINMOVIES` technique. We want to avoid computing movies that are not central with an high probability. To do that we consider the number of votes that each movies has received on the IMDB website. To do that we introduce the constant `VOTES`, considering only the movies with an higher number of votes.During the analysis we will change this value to see how it effects the list of the top-k most central movies.

In this case we don't have to generate a new file, we can apply this condition to `FilmFiltrati.txt`

4 An overview of the code

The algorithm implement is multi-threaded and written in C++

4.1 Data structures

In this case we are working with two simple `struct` for the classes *Film* and *Actor*

```
struct Film {
    string name;
    vector<int> actor_indicies;
};

struct Actor {
    string name;
    vector<int> film_indices;
};
```

Then we need two dictionaries build like this

```
map<int, Actor> A; // Dictionary {actor_id (key): Actor (value)}
map<int, Film> F; // Dictionary {film_id (key): Film (value)}
```

We are considering the files `Attori.txt` and `FilmFiltrati.txt`, we don't need the relations one for now. Once that we have read this two files, we loop on each one brutally filling the two dictionaries created before. If a line is empty, we skip it. We are using a try and catch approach. Even if the good practice is to use it only for a specific error, since we are outputting everything on the terminal it makes sense to *catch* any error.

```
void DataRead()
{
    ifstream actors("data/Attori.txt");
    ifstream movies("data/FilmFiltrati.txt");

    string s,t;
    const string space /* the final frontier */ = "\t";
    for (int i = 1; getline(actors,s); i++)
    {
        if (s.empty())
            continue;
        try {
            Actor TmpObj;
            int id = stoi(s.substr(0, s.find(space)));
            TmpObj.name = s.substr(s.find(space)+1);
            A[id] = TmpObj; // Python notation, works with C++17
            if (id > MAX_ACTOR_ID)
                MAX_ACTOR_ID = id;
        } catch (...) {
            cout << "Could not read the line " << i << " of Actors file" << endl;
        }
    }
```

```cpp


    for (int i = 1; getline(movies,t); i++)
    {
        if (t.empty())
            continue;

        try{
            Film TmpObj;
            int id = stoi(t.substr(0, t.find(space)));
            TmpObj.name = t.substr(t.find(space)+1);
            F[id] = TmpObj;
        } catch (...) {
            cout << "Could not read the line " << i << " of Film file" << endl;
        }
    }
}
```

Now we can use the file `Relazioni.txt`. As before, we loop on all the elements of this file, creating the variables

- `id_film`: index key of each movie
- `id_attore`: index key of each actor

If they both exists, we update the list of indices of movies that the actor/actresses played in. In the same way, we updated the list of indices of actors/actresses that played in the movies with that id.

```cpp
void BuildGraph()
{
    ifstream relations("data/Relazioni.txt");
    string s;
    const string space = "\t";

    for (int i=1; getline(relations,s); i++){
        if (s.empty())
            continue;
        try {
            int id_film = stoi(s.substr(0, s.find(space)));
            int id_attore = stoi(s.substr(s.find(space)+1));
            if (A.count(id_attore) && F.count(id_film)) { // Exclude movies and actors filtered
                A[id_attore].film_indices.push_back(id_film);
                F[id_film].actor_indicies.push_back(id_attore);
            }
        } catch (...) {
            cout << "Could not read the line " << i << " of Releations file" << endl;
        }
    }
}
```

Now that we have defined how to build this graph, we have to implement the algorithm what will return the top-k central elements.

The code can be found here: `https://github.com/lukefleed/imdb-graph`

5 Analysis of the results

In this section we are going to discuss the results of the top-k algorithm applied to the IMDb graphs. We are particularly interested in two factors:

- The time needed to for the execution in function of different filtering values.
- The discrepancy on the results while varying the filtering values

The first one will tell us how much more efficient the algorithm is in terms of time, independently from the results. The second one is the metric to understand how accurate the filtered algorithm is. It's clear that even if we can compute the algorithm 100 times faster, it's of no use if the results are completely different from the real ones.

The platform for the tests is *a laptop*, so can not be considered precise due factors as thermal throttling. The CPU is an Intel(R) Core™ i7-8750H (6 cores, 12 threads), equipped with 16GB of DDR4 @2666 MHz RAM.

5.1 Actors graph

Let's take into analysis the graph were each actors is a node and two nodes are linked the if they played in a movie together. In the case, during the filtering, we created the variable `MIN_ACTORS`. This variable is the minimun number of movies that an actor/actress has to have done to be considered in the computation.

Varying this variable obviously affects the algorithm, in different way. The higher this variable is, the less actors we are taking into consideration. So, with a smaller graph, we are expecting better results in terms of time execution. On the other hand, we also can expect to have less accurate results. What we are going to discuss is how much changing `MIN_ACTORS` affects this two factors

5.1.1 Time of execution

TO DO

5.1.2 Discrepancy of the results

We want to analyze how truthful our results are while varying `MIN_ACTORS`. The methodology is simple: for each results (lists) we take the intersection of the two. This will return the number of elements in common. Knowing the length of the lists, we can find the number of elements not in common.

A way to see this results is with a square matrix $n \times n$, $A = (a_{ij})$, where n is the number of different values that we gave to `MIN_ACTORS` during the testing. In this way the (i, j) position is the percentage of discrepancy between the results with `MIN_ACTORS` set as i and j

This analysis is implemented in python using the `pandas` and `numpy` libraries.

```
dfs = {
    i: pd.read_csv(f"top_actors_{i:02d}_c.txt", sep=’\t’, usecols=[1], names=["actor"])
    for i in [5] + list(range(10, 71, 10))}
sets = {i: set(df["actor"]) for i, df in dfs.items()}

diff = []
for i in sets.keys():
    diff.append([len(sets[i]) - len(sets[i] & sets[j]) for j in sets.keys()])
diff = np.array(diff, dtype=float)
diff /= len(next(iter(sets.values())))
```

Visualizing this analysis we obtain this

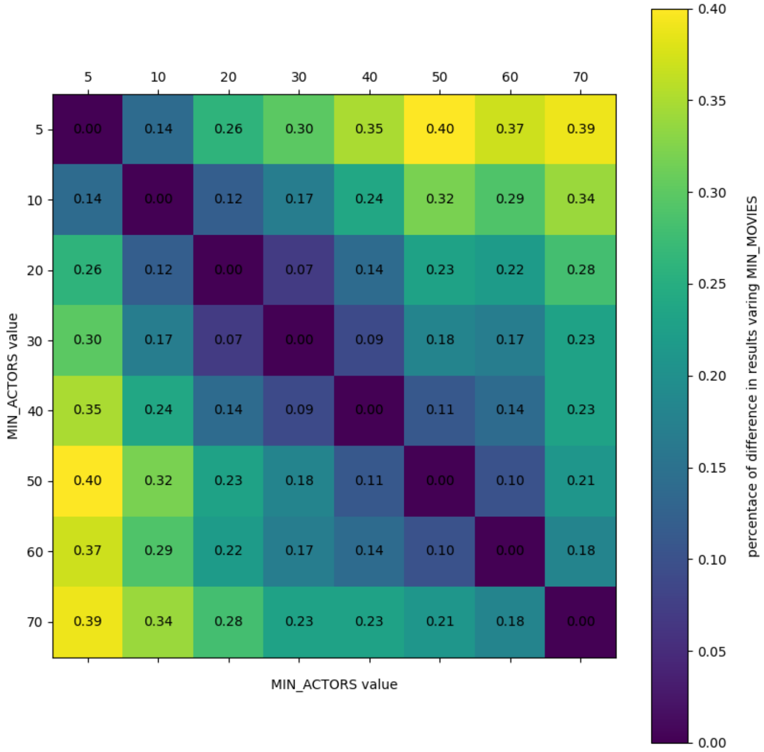

Figure 1: Discrepancy of the results on the actors graph in function of the minimum number of movies required to be considered as a node

As expected, the matrix is symmetrical and the elements on the diagonal are all equal to zero. We can see clearly that with a lower value of `MIN_ACTORS` the results are more precise. The discrepancy with `MIN_ACTORS=10` is 14% while being 39% when `MIN_ACTORS=70`.

This is what we obtain confronting the top-k results when $k = 100$. It's interesting to se how much the discrepancy change with different values of k. However, choosing a lower value for k would not be useful for this type of analysis. Since we are looking at the not common elements of two lists, with a small length, we would get results biased by statistical straggling.

DA FARE: TEST CON CON K=500 E K=1000

5.2 Movies Graphs

In this section we are taking into consideration the graph build over the movies and their common actors/actresses. Due to an elevated number of nodes, to optimize the performance during the execution in the section 3.2 we introduced the variable `VOTES`. It represents the minimum number of votes (indifferently is positive or negative) that a movie need to have on the IMDb database to be considered as a node in our graph.

As seen during the analysis of the actors graph in 5.1, varying this kind of variables affects the results in many ways. All the observations made before are still valid for this case, I won't repeat them for shortness. As done before (5.1.2), we are going to use a matrix to visualize and analyze the results

Giving us:

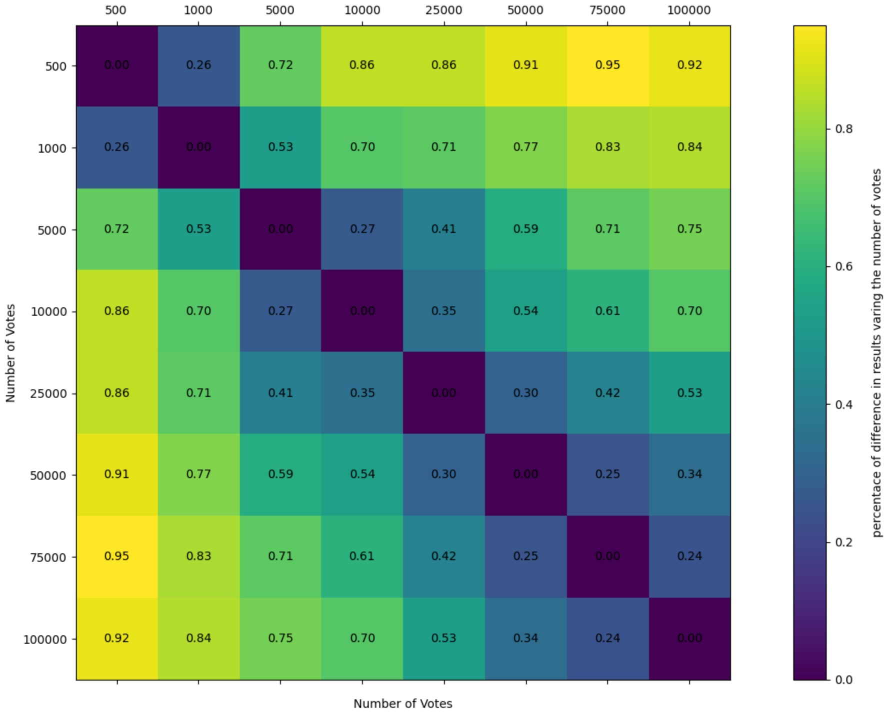

Figure 2: Discrepancy of the results on the movie graph in function of the minimum number of votes required to be considered as a node

```python
dfs = {
    i: pd.read_csv(f"top_movies_{i:02d}_c.txt", sep=’\t’, usecols=[1], names=["movie"])
    for i in [500, 1000, 5000, 10000, 25000, 50000, 75000, 100000]}
sets = {i: set(df["movie"]) for i, df in dfs.items()}

diff = []
for i in sets.keys():
    diff.append([len(sets[i]) - len(sets[i] & sets[j]) for j in sets.keys()])
diff = np.array(diff, dtype=float)
diff /= len(next(iter(sets.values())))
```

Dire qualcosa sull’analisi, ma andrebbe rifatta perché i valori non vanno bene

6 Visualization of the graphs

Graphs are fascinating structures, visualizing them can give us a more deep understanding of their proprieties. To do that we need to make some sacrifices. We are dealing with millions of nodes, displaying them all would be impossibile, especially on a web page as I did.

For each case we need to find a small (in the order of 1000) subset of nodes $S \subset V$ that we want to display. It's important to take into consideration, as far as we can, nodes that are "important" in the graph

All this section is implemented in python using the library `pyvis`. The goal of this library is to build a python based approach to constructing and visualizing network graphs in the same space. A pyvis network can be customized on a per node or per edge basis. Nodes can be given colors, sizes, labels, and other metadata. Each graph can be interacted with, allowing the dragging, hovering, and selection of nodes and edges. Each graph's layout algorithm can be tweaked as well to allow experimentation with rendering of larger graphs. It is designed as a wrapper around the popular Javascript `visJS` library

6.1 Actors Graph

For the actors graph we choose the subset S as the actors with at least 100 movies made in their carrier. We can immediately deduct that this subset will be characterized by actors and actresses of a certain age. But as we have seen, having an high number of movies made it's a good estimator for the closeness centrality. It's important to keep in mind that the graph will only show the relations nodes in this subset. This means that even if an actor has 100 movies made in his carrier, in this graph may have just a few relations. We can see this graph as collaboration network between the most popular actors and actresses.

An interactive version can be found at this web page. It will take a few seconds to render, it's better to use a computer and not a smartphone.

Interactive version: `https://lukefleed.xyz/imdb-graph.html`

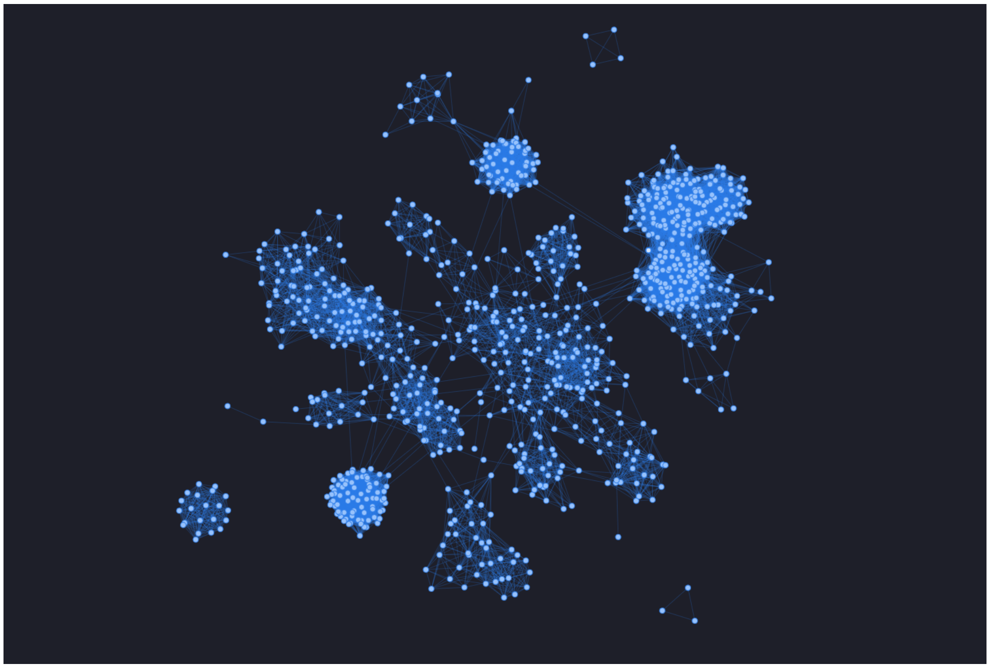

Figure 3: *The collaboration network of the actors and actresses with more that an 100 movies on the IMDb network*

The results obtained is extremely interesting. We can clearly see how this graph it's characterized by different (and some times isolated) communities. The nodes in them are all actors and actresses of the same nationality. There are some very big clusters as the *Bollywood*'s one that are almost isolated. Due to cultural and linguistic differences those actors never collaborated with anyone outside their country.

A visual analysis of this graph can reflects some of the proprieties that we saw during the analysis of the results. Let's take the biggest cluster, the Bollywood one. Even if it's very dense and the nodes have a lot of links, none of them ever appeared in out top-k results during the testing. This happens due to the proprieties of closeness centrality, the one that we are taking into consideration. It can be seen as the ability of a node to transport information efficiently into the graph. But the Bollywood's nodes are efficient in transporting information only in their communities.

A simple and heuristic way to see this phenomena is by grabbing in the interactive graph a node with an higher centrality and dragging him around. We'll see that it will drag with him every community. If we repeat the same action with a Bollywood node, it will only move the nodes of his community, leaving almost un-moved all the other nodes