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:

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

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;
        }
    }
```

```
23
24
25      for (int i = 1; getline(movies,t); i++)
26      {
27          if (t.empty())
28              continue;
29
30          try{
31              Film TmpObj;
32              int id = stoi(t.substr(0, t.find(space)));
33              TmpObj.name = t.substr(t.find(space)+1);
34              F[id] = TmpObj;
35          } catch (...) {
36              cout << "Could not read the line " << i << " of Film
    file" << endl;
37          }
38      }
39  }
```

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.

```
1  void BuildGraph()
2  {
3      ifstream relations("data/Relazioni.txt");
4      string s;
5      const string space = "\t";
6
7      for (int i=1; getline(relations,s); i++){
8          if (s.empty())
9              continue;
10         try {
11             int id_film = stoi(s.substr(0, s.find(space)));
12             int id_attore = stoi(s.substr(s.find(space)+1));
13             if (A.count(id_attore) && F.count(id_film)) { //
    Exclude movies and actors filtered
14                 A[id_attore].film_indices.push_back(id_film);
15                 F[id_film].actor_indicies.push_back(id_attore);
16             }
17         } catch (...) {
18             cout << "Could not read the line " << i << " of
    Releations file" << endl;
19         }
20     }
21 }
```

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`