Università degli Studi di Torino

Corso di Laurea in Fisica dei Sistemi Complessi

Tesi di Laurea Magistrale

Polls in Multilayer Networks

Candidato:
Francesco Vincenzo Surano

Relatore:
Prof. Pietro Terna

Controrelatore:
Prof. Michele Caselle

Sessione di Laurea 2017
Anno Accademico 2016/2017

Like the legend of the phoenix
All ends with beginnings

Contents

Chapter 1

Introduction

This work originates from the curiosity on how polls, particularly in politics, have failed in some major occasions in the last year, e.g. the election of the U.S. president or the "Brexit" referendum; here we try to assess a new interpretation based on how we extract information to research the structure of our society.

In chapter 2 we introduce some basic concepts of multilayer networks with definitions and measures, as these new approaches are of great interest in our work.

Subsequently, in chapter 3, we introduce polls, their basic implementation and several problems, as they are currently one of the main instruments to explore society and predict people's thoughts and behaviors.

In chapter 4 we define Agent-Based Models as our paradigm in simulation; this approach permits the emergence of properties rather than the imposition of them, resulting in an added value for the knowledge of processes.

After the preliminary part, in chapter 5 we first investigate algorithms capable of generating a multilayer network and verify that the obtained network retains the characteristic of a social network.

Then we implement, in following sections, the diffusion process based on the Bounded Confidence Model.

Lastly, voting procedures are implemented and studied, obtaining strong differences in measures if considering layers of the network or the projection of the network.

Finally, simple polls are performed and discussed on a multilayer network.

The multi-layer approach, as discussed in chapter 6, expresses hence interesting properties that can be related to polling and electoral results in real-life scenarios.

In the appendix, chapter 7, original coding tools may be found for a more in-depth look.

Chapter 2

The Network

2.1 Introduction to Multilayer

In this chapter we will see a basic introduction of multilayer network as reorganized by Boccaletti et al. [2014].

2.1.1 Formal definition

A multilayer is a pair $\mathcal{M} = (\mathcal{G}, \mathcal{C})$ where $\mathcal{G} = \{G_\alpha; \alpha \in \{1, ..., M\}\}$ is a family of (directed or undirected, weighted or unweighted) graph $G_\alpha = (X_\alpha, E_\alpha$ (called layers of $\mathcal{M}$) and $\mathcal{C} = \{E_{\alpha\beta} \subseteq X_\alpha \times X_\beta; \alpha, \beta \in \{1, ..., M\}, \alpha \neq \beta\}$ is the set of interconnection between nodes of different layers G_α and G_β with $\alpha \neq \beta$. The elements of $\mathcal{C}$ are called crossed layers, and the elements of each E_α are called *intralayer* connections of $\mathcal{M}$ in contrast with the elements of each $E_{\alpha\beta}$ $(\alpha \neq \beta)$ that are called *interlayer* connections.

2.1.2 Multiplex

A multiplex network is a particular type of multilayer network in wich $X_1 = X_2 = ... = X_M = X$ and $E_{\alpha\beta} = \{(x, x); x \in X\}$, i.e., the only possible types of interlayer connections are those in which a given node is only connected to its counterpart nodes in the rest of layers. In other words, multiplex networks consist of a fixed set of nodes connected by different types of links.

2.1.3 Projection network

Given a multilayer $\mathcal{M} = (\mathcal{G}, \mathcal{C})$ the projection network is a single layer network such that: $\mathcal{G} = proj(\mathcal{M}) \mid \mathcal{G} = \{\bigcup_{\alpha=1}^{M} X_\alpha, \bigcup_{\alpha=1}^{M} E_\alpha\}$.

2.2 Measures on Multilayer

In order to properly capture the structure of multilayer networks, we need a new set of measures to quantify various properties of nodes, edges and more complex structures. At first we can try to extend well-known measures, such centrality or degree, from their formal definition on single-layer network, in some situation we will have to develop a new set of measures specifically for multilayer network.

2.2.1 Degree

In its natural extension, proposed by Battiston et al. [2014], the degree k_i of a node i in a single-layer is a *vector* $\mathbf{k}_i = (k_i^1, ..., k_i^\alpha, ..., k_i^M)$ where $k_i^{(\alpha)}$ is the degree of node i in layer α. Since the new definition of degree is a vector, we don't have a clear ordering in $\mathbb{R}^M$, we can alternatively try to use a derived measure such the *overlapping degree* defined as $o_i = \sum_{\alpha=1}^{M} k_i^\alpha$.

2.2.2 Centrality

Centrality measures, such as closeness and betweenness centrality, are based on the metric structure of the network. These measures can be easily extended to multilayer networks once the metric and geodesic structure are defined. Employing *eigenvector centrality*, based on the spectral properties of the adjacency matrix, considers not only the number of link but also the quality of such connection. This measure can be extended to multilayer network in a number of ways, from simply combining the eigenvector centrality of each layer to obtain a $\mathbb{R}^{N\times M}$ column-stochastic matrix, to a more refined measure involving a *influence matrix* W to measure the influence of each layer in the

calculation of a *heterogeneous eingenvector-like centrality*. Further can be read in Solá et al. [2013] .

2.2.3 Clustering

In order to give a definition of the clustering coefficient of a node in a multilayer network $\mathcal{M} = (X, E, \mathcal{S})$ where $\mathcal{S} = \{S_1, ..., S_M\}$ and $S_\alpha = (X_\alpha, E_\alpha)$, $\alpha | 1 \leq \alpha \leq M$, we need to introduce some notations. For every node $i \in X$ let $\mathcal{N}(i)$ be the set of all neighbours of the node i in $G = proj(M)$, then we define $\mathcal{N}_\alpha(i) = \mathcal{N}(i) \cap X_\alpha$ and $\bar{E}_\alpha(i) = \{\{k, j\} \in E_\alpha | k, j \in \mathcal{N}_\alpha(i)\}$ such that the subgraph of the layer S_α generated by $\mathcal{N}_\alpha(i)$ will be the graph $\bar{G}_\alpha(i) = (\mathcal{N}_\alpha(i), \bar{E}_\alpha(i))$. Note that the largest possible number of link between the nodes of $\mathcal{N}_\alpha(i)$ is such that $|\bar{E}_\alpha(i)| \leq \frac{1}{2} |\mathcal{N}_\alpha(i)|(|\mathcal{N}_\alpha(i)| - 1)$. The clustering coefficient of a given node i in a multilayer network $\mathcal{M}$ is defined as

$$\mathbf{C}_M(i) = \frac{2 \sum_{\alpha=1}^{M} |\bar{E}_\alpha(i)|}{\sum_{\alpha=1}^{M} |\mathcal{N}_\alpha(i)|(|\mathcal{N}_\alpha(i)| - 1)}$$

and the clustering coefficient of $\mathcal{M}$ is the average of all $\mathbf{C}_M(i)$ over the set of nodes, as described by Criado et al. [2012] .

2.2.4 Distance

Given a multilayer network $\mathcal{M} = (\mathcal{G}, \mathcal{C})$ we consider the set

$E(\mathcal{M}) = \{E_1, ..., E_M\} \bigcup \mathcal{C}$. A *walk* in $\mathcal{M}$ (of length *q-1*)is a non-empty alternating sequence $\{x_1^{\alpha_2}, l_1, x_2^{\alpha_2}, l_2, ..., l_{q-1}, x_q^{\alpha_q}\}$ of nodes and edges with $\alpha_2 \in \{1, ..., M\}$ such that for all $r < q$ $\exists \mathcal{E} \in E_\mathcal{M}$ with $l_r = (x_r^{\alpha_r}, x_{r+1}^{\alpha_{r+1}}) \in \mathcal{E}$. If each node is visited only once the *walk* is called *path*. Of course, in a multilayer network there are at least two types of edges, namely intralayer and interlayer edges. Thus, this definition changes depending on whether we consider interlayer and intralayer edges to be equivalent. If it is possible to find a path between any pair of its nodes, a multilayer network M is referred to as connected; otherwise it is called disconnected. Further details may be found in Costa et al. [2007]

2.2.5 Correlation

Besides having to extend classical measures from single-layer to multilayer networks, in the latter a new, specific, kind of measure emerges. Various kind of correlation among different layers needs to be investigated in order to better understand properties of multilayer network.

- The simplest way to analyze the degree correlation of the same node in different layer it is by constructing the matrix $P(k^\alpha, k^\beta) = \frac{N(k^\alpha, k^\beta)}{N}$ where $N = |\bigcup_{\alpha=1}^{M} X_\alpha|$ and $N(k^\alpha, k^\beta)$ is the number of nodes with degree k^α in layer α and degree k^β in layer β.

- We can also calculate the avarage degree, conditioned, as $\bar{k}^\alpha(k^\beta) = \frac{\sum_{k^\alpha} k^\alpha P(k^\alpha, k^\beta)}{\sum_{k^\alpha} P(k^\alpha, k^\beta)}$

- And the more familiar Pearson coefficient as $r_{\alpha\beta} = \frac{\langle k_i^\alpha k_i^\beta \rangle - \langle k_i^\alpha \rangle \langle k_i^\beta \rangle}{\sigma_\alpha \sigma_\beta}$ where σ_α is the standard deviation of k_i^α.

- The total overlap of links, the fact that two nodes are linked both in layer α and in layer β, can be measured with $O^{\alpha\beta} = \sum_{i<j} a_{ij}^\alpha a_{ij}^\beta$, where a_{ij}^α is 1 (or a weighted value) if exist an edge from node i to node j in layer α.

2.2.6 Other measures

Many other useful measures can be defined for multilayer network, for exaple all the natural extension of weighted edges, and useful representation can be made using vector, matrix and tensor. Some useful measure are

- Multilink, defined as a vector $a_{ij} = (a_{ij}^1, ..., a_{ij}^\alpha, ..., a_{ij}^M)$ where m_{ij}^α is 1 (or a weighted value) if node i and node j are linked in layer α.

- The activity, as proposed by Nicosia and Latora [2015], is a useful measure to know if a node i is active in a layer α, for multiplex network we

have $b_{i,\alpha} = 1 - \delta_{0,k_i^\alpha}$ and $B_i = \sum_{\alpha=1}^{M} b_{i,\alpha}$ as the measure of activity of node i.

2.3 Interpretation of layers

What is the use of multilayer network? The most intuitive one is the description of a network evolving in time, realizing each time step with a different layer, a photograph of the network in which nodes and edges are constantly changing.

But multilayer network can also be used to explore the society in a different way. Multiplex networks are the weapon of choice to describe complex social interactions that happen with different means of communication. Assigning to each layer a different instrument of communication we can explore how the information (or really anything else that can be spread among humans) is diffused among the population, noticing interesting emerging phenomena.

For example Lewis et al. [2008], although not deeply using the instrument of multilayer anlysis, shows how it's possible to increase the knowledge of social network and perform a deeper sociological study using a multi-layer approach and combining information retrieved with different methods, in the case through social network and official college dataset.

Beside social analysis the framework of multilayer network can be suited to analyze modern financial market, as in Bargigli et al. [2015] where the italian interbank market is analyzed as a multiplex, in which each node is a bank and each layer a mean of interaction, for example the exchange of certain financial products or loans.

In this work we will study the diffusion of political beliefs in a society that use more than one way to communicate, we will build a multiplex network where each node is a person, an Agent, and each layer is a communication system, for example social networks, telephones, etc. In this way we will have, for each layer, the network induced by the system associated and we will be able to explore how political beliefs spread in the society, we will be also able to retrieve the role played by each layer in the diffusion and the correlation

that active users in one layer have with the rest of society.

Chapter 3

Polls

A poll is an investigation, in order to verify the existence, the diffusion or the dimension of a social phenomenon, thorough the use of a series of questions posed to a number of people of a given population.

As explained by P. Natale in some of his books (*Attenti al sondaggio* Natale [2009] , *Il sondaggio* Natale [2004]) polls are among the most widely used instrument in social science to verify the adherence of a model with the reality and to extrapolate characteristics of the population to use in research projects.

Polls suffer a lot of problems, we will discuss those we can directly analyze and reproduce in a simulation, avoiding more field-specific issues such as the *Hawthorne effect*, describing the influence of the interviewer on the interviewed, or the problem of interpretation of the questions posed, being them seen as sensible data or not, and so on.

In recent years polls have become multi-platform: thanks to the advent of the telephone first and the internet later, evolving form simple face-to-face interviews to more complex protocols.

3.1 Methods

3.1.1 CAPI

The CAPI (*Computer Assisted Personal Interview*) is a survey method based on face-to-face interview. The interviewer may choose a crowded place and

start interviewing people that consent, or physically go to randomly chosen house addresses and submitting the survey to who lives there, registering the answer in a portable electronic device. If the interviewed does not consent or is not at home, the interviewer move to the next subject she can physically find. The choice of the place or the time of the survey is critical to intercept a significative sample.

3.1.2 CATI

The CATI (*Computer Assisted Telephone Interview*) is a survey method based on the use of the telephone. Random numbers, from public or private databases, are extracted and chosen for the survey. The interviewer dials one of the numbers and, if who answer agrees to participate to the survey, pose her a number of questions, registering the answer in the computer. As of today, in Italy, is estimated that 20% of the population does not have a phone at home, using only mobile phones or taking advantages of the Italian law registering to not be contacted for surveys. If the target number of interviews is not reached, due to refuse to participate or the impossibility to contact the chosen number, a new set of random number is extracted that the interviewer will try to contact.

3.1.3 CAWI

The CAWI (*Computer Assisted Web Interview*) is a survey method based on the use of internet. Internet is increasingly widespread among the population, and increasingly bigger databases of voluntarily disclosed e-mail are at disposal of the researchers to use. Random e-mail addresses are extracted from the databases and a survey is sent to them. The receiver can at any time fill and submit the survey. If, after a chosen time, the number of responses is too low, a new set of e-mail addresses is extracted and the survey sent to those new addresses.

3.2 Self-Selection

It is trivial to infer that in any given survey there is the obvious choice to not answer. In the survey methods we have just seen there is also the problem of the inability to contact the chosen subject. Willing to reach a certain size of the sample, the only viable method is to choose a new set of peoples to integrate the existing sample. But we do not have any evidence that the substitute would give the same answer as the first chosen subject or even that they are similar in their ideas and conditions. Our sample has an immediate bias towards subjects that do answer the survey and those who do not. Furthermore the access to the technology used to submit the survey is not homogeneous among the population, creating another bias which needs to be compensated. In CAPI and CATI interviews, one must account for the time at which the survey is submitted and how the interviewer is trained in order to furthermore reduce the bias towards whose that have other commitments or may misunderstand the questions.

3.3 Social Desirability

Social Desirability is a widely known type of response bias that is the tendency of survey respondents to answer questions in a manner that will be viewed favorably by others. This may also include the idea they have of society, their friends or ideas spread by media. This effect may produce different behaviors of the respondent, either refusing to answer a question that may expose her idea as different from the one socially accepted or, worse from the perspective of the interviewer, lying expressing a different opinion than the one she really has. This effect may show up also in real political election creating a *last minute swing* up to 3%-4%. In politics the *Social Desirability* may be influenced by the judgment on government acts, the diffusion of survey's results about voting intentions and results about minor election in municipality (or similar).

3.4 Different Approach

Seen the number of problems the researcher may find in the pursue of a good poll, a series of techniques have emerged in order to try solve some of those, mainly the first of all problems: reducing the cost. Besides all the non-quantitative methods, such as focus-group or deep interviews, notable are the *pondering* and *preconsituted panel*.

Pondering is a series of statistical techniques that, known the distribution of certain characteristics of the whole population (such as sex, age..), tries to align the results of the survey, increasing or decreasing the importance of the answers given by the interviewed subject, ideally obtaining a good representation of the population also if the sample was poorly withdraw.

Preconstituted panel is a set of subjects that voluntarily agree to be periodically interviewed for a certain time on a specific argument. The idea is that, if the panel is sufficiently heterogeneous ad representative of the population, the opinion evolution in the panel will be the same of the population.

Chapter 4

Agent Based Model

One of the main objective of this work is to tackle the study of multilayer network with the use of Agent-Based Models. We think that this approach may be of extreme interest as it mimics the formation of social networks in reality and, as far as we know, no such approach has been tried to create and evolve multilayer social networks so far.

4.1 Introduction to ABM

The history of Agent-Based Model can be traced back to 1940s and the Von Neumann cellular automata, a theoretical machine capable of self-reproduction following simple rules. In the 1970s and 1980s we see the birth of Thomas Shelling's segregation model, Schelling [1969], and Robert Axelrod's tournament of Prisoner's Dilemma algorithms. By the 1990s, with the creation of NetLogo and Swarm software, the Agent-Based approach became widespread in social science and study of complex system.

An Agent-Based Model (ABM) is one of a class of computational models based on the Agent-Oriented Programming (AOP), which simulates the actions and interactions of autonomous Agents with a view to assessing their effects on the system as a whole. In other words the ABMs aim is to re-create and predict the appearance of complex phenomena simulating operations and interactions of multiple Agents, only defining them and not the phenomena; the analogy

to statistical mechanics is to define the system through its micro-scale details and then study the complex macrostate.

Combining elements of complex systems, computational sociology, multi-Agent systems and Monte Carlo methods (to introduce randomness) the complex behavior emerges from the lower (micro) level of systems, i.e. simple behavioral rules, to a higher (macro) level. This principle is extensively adopted in the modeling community. Individual Agents are typically characterized as boundedly rational, i.e. they act pursuing their own interests, but from the point of view of an external observer they express heuristic decision-making rules, learning and adaptation that can be synthesized as: "the whole is greater than the sum of the parts", in the sense that we get more information than those we put in the system.

From a theoretical point of view there are three central concepts that distinguish ABM from other computational model.

- **Emergence** The Agent-Based Model explore the complex system equilibrium deriving from simple rules of the Agents instead of studying the system's equilibrium.

- **Agent-Object** Agent-Based Models consist of dynamically interacting rule-based Agents which can create a real-world-like complexity. Agents are typically situated in space and time, and their location and behaviors are encoded in algorithmic form in computer programs. In some cases, though not always, the Agents may be considered as intelligent and purposeful.

- **Complexity** ABMs complement traditional analytic methods in the sense that the last characterize the equilibria of a system, while the first allow the possibility of generating those equilibria. Agent-Based Models can explain the emergence of higher-order patterns and non-linear systems for which we do not have an analytic solution or the numerical one is too expensive from a computational point of view.

In our work the Agent-Based approach will be central as we want to limit external rules in the creation and growth of the network, those processes will

be carried out by Agents, acting as nodes. This approach is extremely useful because, only setting the behavior of the Agents, we try to mimic the real life interaction that model and create the social structures we all live in, developing a tool that can be reversed to deeper study the behavior of people in a complex context.

Chapter 5

The Model

In our project we aim at exploring the agreement between polls and diffusion of ideas in a social network. How can we approach such problem?

At first we will have to create a social network, this being itself a problem that will be addressed in further sections. Our approach will be Agent-Based, so the network will be self-creating based on rules each Agent apply, this is the *ABM* approach to simulation. As explained in previous sections, multiplex are being widely used to address social studies, so we will use such representation in our simulation. Not many tools are available to rapidly implement multilayer and multiplex, so many of the algorithms used in this thesis will be created by the author for such goal.

In our multilayer, always following the *ABM* approach, Agents will diffuse ideas among themselves. After the diffusion of ideas we will have to perform polls, in fact asking Agents to report a value, e.g. a diffused ideas or a vote, in order to create a sample to be compared to the total distribution of such ideas or vote in the whole network. Studies on rumors spreading are widespread but none, as far as we know, grasp the tool of poll as a practical instrument used in social sciences, with its limitations and peculiarities.

Being ours an Agent-Based multiplex network, we should expect, rather than impose, emerging properties both in the self-generating algorithm and in the diffusion of ideas. To see real-life problems we will be introducing polls, according with previous discussion, and typical problems of polls, e.g. *social*

desirability.

Each Agent posses a set of *beliefs*, either unchangeable (fixed) or changeable (variable) by the environment or other Agents. This is done in order to simulate all set of properties of an individual, fixed beliefs may represent unchangeable features of a humans beings, e.g. physical or deep moral beliefs, variable beliefs may represent all the beliefs, ideas and interpretations of reality that we change during the course of our life, either by confronting our friends or by combining our knowledge.

5.1 Generating the Network

The set of detailed functions used to generate the multiplex network may be found in *GenerativeTool.py*, furthermore described in sec. 7.3. Our network will be self generating: each Agent at a given time-step will add new friends depending on the rules we have chosen.

Multiplex representation, as more and more being seen as correct for social interactions, is useful as each layer of our network can be interpreted in different ways, for our perspective may be interesting considering them as point of access that researchers have in the real network, like means of communication, e.g. the telephone, social networks, or social contexts in which the individual is found discussing and changing her ideas, e.g. Universities, sports Clubs.

In our simulation each Agent will attach to other Agents, creating a network, based on the layer she is into, the "rules" of the layer and the beliefs, either variable or fixed, she, and other Agents, posses.

All our algorithms are based on the comparison of beliefs possessed by Agents, in fact is the Agent that ask other Agents to form a link after performing a series of comparisons with her own belief and those of other Agents.

This is done in order to mimic the human approach of searching new friends based on common interests, on the computational side this include a lot of comparisons between arrays and the right choice of how to do that is not trivial. For example the choice of what kind of normalization to use in such arrays may vary the result of operations. Beside everything what is most important is to keep coherence trough all the experiments to avoid misinterpretation of results.

5.1.1 First Implementation

Random Algorithm

At first we have tried the very simple algorithm *AttachForBelief* in sec. 7.3.1, it asks a random Agent of the network a certain number of randomly chosen fixed beliefs and compares them with those of the chosen Agent, if the distance of the beliefs is acceptable a link between the two Agents is created in a layer randomly changed for each Agent, the process is repeated until either we run out of Agents or we reach the desired number of friends.

All Agents are created, then algorithm 7.3.1 is run for each Agent. We explored a parameter space evaluating three main characteristics of the generated network: degree assortativity, clustering coefficient and average degree.

The variable parameters chosen are:

- number of Agents created in the network
- number of desired friends for each Agent, consider that not all Agents reach the desired number of friends and some exceed that
- number of layers that can be accessed
- number of beliefs to be simultaneously close enough in order to create a link between two Agents
- the acceptable distance for two beliefs to be considered close one another
- total number of beliefs possessed by an Agent

Since layers are randomly chosen, the result is averaged over all layers in order to have a simpler visualization of results.

In these results, figure 5.1, 5.2 and 5.3, we can see that the main measures explored are not indicative of a social network: degree assortativity is negative, the clustering coefficient is only slightly positive.

Furthermore we considered the projection network of the generated multiplex to search for emerging properties looking at degree assortativity and average degree, shown in 5.4 and 5.5. The projection network is analyzed as

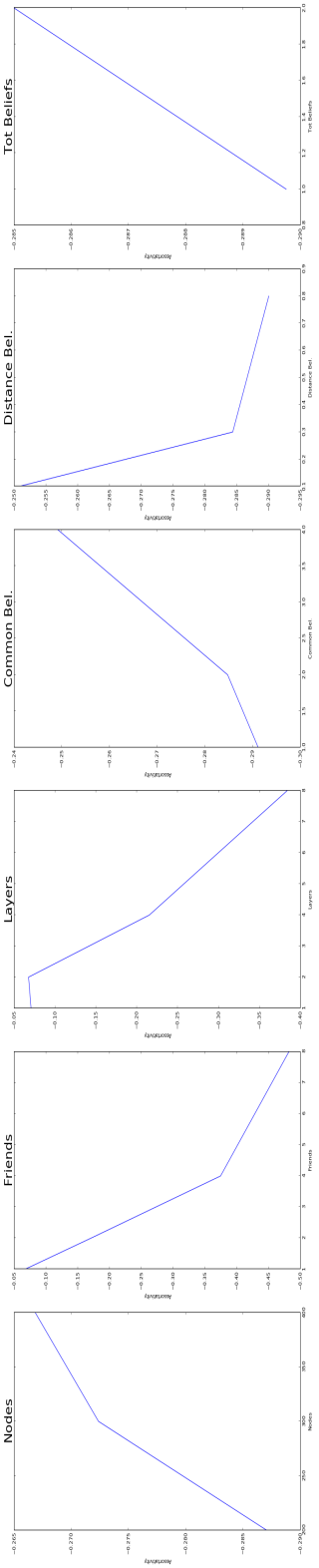


Figure 5.1: Degree assortativity related to changing parameters

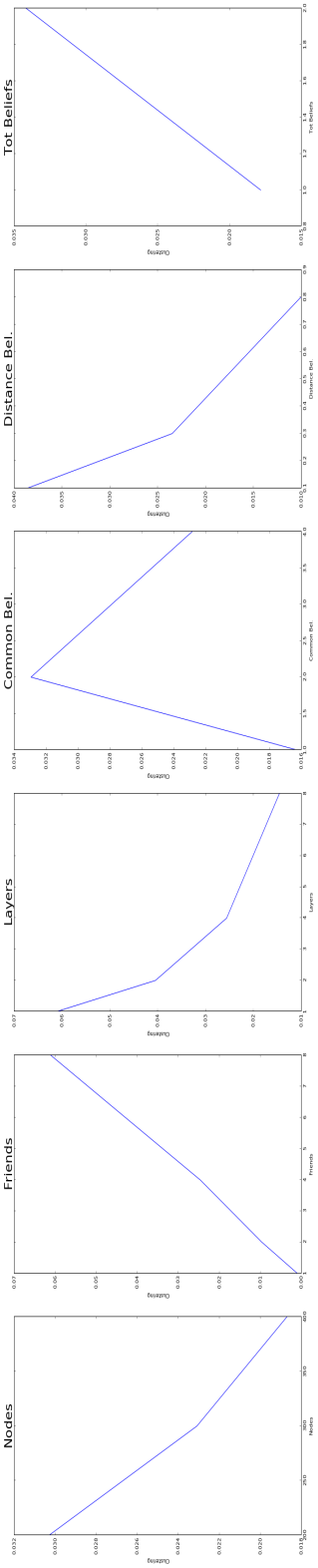


Figure 5.2: Clustering Coefficient related to changing parameters

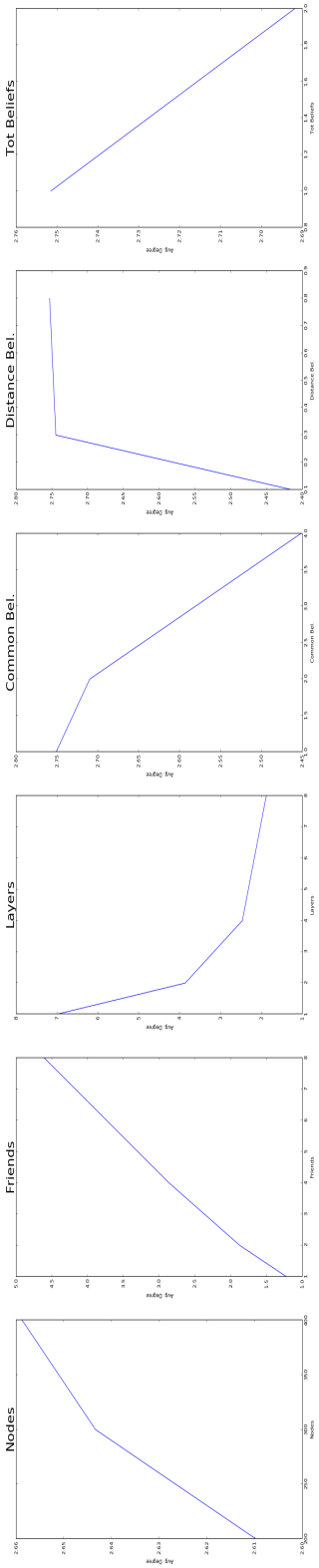


Figure 5.3: Average degree related to changing parameters

it can represent the overall social network for an Agent emerging from the combination of all the layers of interaction.

From the above analysis we can infer how to set our parameters to tune the properties of our network. It is to be noted that degree assortativity has a negative value for the parameter space analyzed. It is widely known that social networks have positive values for degree assortativity, although some parameters may be tuned to increase its value, a radical change in the generating algorithm is needed in order to make our Agents create a network that can be assumed similar to a social network.

Layer-Wise Beliefs

The algorithm *AttachChosenBelief* in sec. 7.3.2 differs from algorithm 7.3.1 for the fact that the set of fixed beliefs to compare between two Agents is not randomly chosen but is related to the layer chosen for the interaction between Agents, breaking the symmetry between the layers and making the creation of a link in a layer easier (or harder) than in another. For each layer, that is randomly chosen at each interaction, a set of beliefs randomly set at beginning to check is associated, this in order to create a more lifelike situation: in different context an Agent find itself in a layer where different set of interests or topics of interaction are dominant.

At first all Agents are created, then algorithm 7.3.2 is run for each Agent. We explored a parameter space evaluating three main characteristics of the generated network: degree assortativity, clustering coefficient and average degree.

The variable parameters chosen are:

- number of Agents created in the network
- number of desired friends for each Agent, consider that not all Agents reach the desired number of friends and some exceed that
- number of layers that can be accessed
- the acceptable distance for two beliefs to be considered close one another

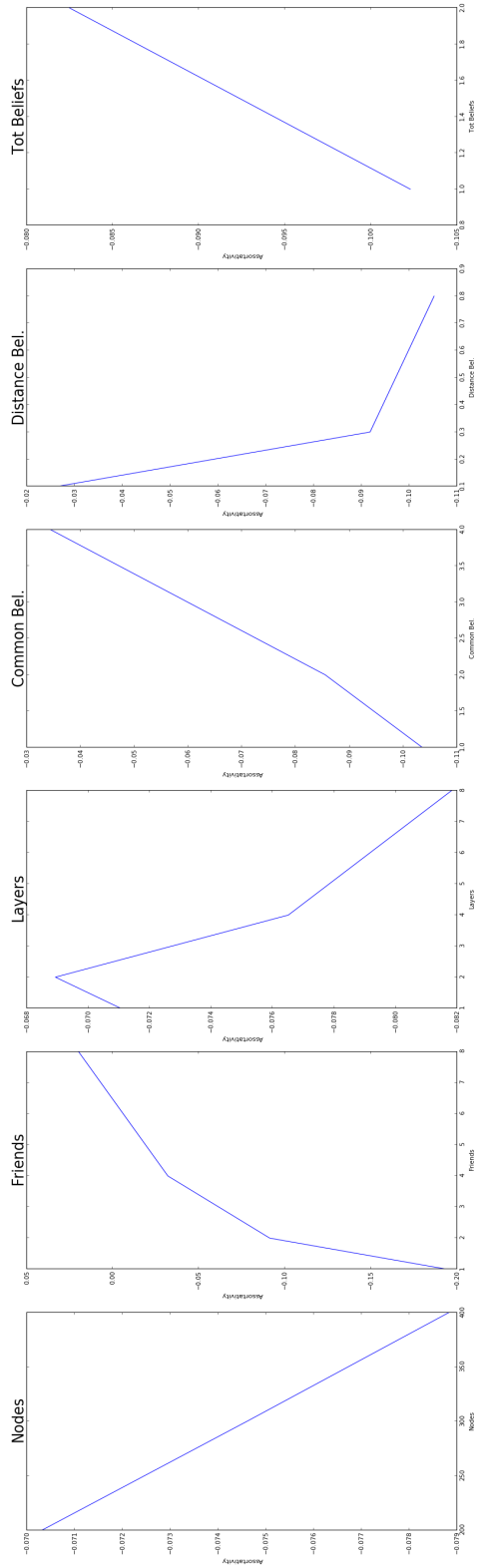


Figure 5.4: Degree assortativity related to changing parameters

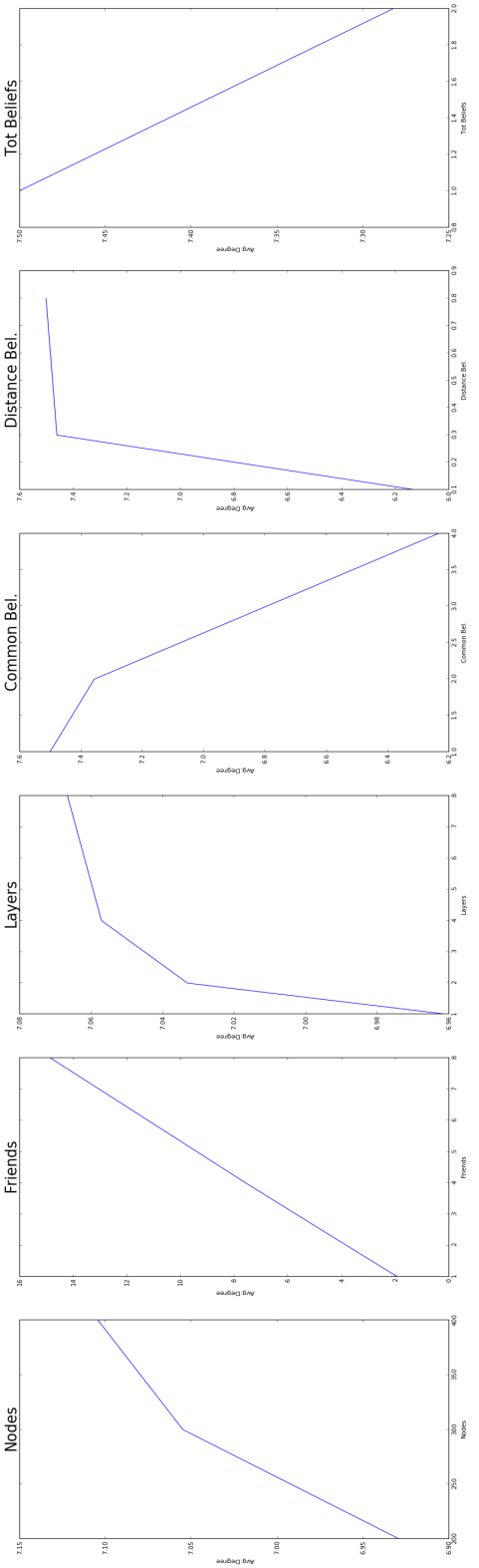


Figure 5.5: Average degree related to changing parameters

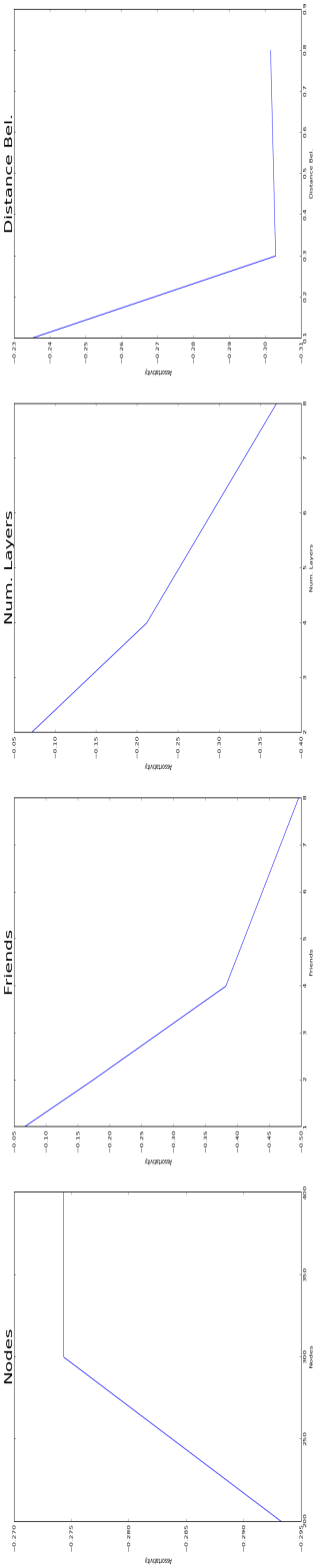


Figure 5.6: Degree assortativity related to changing parameters

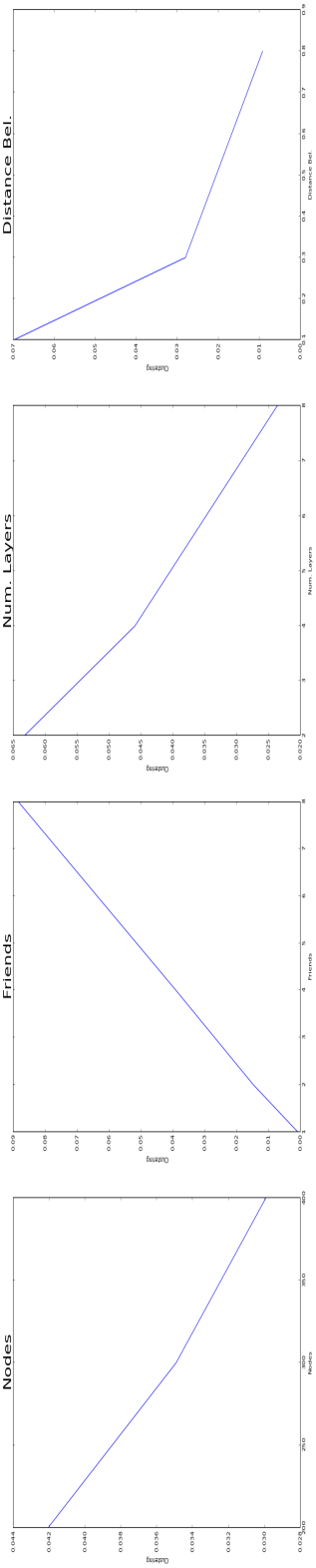


Figure 5.7: Clustering Coefficient related to changing parameters

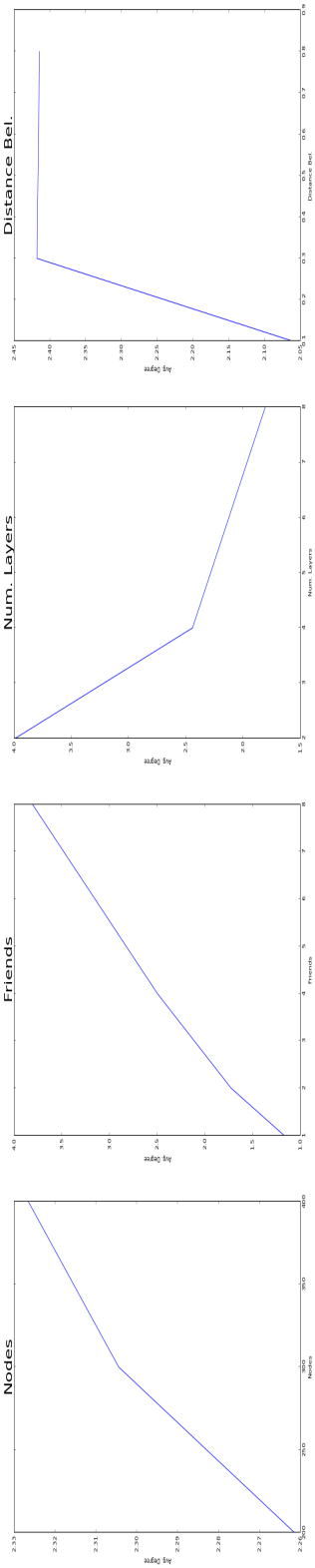


Figure 5.8: Average degree related to changing parameters

Since layers are randomly chosen, the result is averaged over all layers.

In the results show in 5.6, 5.7 and 5.8, we can still notice that degree assortativity is negative in value for all parameters and the clustering coefficient is very small.

Furthermore we considered the projection network of the generated multiplex, holding the same interpretation as before, to search for emerging properties looking at degree assortativity and average degree as shown in 5.9 and 5.10.

Besides the intrinsic difference between algorithm 7.3.2 and 7.3.1, being the second one properly designed to take advantage of the multilayer structure of our generated multiplex, we see that they held similar measurements, degree assortativity still has a negative value and for that reason we must exclude this generated network to be considered a social network.

Friend First

The last generating algorithm that will be tried, *AttachFriendFirst* in sec. 7.3.3, is based on algorithm 7.3.2 but with a major difference: each Agent, while searching for a candidate to form a link with, will at first try to bond with the friends of her friends. This is done to mimic real life situation in which a person is introduced to new possible acquaintances by her friends, thus expanding her social network trough existing link of her first neighbors. If the desired number of friends is not found in this subset of the network, the research will be expanded randomly to all others Agents of the network, as done by algorithm 7.3.2.

At first all Agents are created, then algorithm 7.3.3 is run for each Agent. We explored a parameter space evaluating three main characteristics of the generated network: degree assortativity, clustering coefficient and average degree.

The variable parameters chosen are:

- number of Agents created in the network
- number of desired friends for each Agent, consider that not all Agents reach the desired number of friends and some exceed that

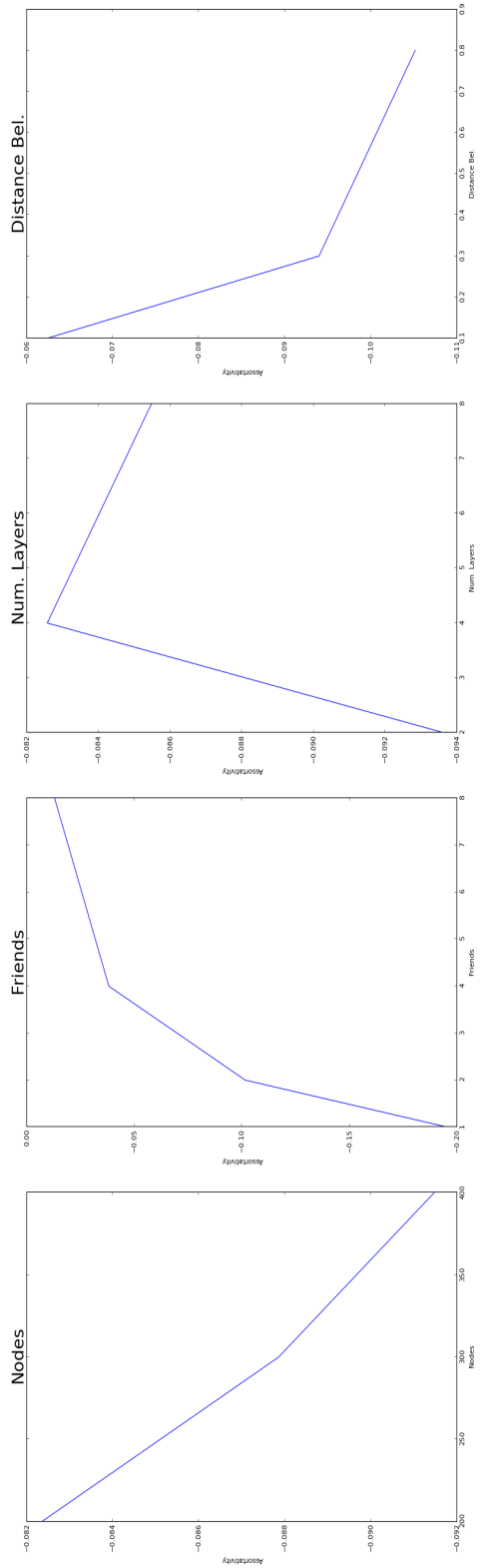


Figure 5.9: Degree assortativity related to changing parameters

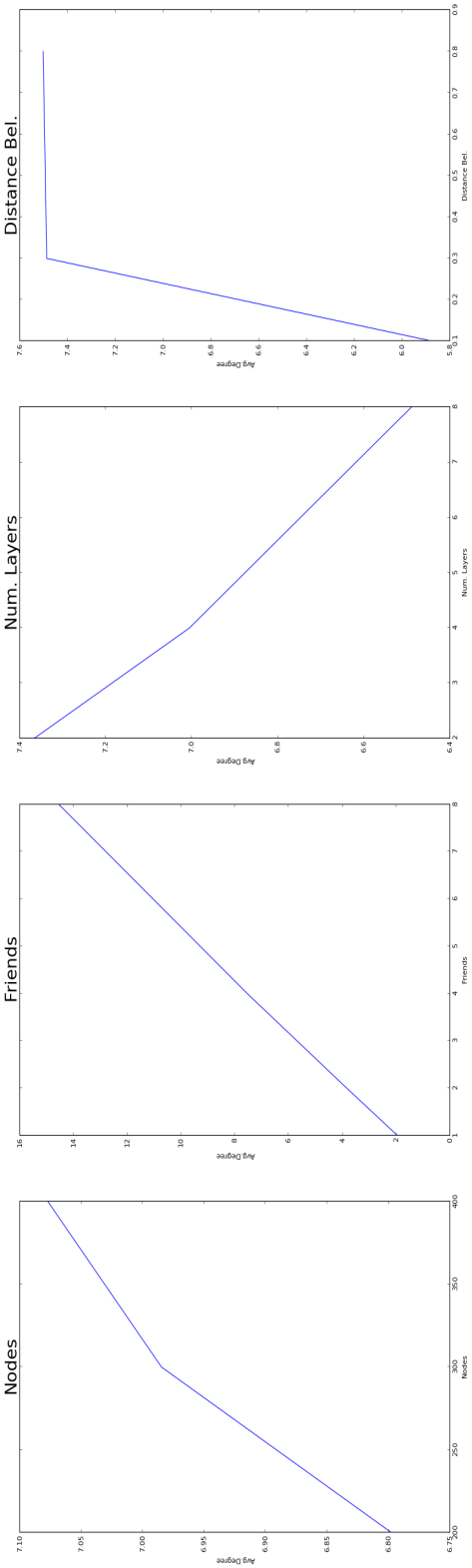


Figure 5.10: Average degree related to changing parameters

- number of layers that can be accessed
- the acceptable distance for two beliefs to be considered close one another

Since layers are randomly chosen, the result is averaged over all layers.

Again we see in 5.11, 5.12 and 5.13, that degree assortativity is negative and clustering is small, these being the first measures to asses if we have created a social network, the result is not promising.

Furthermore we considered the projection network of the generated multiplex, holding the same interpretation as before, to search for emerging properties looking at degree assortativity and average degree, shown in 5.14 and 5.15.

Also in this last algorithm we found the same problem as before: being widely known that positive degree assortativity is a key property of social networks, we could not find a wide enough region of the parameters space in which this property emerges from our algorithms.

A rethinking of the generating algorithm is needed at this point.

5.1.2 Reconsidering the Implementation

The first implementation showed us that our self-generated network could not be considered a social network, mainly because it lacked some specific property, namely a positive degree assortativity.

In the second implementation we reconsidered just one step: instead of creating all Agents and then run the attachment algorithm for all Agents, we run the attachment algorithm for a specified Agent right after her creation. The implementation now is that an Agent is created, the algorithm is run for that Agent, then another Agent is created.

May be opposed that this change in the algorithm is such that all Agents are not presented with the same size of candidates to bond with, must be remembered that the layer differentiation of the interaction has a similar effect also in the previous implementation.

Nevertheless this difference held a change in results that here we show for each algorithm before mentioned.

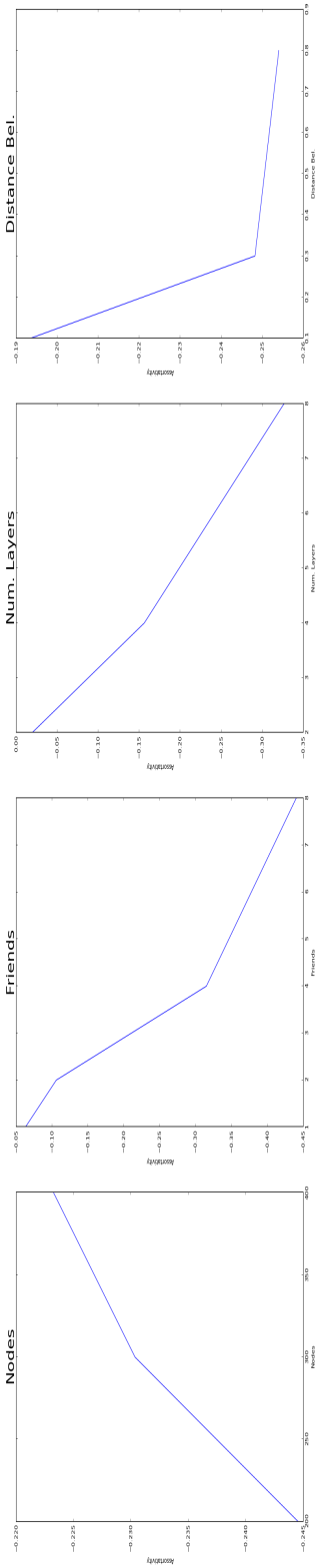


Figure 5.11: Degree assortativity related to changing parameters

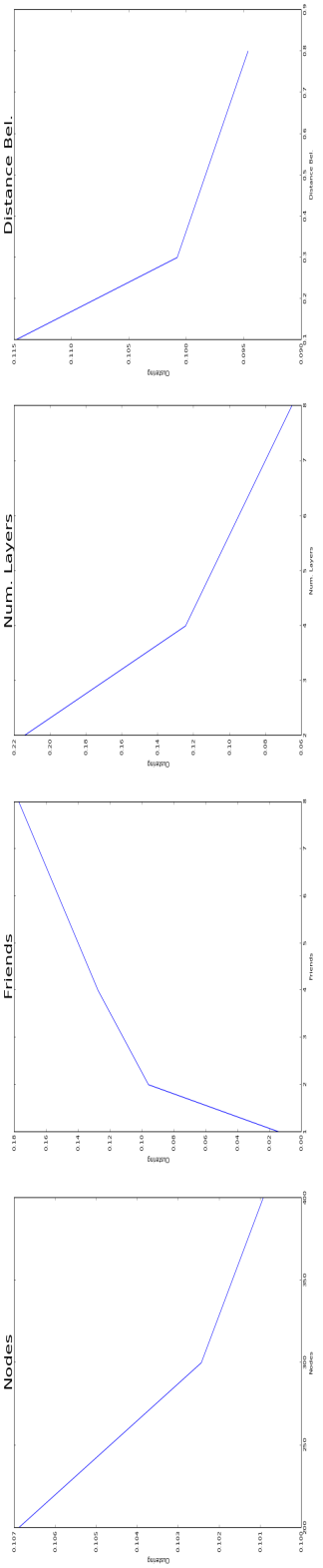


Figure 5.12: Clustering Coefficient related to changing parameters

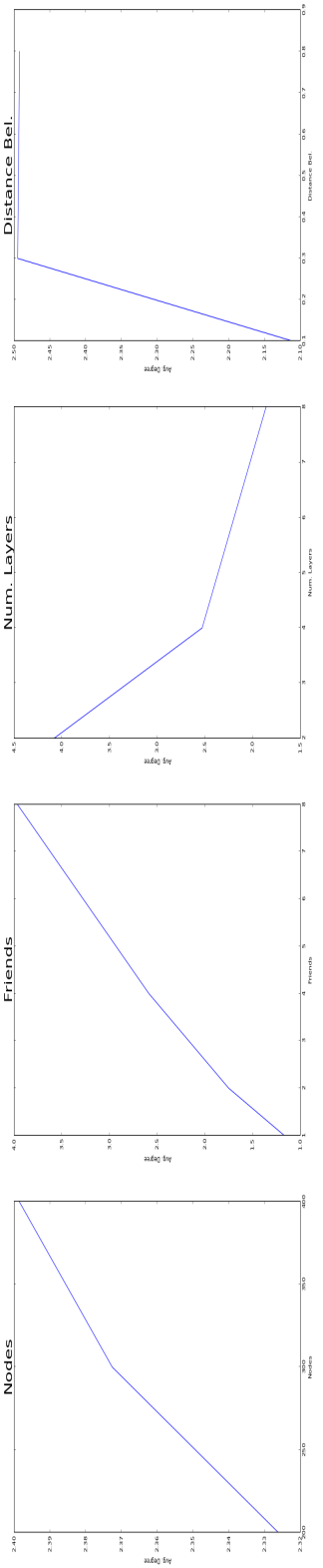


Figure 5.13: Average degree related to changing parameters

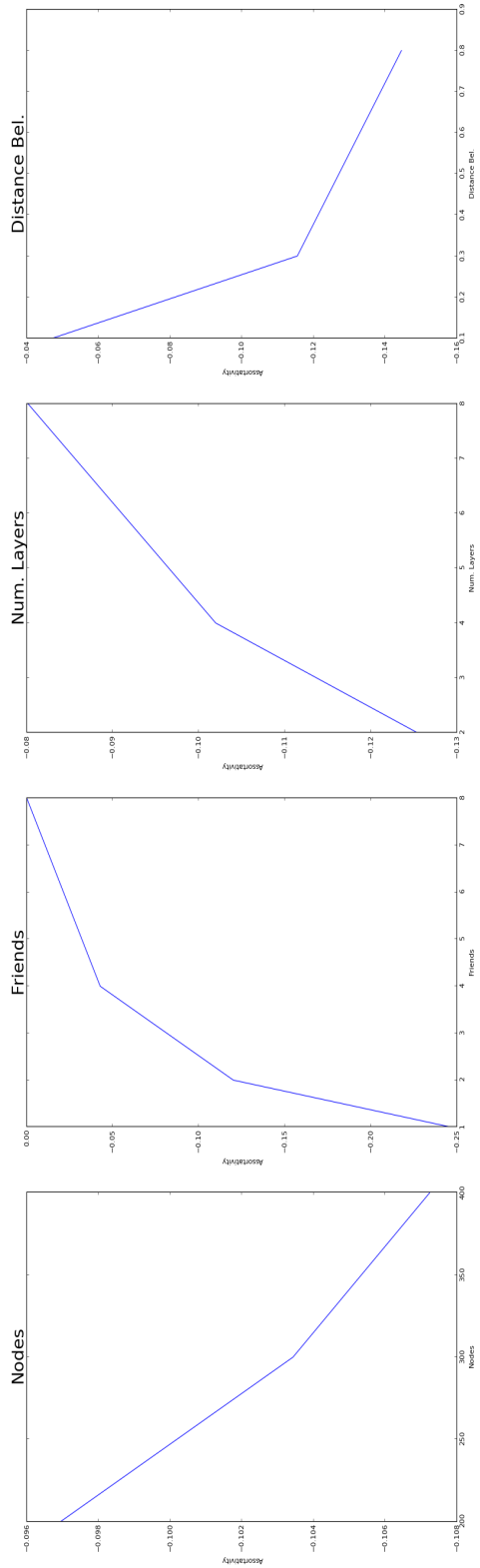


Figure 5.14: Degree assortativity related to changing parameters

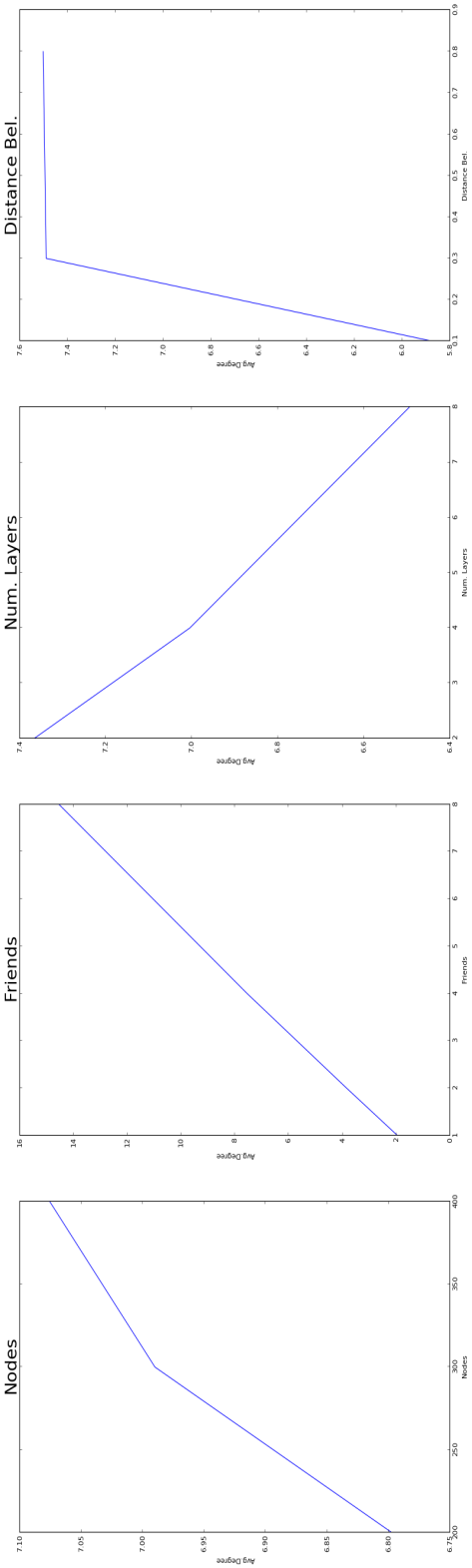


Figure 5.15: Average degree related to changing parameters

Random Algorithm

At each step an Agent is created, then algorithm 7.3.1 is run for the new Agent. We explored a parameters space evaluating three main characteristics of the generated network: degree assortativity, clustering coefficient and average degree.

The variable parameters chosen are:

- number of Agents created in the network
- number of desired friends for each Agent, consider that not all Agents reach the desired number of friends and some exceed that
- number of layers that can be accessed
- number of beliefs to be simultaneously close enough in order to create a link between two Agents
- the acceptable distance for two beliefs to be considered close one another
- total number of beliefs possessed by an Agent

Since layers are randomly chosen, the result is averaged over all layers.

As in the first implementation, in 5.16, 5.17 and 5.18 we do not see a clear region of parameters in which degree assortativity is positive, some values of numbers of layers held a positive value.

Furthermore we considered the projection network of the generated multiplex, holding the same interpretation as before, to search for emerging properties looking at degree assortativity and average degree, here in 5.19 and 5.20.

In these result for the projection network we start to see consistent regions of the parameters space in which the degree assortativity is positive yet small.

We proceeded examining the result for the other algorithms used in the first implementation to better explore the behavior of our social network.

Layer-Wise Beliefs

At each step a Agent is created, then algorithm 7.3.2 is run for the new Agent. We explored a parameters space evaluating three main characteristics of the

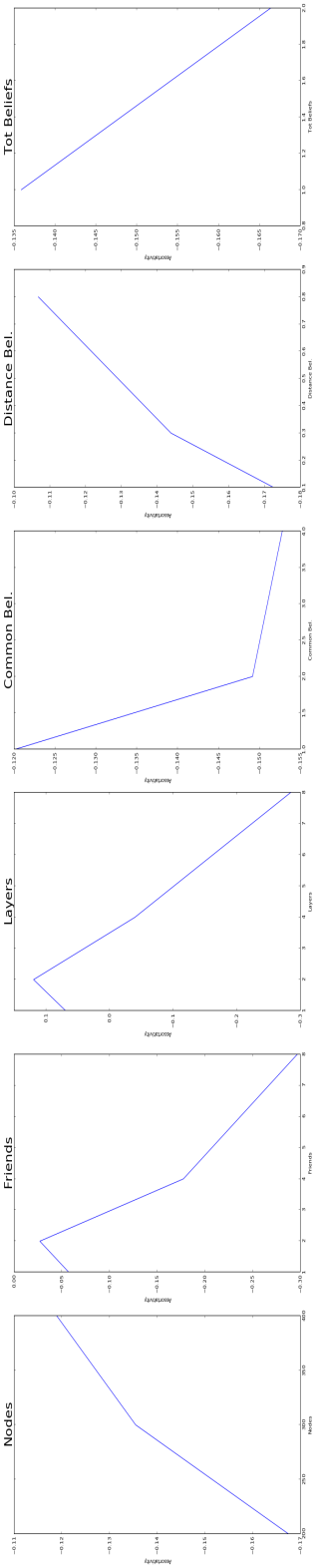


Figure 5.16: Degree assortativity related to changing parameters

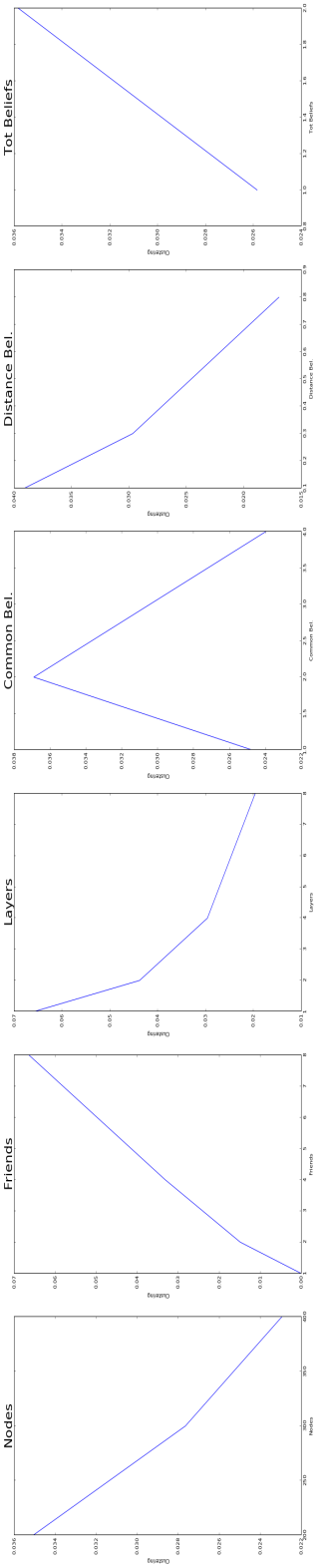


Figure 5.17: Clustering Coefficient related to changing parameters

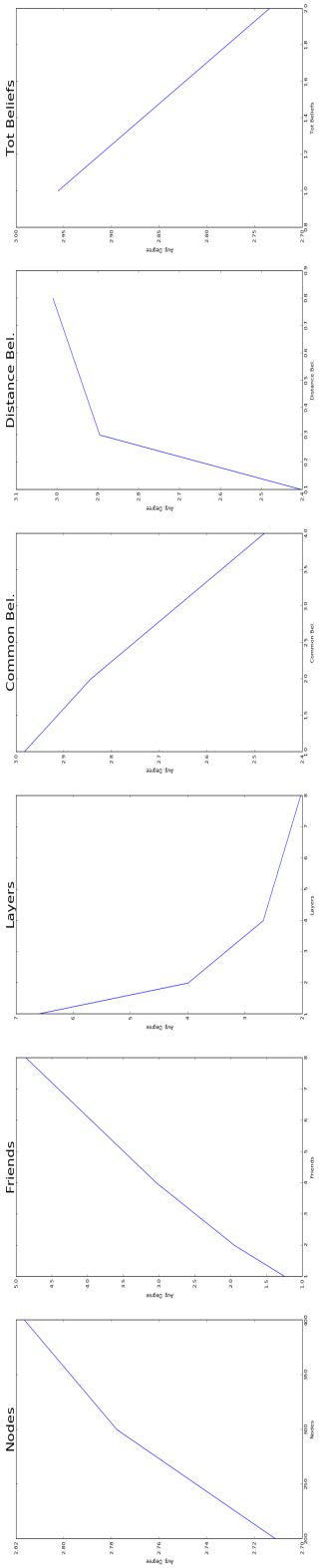


Figure 5.18: Average degree related to changing parameters

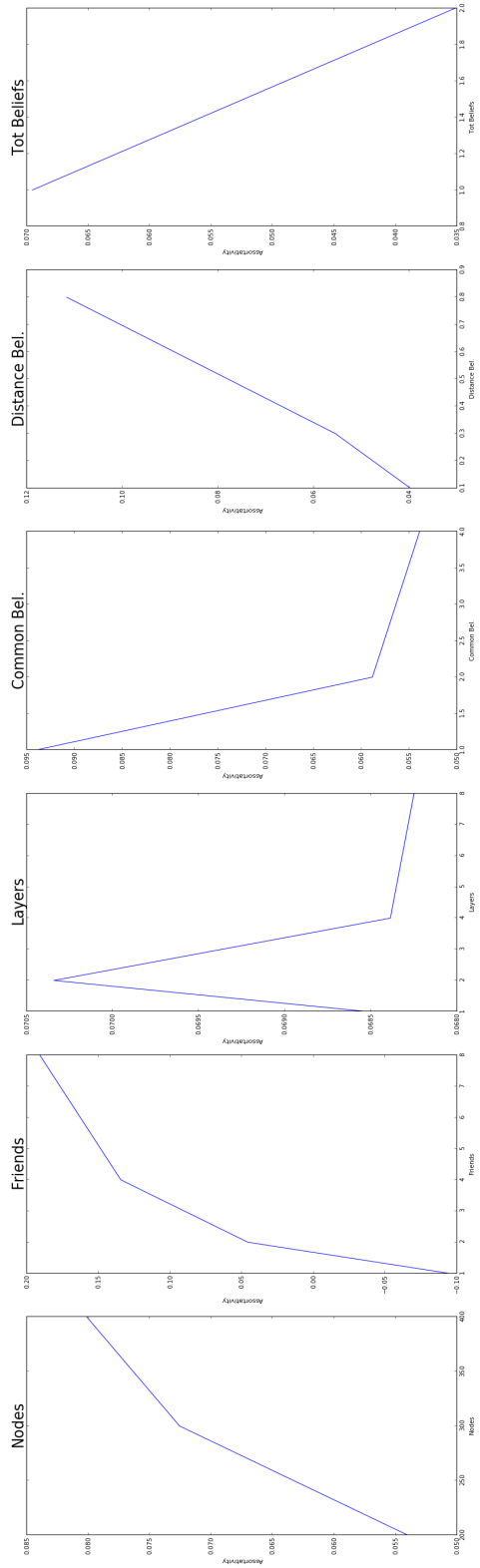


Figure 5.19: Degree assortativity related to changing parameters

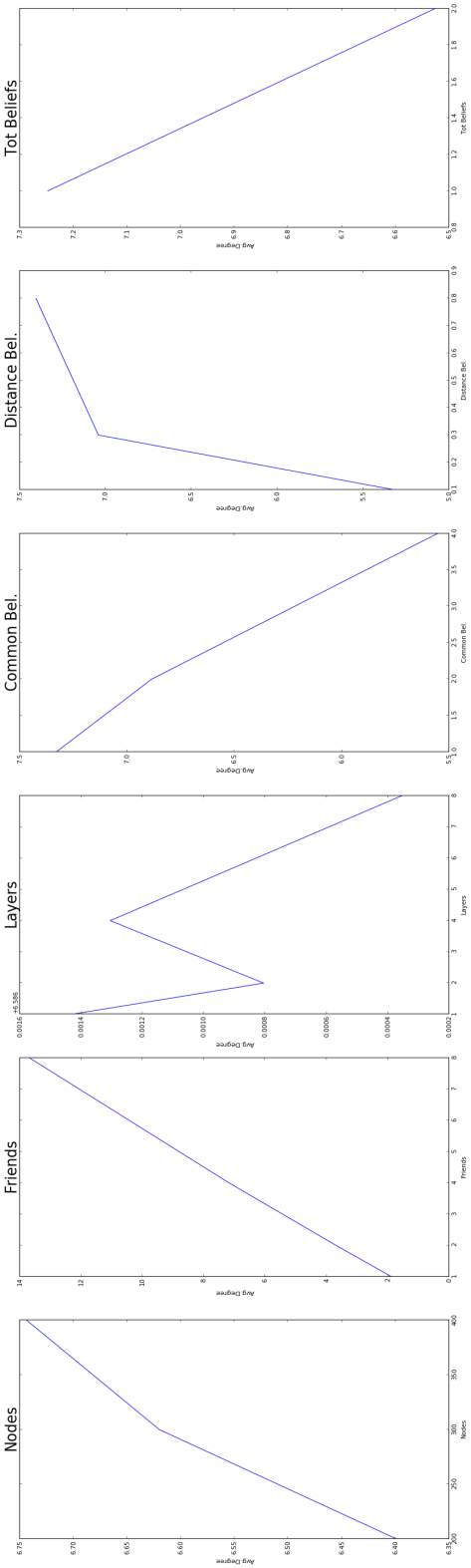


Figure 5.20: Average degree related to changing parameters

generated network: degree assortativity, clustering coefficient and average degree.

The variable parameters chosen are:

- number of Agents created in the network
- number of desired friends for each Agent, consider that not all Agents reach the desired number of friends and some exceed that
- number of layers that can be accessed
- the acceptable distance for two beliefs to be considered close one another

Since layers are randomly chosen, the result is averaged over all layers.

Here in 5.21, 5.22 and 5.23, we start to see a remarkable difference with respect to previous algorithms since there is a region of parameters that held positive values for degree assortativity and the clustering coefficient.

Furthermore we considered the projection network of the generated multiplex, holding the same interpretation as before, to search for emerging properties looking at degree assortativity and average degree, results are in 5.24 and 5.25 .

In the projection network we clearly have a wide region with positive values for degree assortativity, telling us that we made a first, clear, step in the direction of creating a network that held characteristics of a social network.

Friend First

At each step a Agent is created, then algorithm 7.3.3 is run for the new Agent. We explored a parameters space evaluating three main characteristics of the generated network: degree assortativity, clustering coefficient and average degree.

The variable parameters chosen are:

- number of Agents created in the network
- number of desired friends for each Agent, consider that not all Agents reach the desired number of friends and some exceed that

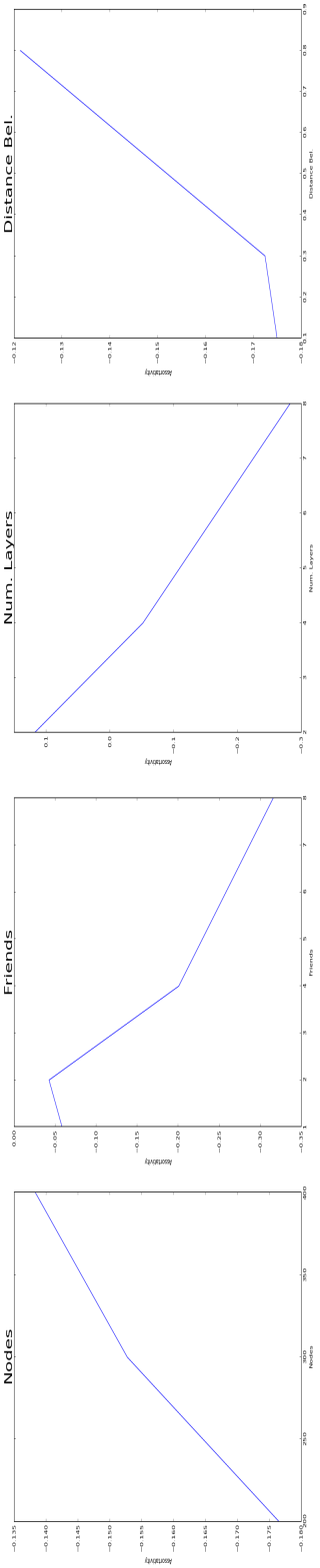


Figure 5.21: Degree assortativity related to changing parameters

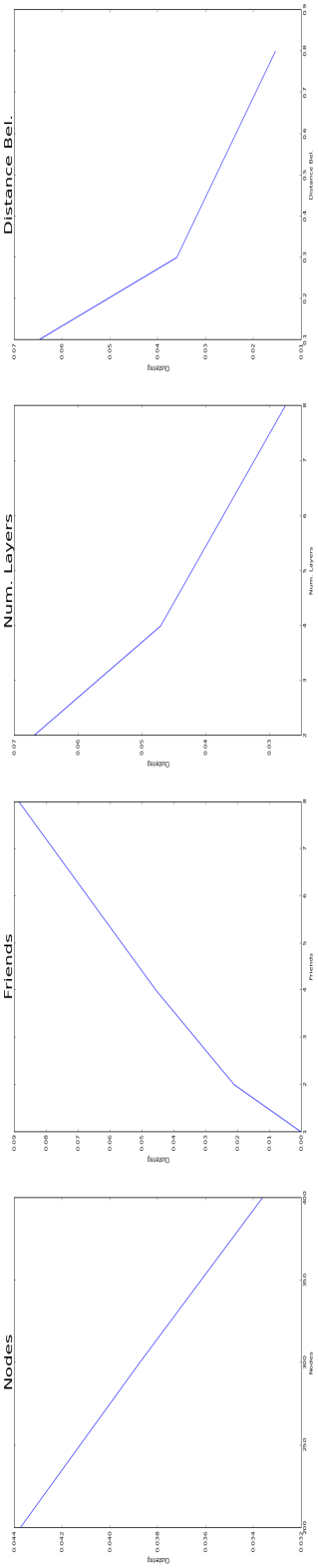


Figure 5.22: Clustering Coefficient related to changing parameters

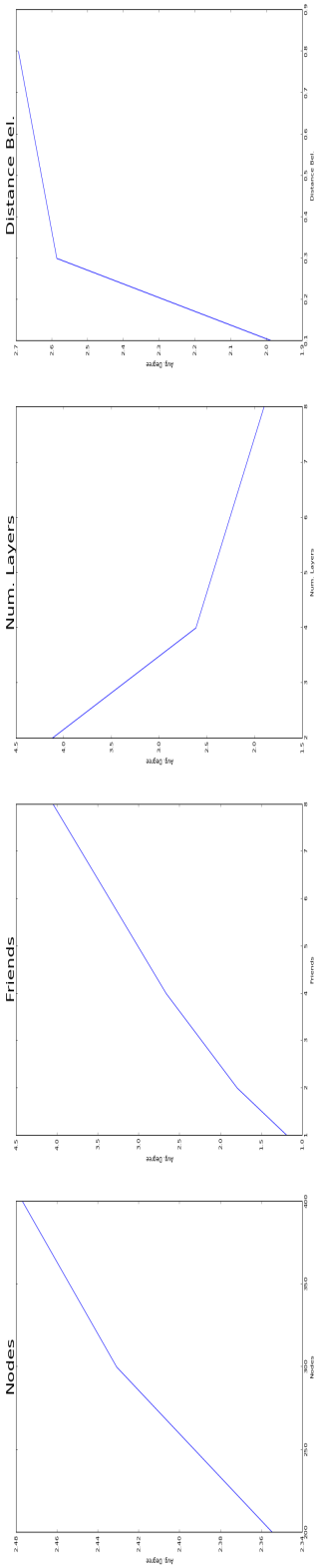


Figure 5.23: Average degree related to changing parameters

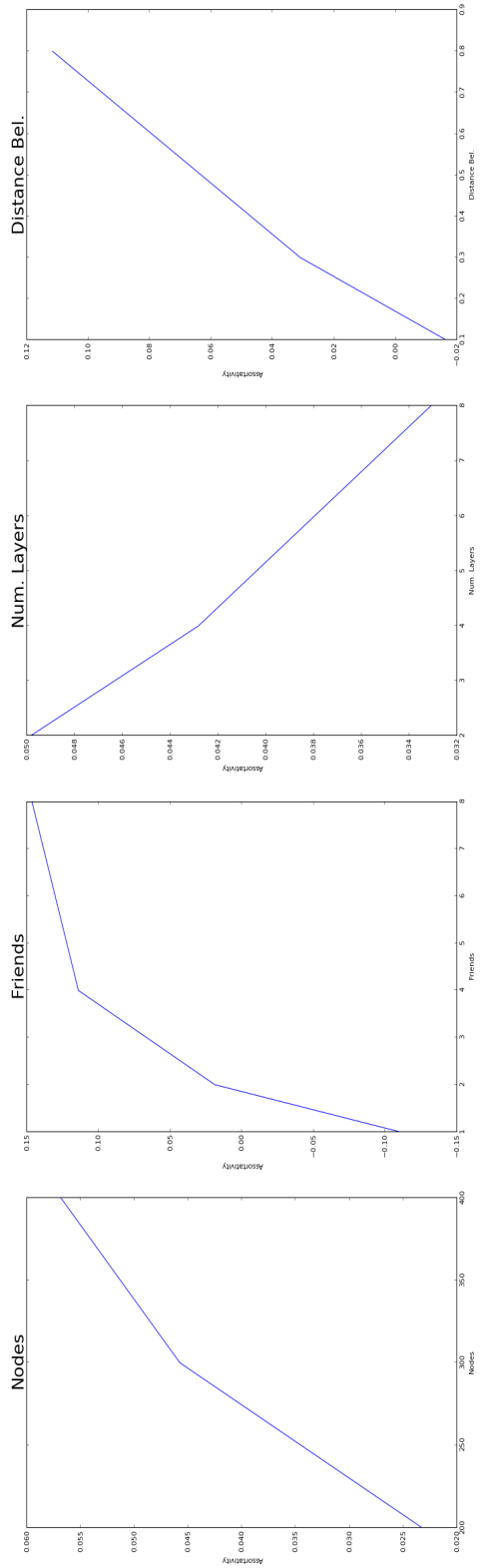


Figure 5.24: Degree assortativity related to changing parameters

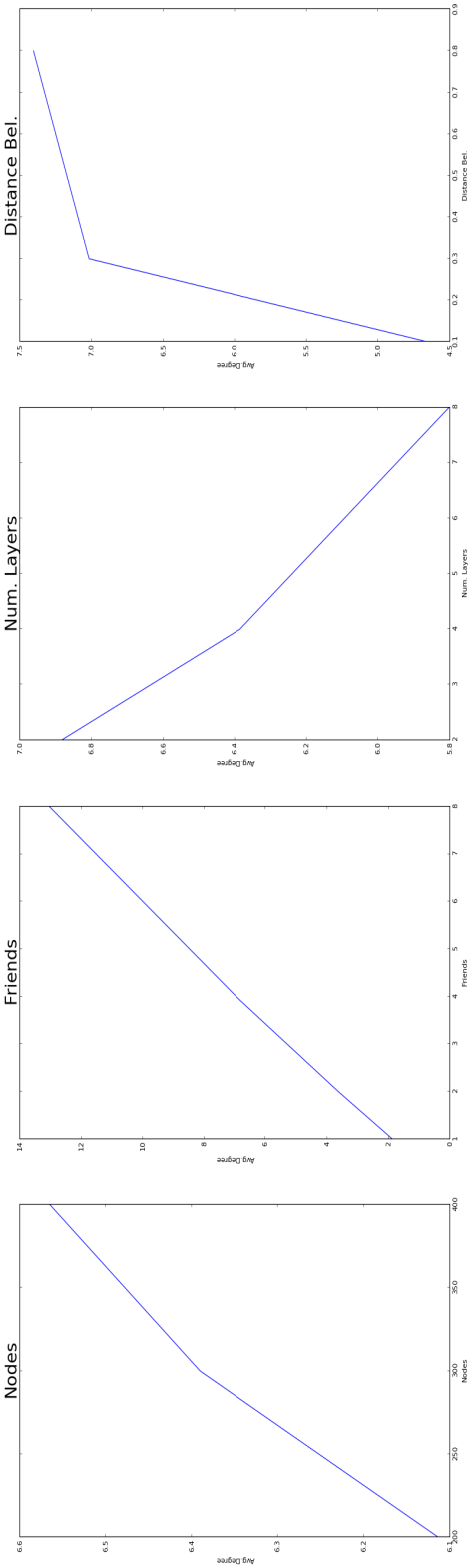


Figure 5.25: Average degree related to changing parameters

- number of layers that can be accessed
- the acceptable distance for two beliefs to be considered close one another

Since layers are randomly chosen, the result is averaged over all layers.

Again in 5.26, 5.27 and 5.28, we see a region of the parameters space indicating that a positive degree assortativity and a positive clustering coefficient is an emerging property in our layers.

Furthermore we considered the projection network of the generated multiplex, holding the same interpretation as before, to search for emerging properties looking at degree assortativity and average degree, shown in 5.29 and 5.30.

As with algorithm 7.3.2 we see a clear region of positive degree assortativity, indicating that the algorithm is generating a network with clear and measurable properties.

5.1.3 Consideration

As clearly shown in this second implementation, we obtain far different results for all three algorithms. Several parameters held positive values for degree assortativity and clustering coefficient on average on the layers. The parameters in which these measures are negative on average on the layers, the degree assortativity is positive if calculated on the projection network.

It may be interesting to compare the graphs of the first implementation with those of the second one. In particular, regarding degree assortativity, we can see a change in the behavior for some parameters.

For instance, one can note that in 5.1 and 5.16 there is a clear change of behavior for the value of degree assortativity related to the following parameters: the number of beliefs that have to be simultaneously close enough in order to create a link between two Agents; the acceptable distance for two beliefs to be considered close to one another; the total number of beliefs owned by an Agent. Similar differences can be found respectively in 5.4, 5.5 and 5.19, 5.20; 5.6 and 5.21; 5.9 and 5.24; 5.11 and 5.26; 5.14 and 5.29.

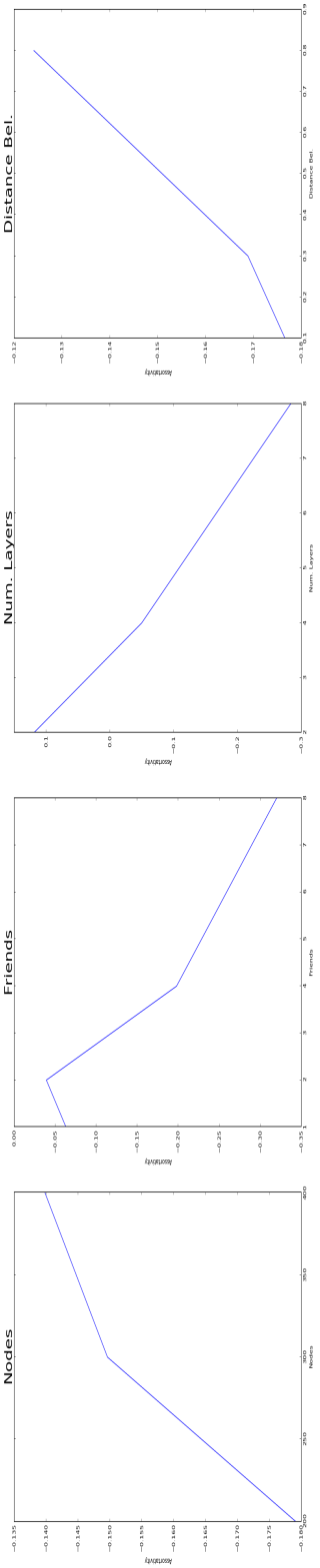


Figure 5.26: Degree assortativity related to changing parameters

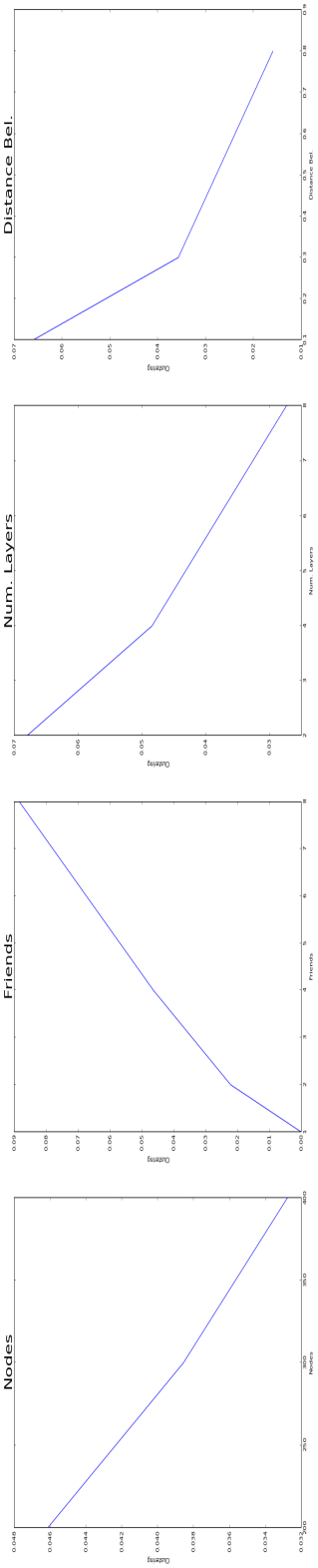


Figure 5.27: Clustering Coefficient related to changing parameters

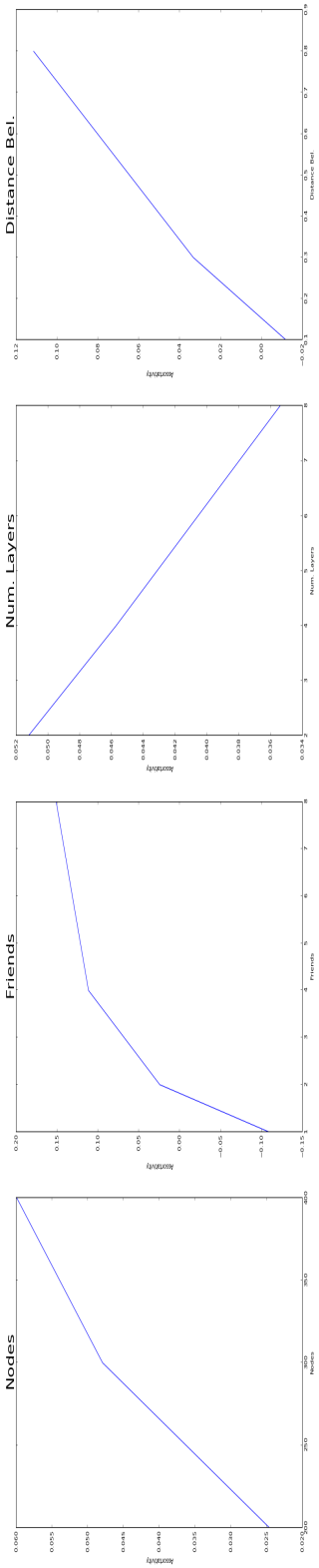


Figure 5.28: Average degree related to changing parameters

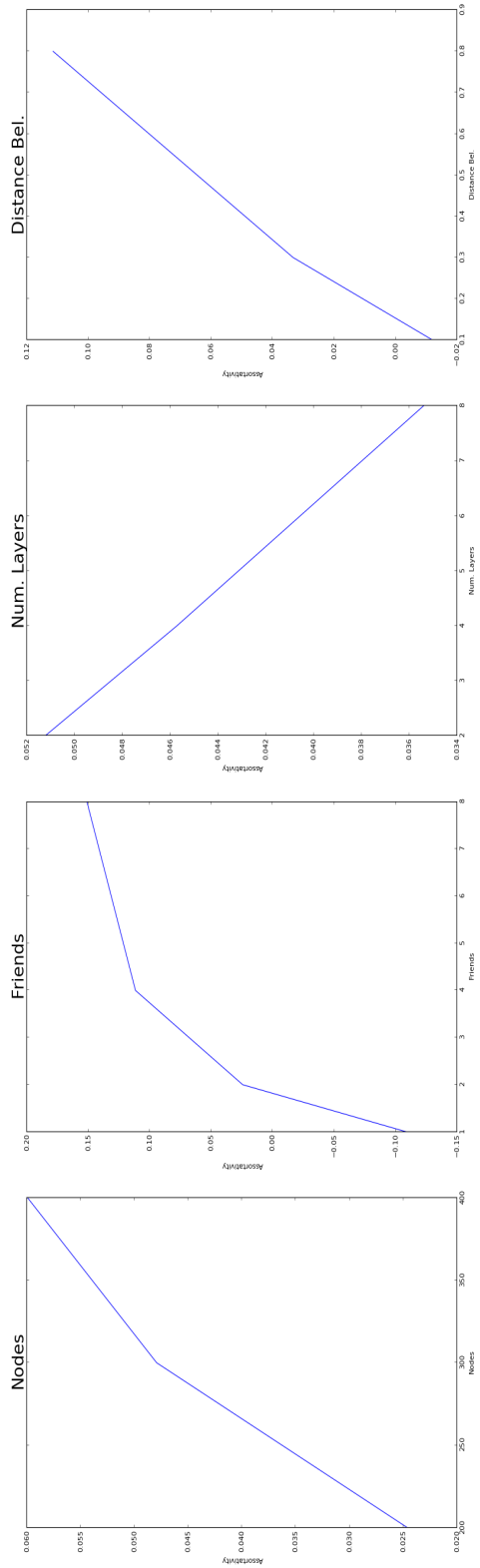


Figure 5.29: Degree assortativity related to changing parameters

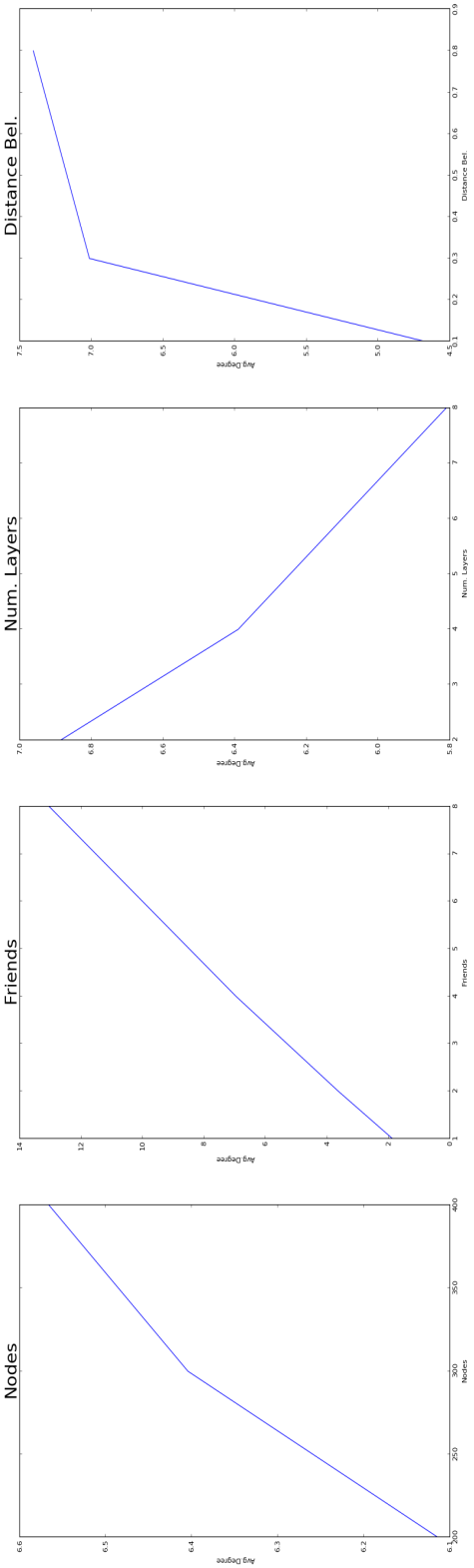


Figure 5.30: Average degree related to changing parameters

In particular, the change of behavior for the parameters of number of layers, only found in multilayer networks, and the acceptable distance for two beliefs to be considered close to one another, it could possibly be the reason why we obtain a positive degree assortativity in the second implementation but not in the first.

Beside the interpretation of each layer, here we see that positive degree assortativity is an emergent property of the projection network of the multiplex, even if the algorithm does not implement any rules of preferential attachment or any probability of link creation related to the degree of the Agents.

We have hence obtained a network that held some of the well-known properties of social network but originated from an Agent-Based interaction without the requirement of preferential attachment.

From this point we know the range of feasibility for our parameters, and we will be using randomly generated parameters in such range in all our simulations. This guarantees that all our simulations will have the aforementioned properties, that we will not influence any results, and that we will be able to generate a strong enough statistics for all analysis needed.

5.1.4 Degree distribution

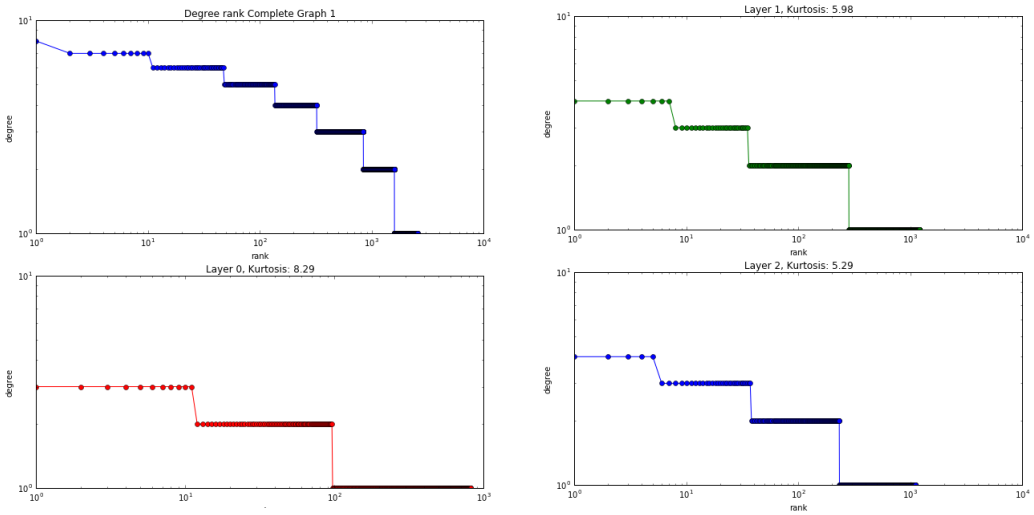
We take a look at the degree distribution of the layers and the projection network. We collected a statistic of self-generated networks using both algorithm 7.3.2 and 7.3.3, varying the number of Agents in the network and the number of layers, having chosen for other parameters randomly generated values suitable with previous analysis.

Here is shown a typical results for a generated network and its layers.

The degree distribution is clearly exponential, an analysis with respect to some changing parameters is conducted in order to verify that a linear fit with the logarithm of the data is consistent. For such purpose we used a χ^2 test, divided by the degree of freedom.

As shown in the following measure, generated with algorithm 7.3.2, the test value is widely below one for all parameters examined both on average over all layers, 5.32, and on the projection network, 5.32.

Figure 5.31: Example of degree distribution



Similarly for algorithm 7.3.3, the test value is widely below one for all parameters examined both on average over all layers, 5.34, and on the projection network, 5.34.

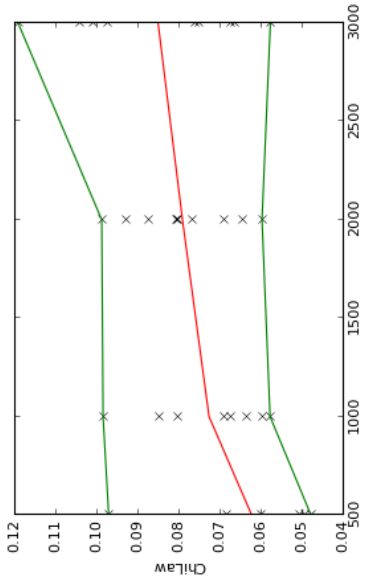
We can therefore accept without doubt that the degree distribution generated by our algorithm 7.3.2 and 7.3.3 is compatible with a power law.

These results are of great interest and, joined with the previous results about degree assortativity and clustering coefficient, we further show that our self-generated network express all the characteristic of a social network.

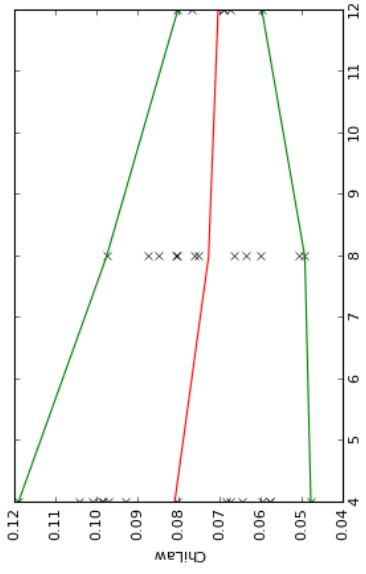
A small deviation arise from the analysis of the exponent, in the projection network and on average over all layers, is smaller than the accepted range of $2 < \gamma < 3$.

Our fit return a value for $\gamma \sim 0.5$ both for algorithm 7.3.2 and 7.3.3, our degree distribution is not *steep* enough and our hubs not rich enough. This may be due to the relatively low number of Agents in our simulation, especially if compared with the large number, up to twelve, of layers that we testes for our multilayer network. Our configuration may lead to some "underpopulated" layers, this supposition may be investigated in the future with a greater computational power at disposal.

Nevertheless we decided to accept the network generated by algorithm 7.3.2 and 7.3.3 as a good enough first implementation of an Agent-Based model

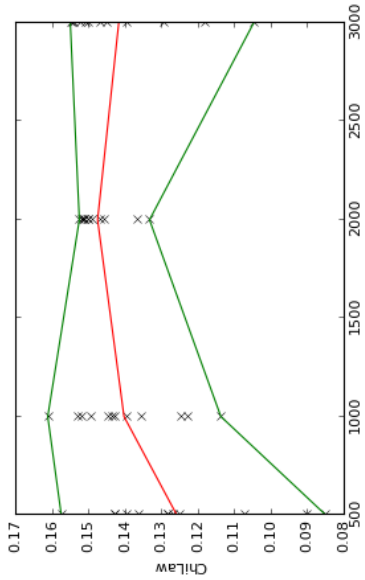


(b) vs Agents

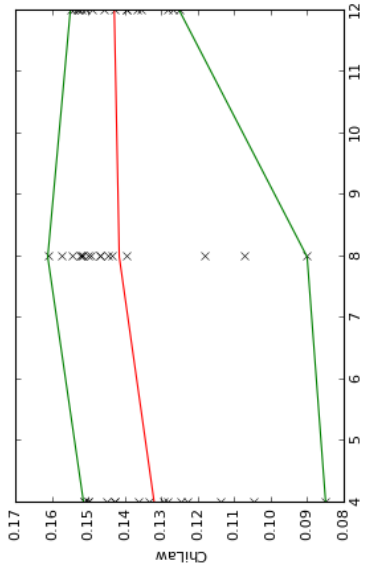


(a) vs layers

Figure 5.32: χ^2/DF, averaged over all layers, algorithm 7.3.2

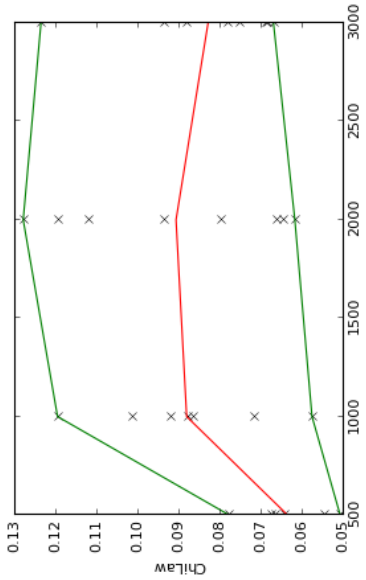


(b) vs Agents

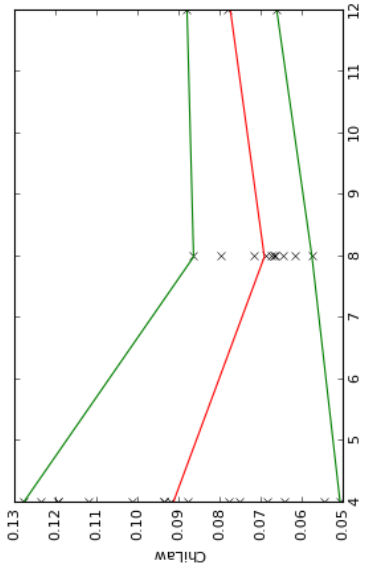


(a) vs layers

Figure 5.33: χ^2/DF, on projection network, algorithm 7.3.2

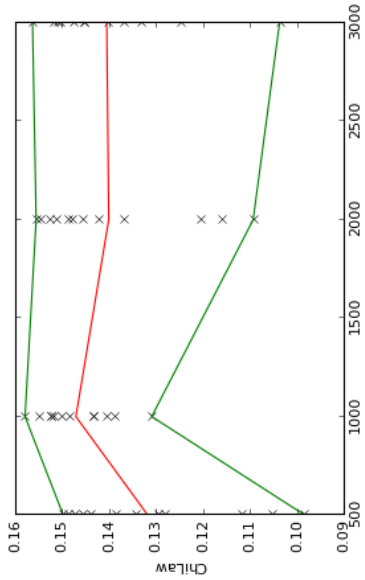


(b) vs Agents

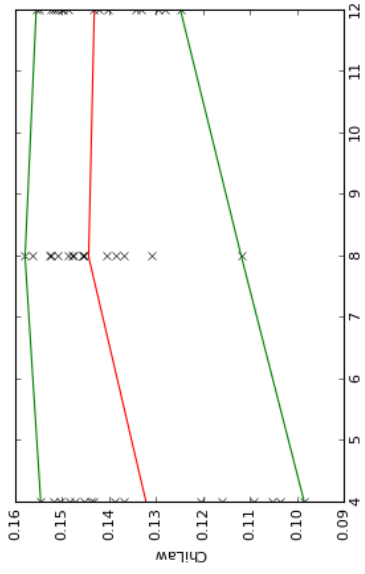


(a) vs layers

Figure 5.34: χ^2/DF, averaged over all layers, algorithm 7.3.3



(b) vs Agents



(a) vs layers

Figure 5.35: χ^2/DF, on projection network, algorithm 7.3.3

generation for multilayer social-network and will proceed in developing and using one of those algorithms for our simulation.

5.1.5 Further improvements

To maintain a wide enough generalization but in order to be keen to an instrumental interpretation of each layer, a reasonable choice would be to adopt the algorithm *AttachChoosenBelief* as in 7.3.2 for the generation and development of our simulation.

Further research in the field of multiplex, Magnani and Rossi [2013], show that not only *degree assortativity* should be positive but *degree centrality* must be positively correlated for the Agents that participate in multiple layers. An analysis of the algorithm 7.3.2 and on algorithm 7.3.3 shows that such properties weakly emerge in our multiplex.

We analyzed a small number of self generated network with parameters randomly generated but suitable with previous analysis in order to have a deeper look at the code needed to reproduce the results of the aforementioned article without loss of generality and maintaining adherence with the framework of our simulation.

The measure shows a positive value for Pearson coefficient in the *degree centrality* of those Agents that participate in more than one layer, in a moderate portion of the layers.

Here we show a typical result of correlation between *degree centrality* for two layers. To be coherent with the result of the aforementioned paper, only Agents present in all layers have been considered.

5.1.6 Remarks

The algorithm 7.3.3 and 7.3.2 held very similar behavior and measures seeming interchangeable, showing both positive *degree assortativity*, but also a positive correlation of *degree centrality* across layers, plus a positive *clustering coefficient*, but with different interpretation.

Further investigation could be made, but for the moment we will use algorithm 7.3.2 to generate a multiplex, this algorithm offers a more clear mecha-

Figure 5.36: Degree centrality correlation between two layers with algorithm 7.3.2

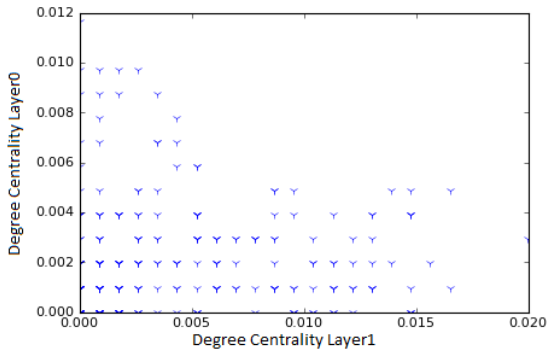
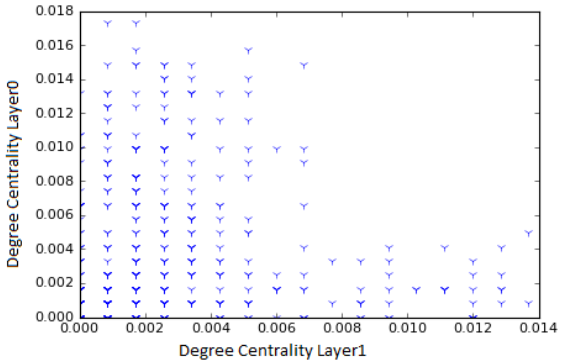


Figure 5.37: Degree centrality correlation between two layers with algorithm 7.3.3



nism of bound creation in terms of human interactions.

5.1.7 Beliefs distance

Our network, based on algorithm 7.3.2 is therefore created based on Agents beliefs and distances among them. It seems interesting therefore to measure the distance between two arrays of beliefs to see how an Agent has *chosen* her friends.

Remembering that in each layer not all belief are considered, we do not expect this measure to be trivial as the compartecipation of friends from different layer creates for an Agent a rich set of neighbors, in terms of beliefs among them. We did this measure with the algorithm 7.4.1, it measure the

module of the difference of the vector of beliefs, either fixed or variable, of an Agent and her neighbors in all layer, or capturing the neighbors of an Agent in the projection network.

What we found is that the distribution of distance is generally far from Gaussian and held a positive skewness, meaning that the distribution is asymmetrical towards smaller values with respect to the average.

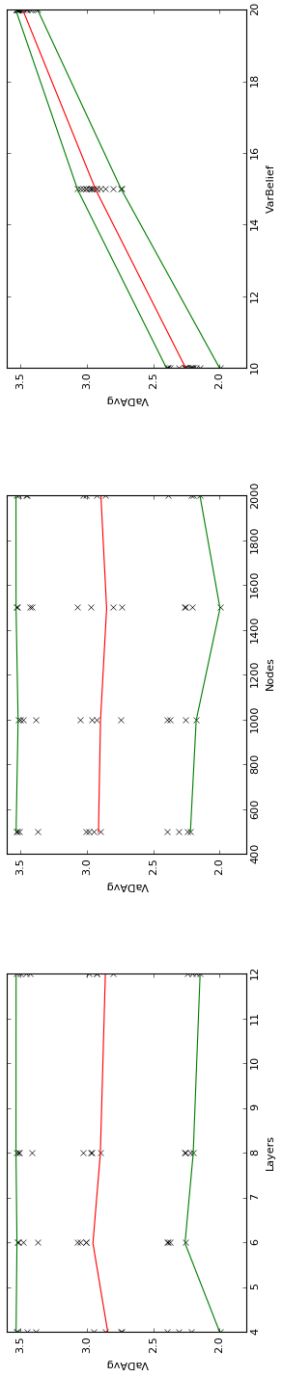
That can be easily inferred by the algorithm used: each Agent searches for other Agents with similar beliefs, being that this search can vary from layer to layer, it sure leads to an uneven distribution in the distances of beliefs of the neighbors.

In the following graph we will show the average distance of the array of variable beliefs both on the projection network and on the average of all layers, for changing values of:

- Total number of Agents
- Total number of accessible layers
- Number of variable beliefs possessed by an Agent

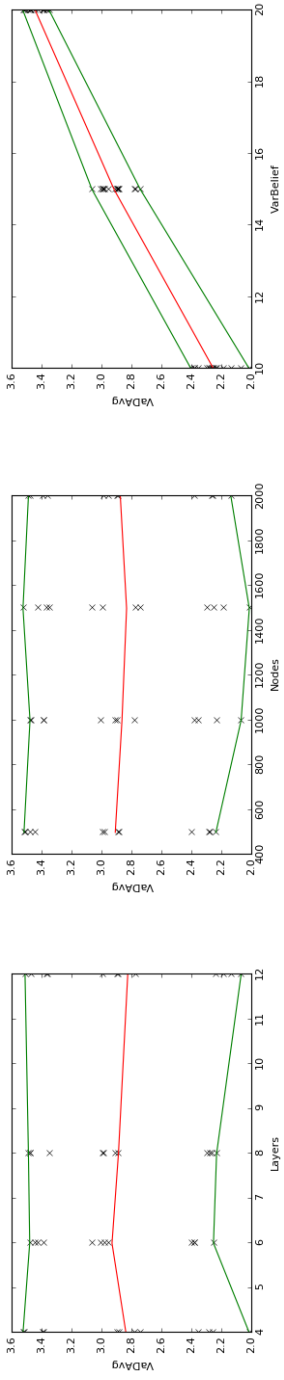
As we can see in 5.38 and 5.39, the distribution average of the distance of beliefs relative to an Agent and her neighbors is almost flat with respect to the variation of number of Agents and number of layers, but show a clear positive correlation with the total number of beliefs possessed by an Agent, both on the average in all layers and in the projection network. This is intuitive as more dimensions a vector has, higher is the probability that, being the link created only on the basis of some values of the whole vector, the vector of beliefs are not one near the other. No great differences are found comparing this measure on average over all layers and on the projection network. Is to be noted that the standard deviation is generally high, meaning an high possibility of great deviation or an insufficient statistics, as always limited by the available computational power.

In 5.40 and 5.41 we see an analysis of the skewness of the distribution of the distances. At first sight we see that the skewness in almost every case flat with respect of changing parameters. However it appear to be a slightly correlation



(a) vs layers (b) vs Agents (c) vs number of beliefs

Figure 5.38: Distribution average, averaged over all layers



(a) vs layers (b) vs Agents (c) vs number of beliefs

Figure 5.39: Distribution average, on projection network

in 5.41 (a) and (c), nevertheless insufficient to claim any clear relation. As before we note that the standard deviation is generally high, meaning an high possibility of great deviation or an insufficient statistics, as always limited by the available computational power.

A comparison between randomly generated multilayer network, in figure 5.42 and one generated with algorithm 7.3.2, 5.43 has been made in order to show how a typical distribution of beliefs distances differs with our algorithm.

Figure 5.42: Neighbors Agents vs belief distance with algorithm 7.3.2

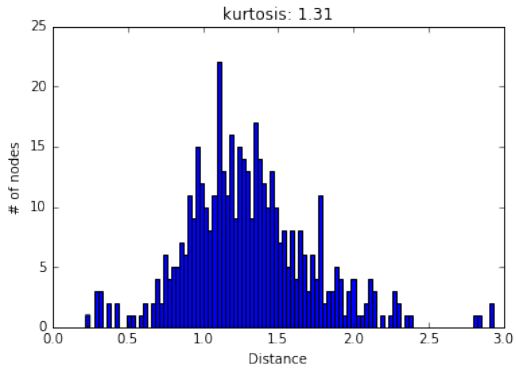
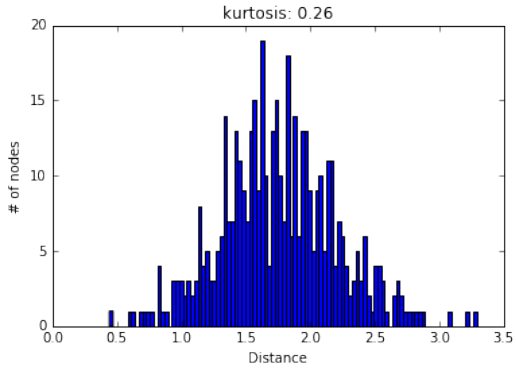
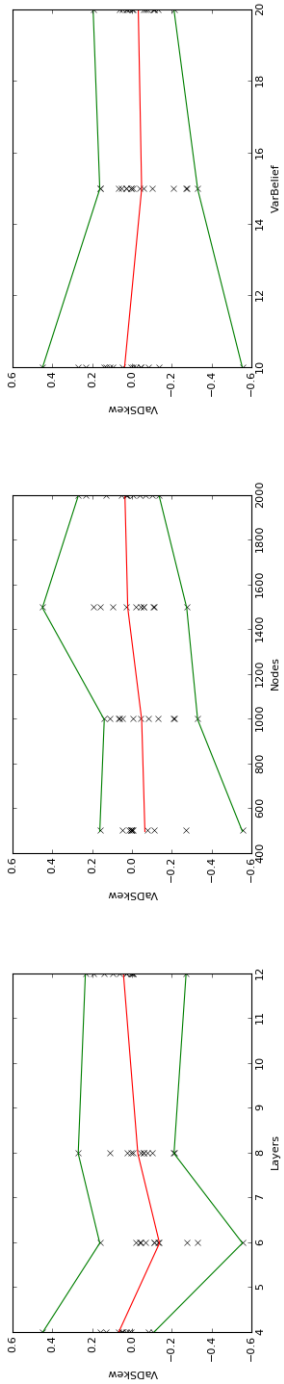


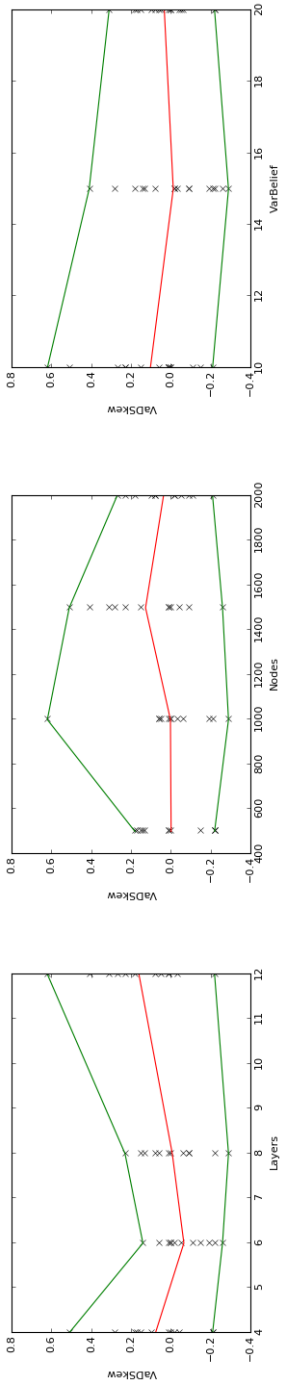
Figure 5.43: Neighbors Agents vs belief distance with random algorithm





(a) vs layers (b) vs Agents (c) vs number of beliefs

Figure 5.40: Distribution skewness, averaged over all layers



(a) vs layers (b) vs Agents (c) vs number of beliefs

Figure 5.41: Distribution skewness, on projection network

5.2 Influence

In all real network there is some sort of interaction between Agents, and the result of these interactions is ultimately one of the most interesting features to study, in our experiment we have an exchange of information between Agents, in the form of mutual influence of some beliefs. This diffusion of information will change the *ideas* of our Agents and the way they decide to vote.

We expect from our multilayer network a complex behaviour, inspired by the work of Diakonova et al. [2016], which demonstrated the non reducibility of the simple voter model in a two-layer multilayer network to a single layer. The main difference in the diffusion of ideas in a multilayer, as can be easily deduced, is in the partecipation of an Agent to more to one layer and therefore to more chances of being influenced in different ways, fhurter can be read in Boccaletti et al. [2014].

5.2.1 Spreading Beliefs

For the process of spreading beliefs and ideas, getting inspiration from Quattrociocchi et al. [2014], we will use the *Bounded Confidence Model (BCM)* first proposed by Deffuant et al. [2000]. In the proposed interaction model two Agents will interact only if the beliefs or ideas they share or choose to look at are not too distant. This model of interaction rises from the ideas that people tent not to interact if there is not a common field of discussion or if their beliefs and characteristics are too different for them to start some sort of relation.

Being influenced

The *BCM* proposes that beliefs / ideas are influenced among Agents trough two parameters: one expressing the existing distances between beliefs / ideas of Agents and the other expressing the magnitude of the influence (a greater parameter will mean an higher change in the beliefs / ideas of the Agent). In our model each Agent will posses an array of beliefs, each value i of the array can be confronted with a neighbor and influenced following the equation: $x_i = x_i + \mu(x'_i - x_i)$ if the distance is $\mid x_i - x'_i \mid < d$. The magnitude of influence,

μ, will be Agent-wise, interpreting that all Agents are differently influenced by others, the acceptable distance, d, will be fixed. A small random probability of the event not to happen will be added.

Further investigation could take in consideration parameters μ, d to be layer-wise or Agent-wise. In his paper Deffuant found that parameter μ and the number of Agents seems to influence only the time of convergence, but all simulation where held on regular, monolayer, lattice, thus changing those parameters, or making them dependent from layer or Agent may change the number and position of the point of convergence for the system.

First implementation

The algorithm implemented in 7.2.1 implement the Bounded Confidence Model, with parameter μ different for each Agent and parameter d different for each layer. Only variable beliefs will be influenced by other variable beliefs and only by neighbors of the Agent.

For each layer the set of beliefs to be changed will be different such to simulate different interaction between Agents depending on the context.

Summarizing we will have different sets of Agents, interacting in different layers on different set of beliefs for each layer with the *BCM*. The magnitude of influence will be different for each Agent. This seems a generalized enough situation to be analyzed in order to verify the effect of multilayer and the way our network as been created, always considering that other parameters have already been investigated and will be kept random in an acceptable range.

The run for ideas diffusion is implemented in function 7.2.2, an issue is how many cycles have to be made in order to have a good enough diffusion. The paper of Deffuant et al. [2000] suggest us an order of ten of thousands, in our work the only limit will be the computational resources: for what has been possible to do we observed no full convergence.

Measuring influence

We need a way to measure the diffusion of ideas on the network. Being the diffusion extremely complex, we cannot measure the direct diffusion of a single

change in the beliefs of each Agent. We chose to measure the *distance* between the vectors of the beliefs of an Agent and her neighbors as described in 7.4.1 and 7.4.2. Obviously the distance between fixed beliefs will not change during the diffusion of ideas.

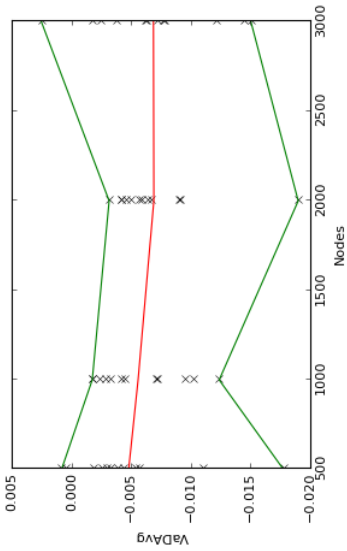
In the following graph we will show the difference between the average distance of the array of variable beliefs both on the projection network and on the average of all layers before and after a cycle of ideas diffusion and influence in the network, for changing values of:

- Total number of Agents
- Total number of accessible layers

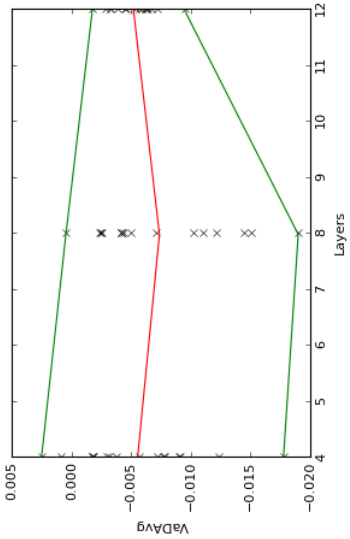
As in other simulation, all other parameters are randomly set, respecting previous analysis.

What found in 5.44 and 5.45 is that the distance of beliefs of an Agent in relation with her neighbors slightly diminish as the influence process is run, meaning that ideas are slowly getting closer for all Agents. As mentioned before, we did not have enough computational power to exclude a convergence of the model.

In 5.46 and 5.47 we see the variation in the skewness of the distribution of beliefs distances also slightly diminish as the influence process is run, meaning that the process of ideas diffusion tend to make the network more homogeneous.

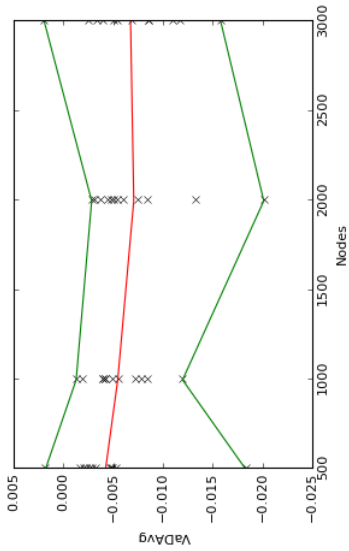


(b) vs Agents

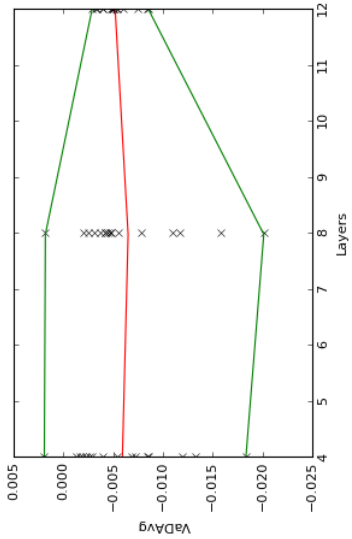


(a) vs layers

Figure 5.44: Average distance variation, averaged over all layers

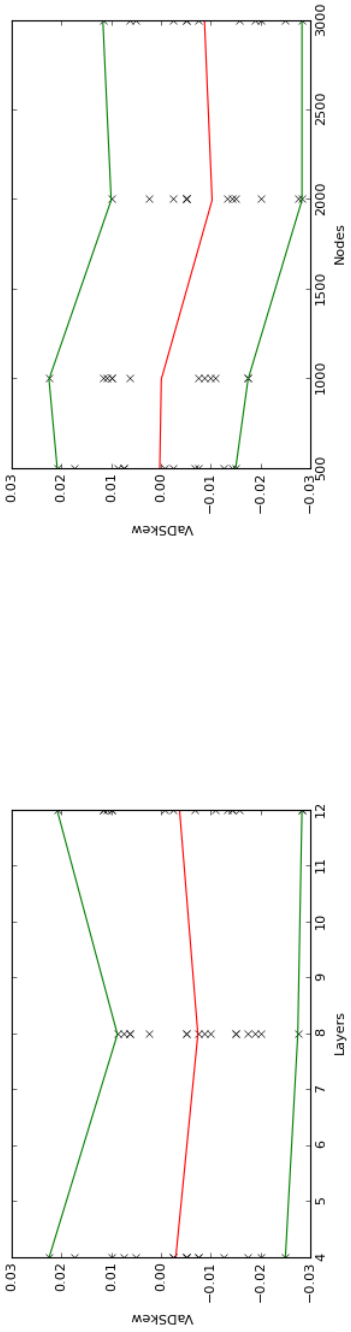


(b) vs Agents



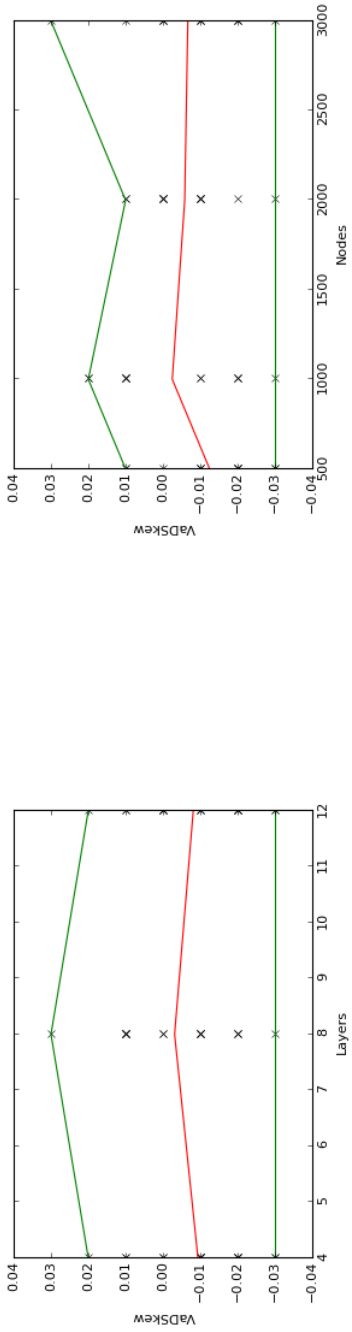
(a) vs layers

Figure 5.45: Average distance variation, on projection network



(a) vs layers (b) vs Agents

Figure 5.46: Skewness variation, averaged over all layers



(a) vs layers (b) vs Agents

Figure 5.47: Skewness variation, on projection network

5.3 Voting

The act of voting forces all citizen to interpret, process and summarize a whole set of information of the society in order to express a concise opinion or preference for a political view.

Modeling the processes that lead to vote is complex, first we need to model the spread of beliefs and ideas that form the cultural basis for interpreting information, then we have to find a way to model the interpretation of information, based on cultural differences, that leads to express a certain opinion in the vote.

Referendum

A referendum is one of the simplest form of voting in modern democracies, all citizen are to express their ideas about a specific matter either voting yes or no, abstention may be interpreted in different ways depending on the law of the state. These voting system seems to be easier to analyze and simulate, so at first our work will look at scenarios where Agents can only express a binary vote or choose not to vote.

Possible approaches

We have multiple possibilities to model the diffusion of ideas and the process of voting:

- Each Agent posses an array, this may represent her beliefs
- The vote may be a function converting the array of the Agent to a single value between minus one and one, or combining the array of beliefs with an array representing the matter of the vote in order to obtain a single value between minus one or plus one.
- The change in the intention of vote will be represented by the changes in the array of beliefs, due to influences of one Agent to its neighbors.

- Instead of being given externally, the array representing the matter of the vote may be diffused among Agents, eventually acquiring error or changes.
- Media may be added, influencing either the beliefs of Agents or the matter of the vote.

Adopting different combination of the aforementioned techniques, we can represent both the idea that voting is the result of changing ideas on different aspects, being influenced by acquaintances in different context and the fact that our knowledge of the argument of the vote is created during communication with our acquaintances.

For our experiment we chose to keep the following setting: the vote will be expressed with a inner product of the beliefs array and an array expressing the matter of the vote, the *issue*. Agents will be influencing one another only on the beliefs, as we deeply examined before.

The vote

The vote is implemented with algorithm 7.1.1 in which we use the inner product of the normalized vector of the issue and the vector beliefs, intended as the concatenation of the vector of fixed beliefs and variable beliefs. The issue is the same for all Agents for the same run and is randomly generated.

We expect that, if no influence between Agents has occurred, since both Agent's beliefs and the issue have been randomly generated, the distribution of the vote will be a normal distribution around zero, being the normalization of the voting process such that minus one and plus one are the extremes of the range.

We will interpret a positive vote as being "in favor" of the issue, and a negative one as being "against" the matter of the issue. Abstained people will be those in a small boundary around zero, interpreting that they are so uncertain, or have feeble opinion, about the issue that will tend not to vote. In order to have a better generalization we will mostly focus on the distribution of votes on the whole range of admitted voting values.

More complex mechanism of voting could be implemented, for example the "herd" effect, in which an Agent tent, if uncertain, to conform to her neighbor's ideas, of the "protest" effect, in which if uncertain the Agent will vote the opposite of what her neighbors do.

5.3.1 Voting in a Layer

We have now thoroughly analyzed our network: from its creation to the diffusion of ideas and the voting procedure.

What we want to explore now is how knowing the voting distribution in a layer can help us to infer the distribution on the projection network.

This is central to this research because, when a poll is performed, researchers have access to only one, or a limited number, of layers of interaction in a society. Then, once a good enough sample is drawn from the layers, researchers try to infer the true distribution of ideas and beliefs in the whole society trying to predict what the vote outcome will be.

Thanks to the simulation we created, we have a perfect knowledge of both the distribution of beliefs, and therefore the voting outcome, on the projection network and in every single layer that composes our multilayer social network.

We decided to directly compare the voting distribution of each layer with the voting distribution of the projection network. Keep in mind that all parameters are generated randomly, in a suitable range previously discussed, and the free parameters are therefore few, namely the number of Agents and the number of layers.

To compare the distribution we adopted the χ^2 test as proposed in "Numerical recipe for C", Press et al. [1992], divided by the degree of freedom and then averaged for all layers. What will be shown in figure 5.48 and 5.49 is therefore the average of the χ^2 test, divided by the degree of freedom, for every layer with respect to the projection network.

What we see from that analysis is that we can not accept the distribution of votes found in a layer to be representative of the whole network. The value of χ^2 divided by the degree of freedom, if near one, expresses the compatibility between two distributions, here the value is always higher than one.

Figure 5.48: χ^2 test / DF , averaged for all layers vs layers

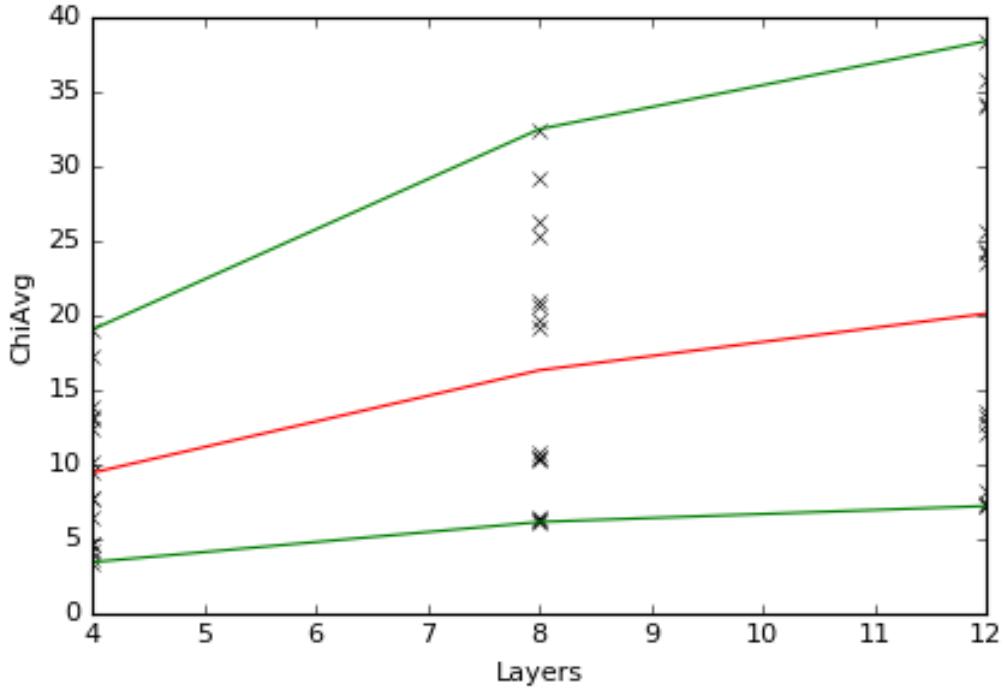
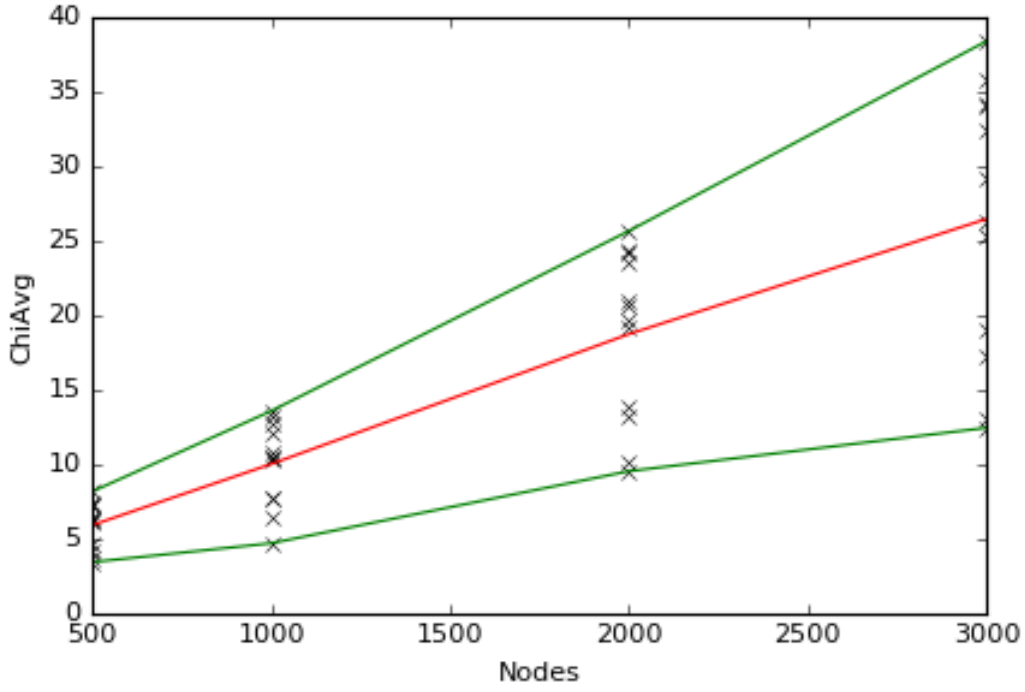


Figure 5.49: χ^2 test / DF , averaged for all layers vs Agents



We see that the value of the test is positively correlated both with the number of layers and the number of Agents. More layers intuitively represent more "partitions" of the network and therefore there is a higher probability that wider differences with the projection network may emerge in one of the layers.

These results tell us that, if our society is really better represented by multilayer network than single-layer one, the perfect knowledge of the distribution of voting, or any other measure, on a single layer is not representative of the same measure on the society as a whole.

Could this be a limit to our possibility to investigate the diffusion of ideas in the real world?

5.4 Polling

Lastly, we propose a simple example of what means to make a poll in a society.

Our access to the structure of the multilayer network will be limited to few layers and a subset of the totality of the Agents, as it is in a real-life situation in which researchers only have access to one or few communication systems, as examined in previous chapters, in which they try to get the answer from a sample of the population.

5.4.1 Simple draw

A simple poll in our network will look like 5.51.

What essentially is done is a random draw of a number of Agents active in a randomly chosen layer, whose is asked how they will vote.

Basically that is how a poll works, since a researcher can only access a limited number of layers, the knowledge of the distribution of the real voting intention is limited, as we previously demonstrated.

In our trivial example in 5.51 we show that the knowledge of the poll in two different layers, shown in 5.50, can give two really different results. Our idea is that this difference is at the root of many misprediction operated by polls in the recent years.

If we have different vote distribution in layer, that are not compatible with the voting distribution of the whole network, we may not get a representative sample of what will be the outcome of a vote.

In the simple experiment 5.52, generated with all parameters randomly set as previously discussed, we randomly chose a single layer and interview one hundred Agents, then we measured the difference with the vote expressed in the whole network.

Considering that the network is randomly generated and the vote is in the range $(-1, 1)$ we see a small but steady difference, in the order of $10^{-2} \sim 10^{-3}$, from the simple poll we performed and the real distribution of the vote.

This results may be due to multiple effect, we suppose that it is also due to the multi-layer nature of our network, in which each layer has a defined distri-

Figure 5.50: Voting distribution on two layers

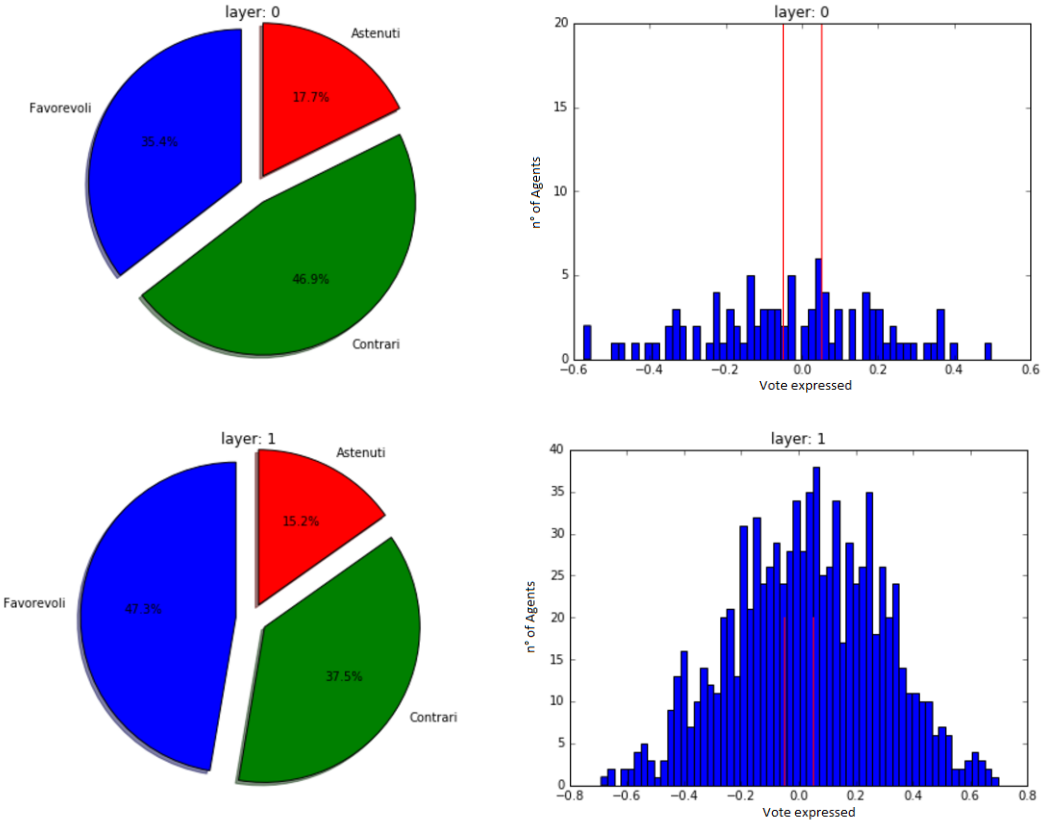


Figure 5.51: Simple poll on two layers

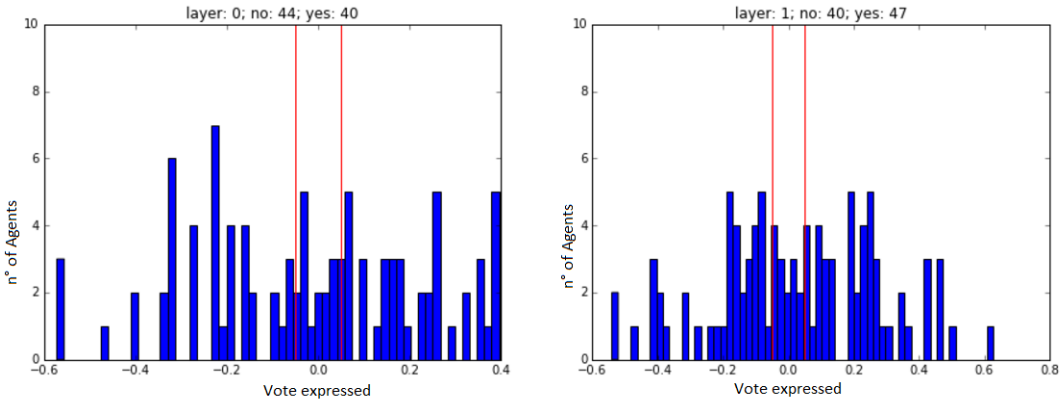
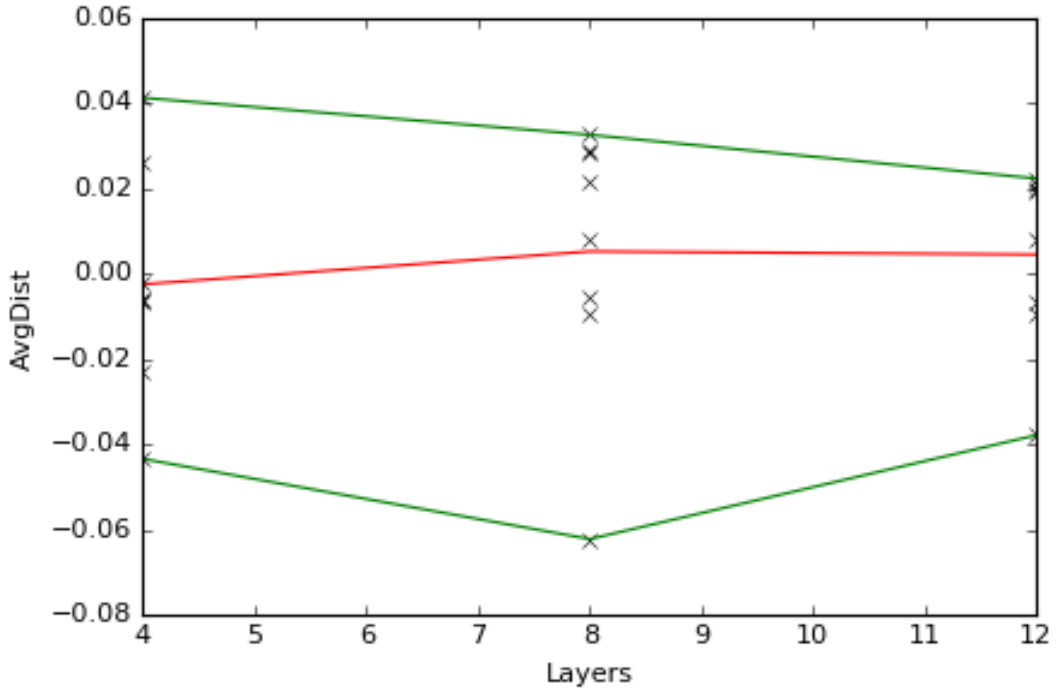
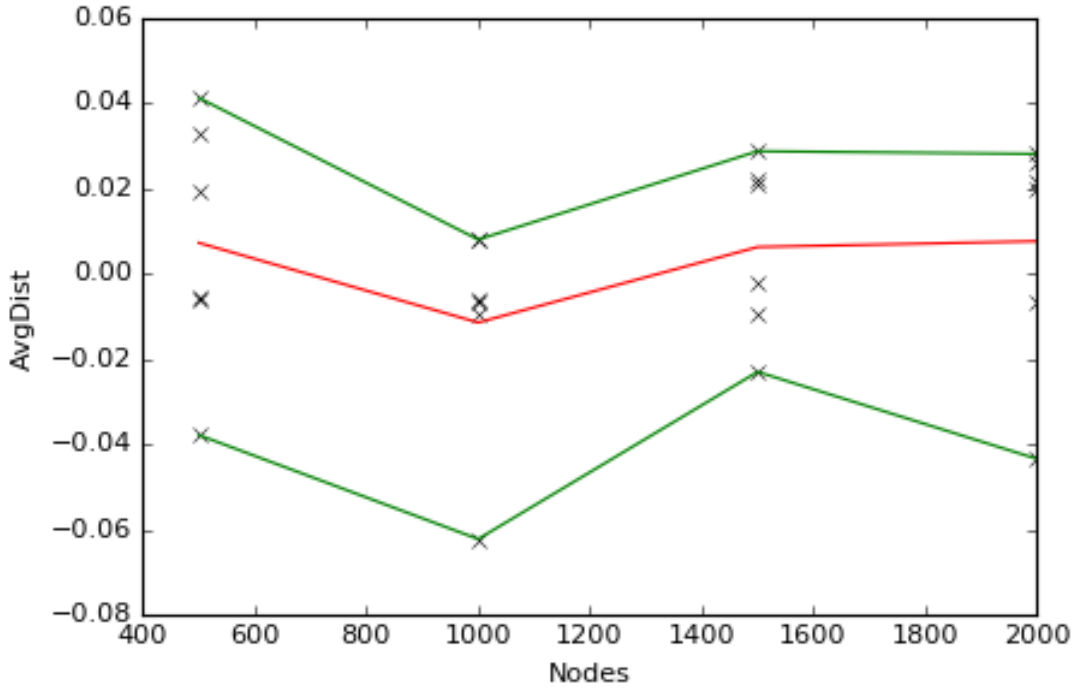


Figure 5.52: Average vote difference of poll

(a) vs layers



(b) vs nodes



bution of vote not corresponding with the distribution in the whole network.

5.4.2 Exploring polls

Portion of interviewed

We tried to investigate how the number of respondents affect the coherence between the result obtained on layers and the result over the whole network. Having more than one layer, the difference from the average over the layers and the whole distribution is shown separately form the difference of each layer and the whole distribution.

The following experiment has been done with three layers randomly extracted from where a variable portion of Agents is interviewed to simulate a poll. All other parameters are randomly chosen according to previous discussion.

In figure 5.53 and 5.54 are shown the average of the data, the maximum and the minimum. We do not see a great difference with the previous experiment, anyway having a free parameter more can be noticed that, as expected, increasing the portion of Agent to interview in our poll reduces the variation of the data. Having a larger sample makes the fluctuation between layers and the whole network smaller.

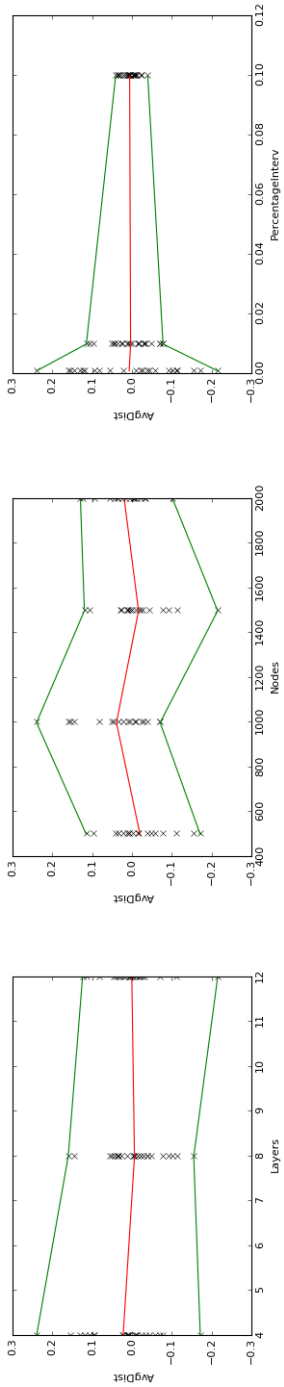
Conforming and rebelling

We tried, in a very simple way, to simulate how an uncertain Agent may choose how to vote. In our interpretation if the vote of an Agent is in a boundary close to zero, she is considered abstained from the vote.

Two algorithm are tried: in the first the abstained Agent conforms with the vote of her neighbors in a given layer, in the second she rebels and vote the opposite of her neighbors in a given layer.

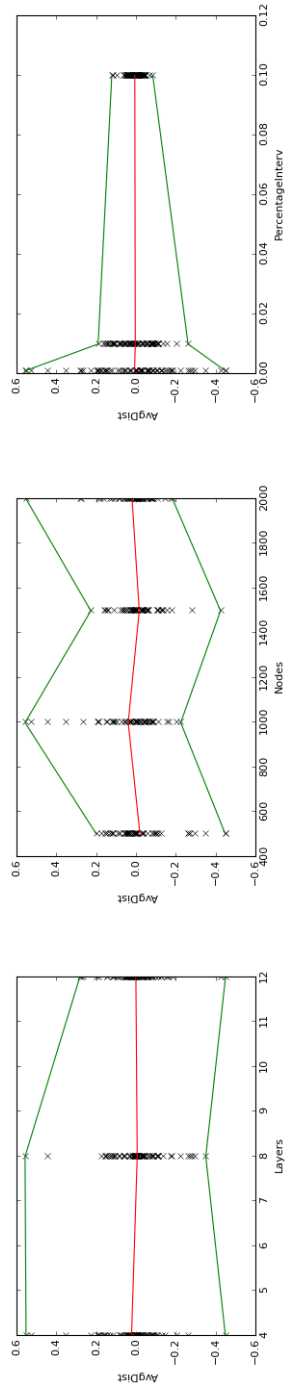
Results, shown in 5.55 and 5.56, are given in the form of the difference of vote from the distribution in a given layer after the use of the algorithms and the whole network without any alteration.

A simulation with greater computational power is clearly needed to deeper



(a) vs layers (b) vs nodes (c) vs portion of interviewed

Figure 5.53: Average vote difference, averaged over layers



(a) vs layers (b) vs nodes (c) vs portion of interviewed

Figure 5.54: Average vote difference of each layer

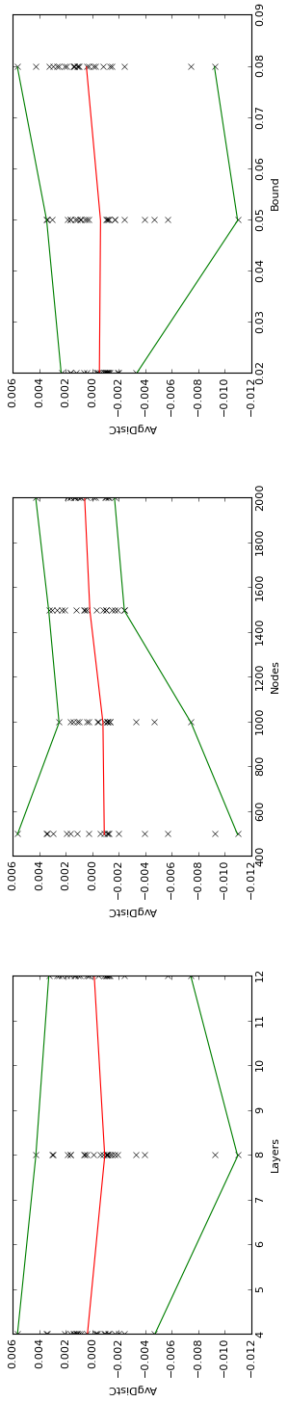


Figure 5.55: Conforming algorithm

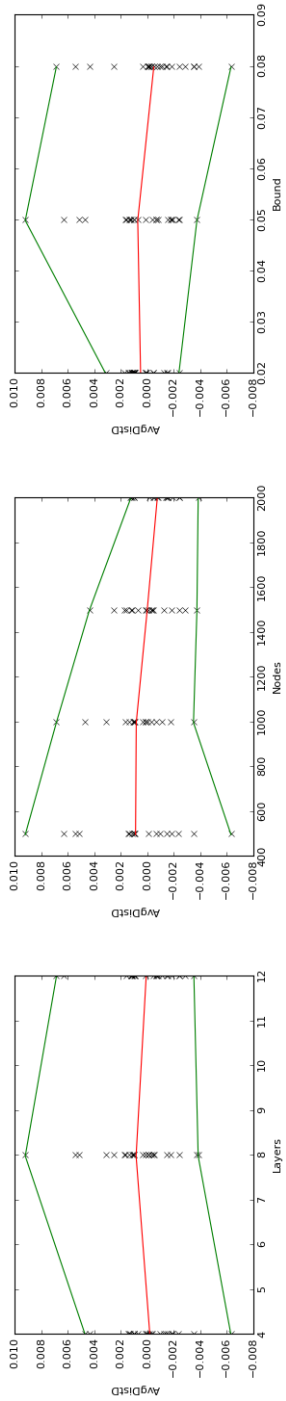


Figure 5.56: Rebelling algorithm

investigate what may happen in a bigger network, nevertheless we start to see some interesting effect.

The two algorithm, although having near-zero results, express two opposing tendencies. What is probably happening here is that in the two situation the Agents that have to choose how to vote, based on the conformation or rebellion, are in each layer taking opposing decision.

This means that each layer is expressing a peculiar situation, different from what is found in the network as a whole, from which the Agent tries to differ or conform, in opposing direction but consistently.

Needing further investigation, this simple experiment is anyway a hint of the role that layers have in a multi-layer network and how they can disrupt our comprehension of the whole network.

Chapter 6

Concluding Remarks

In this work, we introduced some basic concepts of multilayer networks in chapter 2 with definitions and measures, then, in chapter 3, we introduced polls, their basic implementation and several problems, and after, in chapter 4, we defined Agent-Based Models as our paradigm in simulation.

From chapter 5, we started the implementation of our simulation and explored all subsequent steps to be taken into account.

Our multilayer network has been generated, coherently with our Agent-Based paradigm, in order to reproduce basic and advanced features commonly accepted for social network. Parameters have been explored and an algorithm, based on links created by Agents examining common beliefs, and inspired by Axelrod works, has been chosen to generate our network.

Methods of diffusion of ideas have been implemented, taking inspiration from the Bounded Confidence Model, and studied on the network with several innovative measures.

The act of voting used for the simulation has been chosen among many, and implemented in the model in order to mimic the process of interpreting and expressing an opinion.

What has been found is that, under previous constrictions, the distribution of voting, and therefore beliefs, in a multilayer network that retains social characteristic, has a peculiar behavior.

In particular, the distribution of vote on a single, randomly chosen, layer

does not reflect the distribution on the projection network, therefore the study of such measure on a single layer is not representative of the same measure on the whole network.

The interpretation we propose is that, when performing a social investigation with the use of polls, part of the error in the retrieval of real ideas diffusion is due to the fact that researchers truly access only one, or few, layers of which our society is composed.

Simple techniques of polling, intended in our simulation as independent extractions from a population, have been implemented and observed.

Our work may explain that we further need to develop our research techniques in the field of social science. Simple polls can not be held reliable enough to allow a fulfilling comprehension of the social structure and ideas diffusion of a complex social network that is our society.

In the appendix, chapter 7, all coding tools specifically developed for the study and creation of our simulation are collected for in-depth look.

6.1 Future development

Further developments of this work may include different directions of exploration.

Surely, a greater computational power is needed to reach sizes of simulation comparable with those found in real-life situation.

A better study of communities in the multilayer network may be interesting, along with the implementation of weighted links that will make information diffusion somehow asymmetrical and more realistic.

Deepening the comparison with sociological studies and the research of realistic data, often retained as an asset by companies, is a necessary further step to be taken in account in order to validate the suitability of the model.

Chapter 7

Appendix

Here we see, in a more detailed way, the implementation of the simulation, discussing the problems encountered and solved throughout the research.

The whole code is written in **Python 3** programming language.

7.1 Node.py

Each node has two vectors of floating numbers of variable length, for simplicity accepted values are $[0, 1)$, the first vector being associated with variable beliefs and the second one with fixed beliefs. Each node also store an identificative number and the time at which she was created. Node.py has a wide number of getter to retrieve all sort of useful data, a set of function has been implemented to keep record of friends added and removed or influence received.

```
class Node:

    def __init__(self, num, FixBel, VarBel, Gull, time):

        self.FixBel = FixBel[:]
        self.nFixBel = len(self.FixBel)
        self.VarBel = VarBel[:]
        self.nVarBel = len(self.VarBel)
        self.num = num
        self.born = time
        self.activeLayer=[]
        self.Vote = 0
        self.Gullible = Gull #magnitude of influence

        '''these two just to track evolution'''
        self.FriendHist = {}
        self.InfluHist = {}
```

```
'''return value belief'''
def getOneVarB(self, pBel): #get value in pBel position
    return self.VarBel[pBel]
def getOneFixB(self, pBel):
    return self.FixBel[pBel]
def getMoreVarB(self, n): #n is a list of position, return belief in position
    if np.sort(n)[-1] >= self.nVarBel:
        return -1
    temp = []
    for i in n:
        temp.append(self.VarBel[i])
    return temp[:]
def getMoreFixB(self,n):
    if np.sort(n)[-1] >= self.nFixBel:
        return -1
    temp = []
    for i in n:
        temp.append(self.FixBel[i])
    return temp[:]

'''various getter'''''
def getFixB(self): #various getter to look in to the agent
    return self.FixBel[:]
def getLenFixB(self):
    return self.nFixBel
def getVarB(self):
    return self.VarBel[:]
def getLenVarB(self):
    return self.nVarBel
def getNum(self):
    return self.num
def getBorn(self):
    return self.born
def getOld(self, time):
    return time-self.born
def getFriH(self):
    return dict(self.FriendHist)
def getInfH(self):
    return dict(self.InfluHist)
def getActiveLayer(self):
    return self.activeLayer[:]
def getGull(self):
    return self.Gullible #[:]
def getVote(self):
    return self.Vote

'''voting algorithm''' #essentially a dot product normalized

def doVoteC(self, issue):
```

```
        if len(issue)==(self.nFixBel+self.nVarBel):
            self.Vote = ((self.getFixVote(issue[:self.nFixBel])+self.getVarVote(issue[self.nVarBel:]))/2)
            #return(sum([(i*j)**(0.5) for (i, j)
             in zip(issue, self.FixBel+self.VarBel)])/(self.nFixBel+self.nVarBel))
        else:
            return(-1)
    def doVoteA(self, issue):
        import numpy as np
        if len(issue)==(self.nFixBel+self.nVarBel):
            tT=list((self.VarBel+self.FixBel)/(np.linalg.norm(self.VarBel+self.FixBel)))
            tI=issue/np.linalg.norm(issue)
            self.Vote = np.inner(tT,tI)
        else:
            return(-1)
    def doVoteB(self):
        import numpy as np
        self.Vote = np.mean(self.VarBel+self.FixBel)
    def doVarVote(self,issue):
        if len(issue)==(self.nVarBel):
            self.Vote = (sum([(i*j)**(0.5) for (i, j) in zip(issue, self.VarBel)])/self.nVarBel)
        else:
            return(-1)
    def doVarVoteA(self,issue):
        import numpy as np
        if len(issue)==(self.nVarBel):
            tI=issue/np.linalg.norm(issue)
            tV=self.VarBel/np.linalg.norm(self.VarBel)
            self.Vote = np.inner(tV,tI)
        else:
            return -1
    def doVarVoteB(self):
        import numpy as np
        self.Vote = np.mean(self.VarBel)
    def doFixVote(self,issue):
        if len(issue)==self.nFixBel:
            self.Vote = (sum([(i*j)**(0.5) for (i, j) in zip(issue, self.FixBel)])/self.nFixBel)
        else:
            return(-1)
    def doFixVoteA(self,issue):
        import numpy as np
        if len(issue)==(self.nFixBel):
            tI=issue/np.linalg.norm(issue)
            tV=self.FixBel/np.linalg.norm(self.FixBel)
            self.Vote = np.inner(tV,tI)
        else:
            return(-1)
    def doFixVoteB(self):
        import numpy as np
        self.Vote = np.mean(self.FixBel)
```

```
'''change the val in pos pBel of variable belief'''
def setOneVarB(self, pBel, vBel):
    self.VarBel[pBel] = vBel
'''setter to change variable and fixed belief, not to be used'''
def setFixB(self, FixB):
    self.FixBel = FixB[:]
    self.nFixBel = len(self.FixBel)
def setVarB(self, VarB):
    self.VarBel = VarB[:]
    self.nVarBel = len(self.VarBel)

def setNewLayer(self,layer):
    if layer not in self.activeLayer:
        self.activeLayer.append(layer)
def setVote(self, extvote):
    self.Vote = extvote
def setGull(self, Gull):
    self.Gullible = Gull #[:]

''' from here on is just to track evolution of the node'''''
def getFriStep(self, time): #get the dict of friend at a certain time
    if time in self.FriendHist:
        return dict(self.FriendHist[time])
    else:
        return -1
def getInfStep(self, time): #get the dict of influence at a certain time
    if time in self.InfluHist:
        return dict(self.InfluHist[time])
    else:
        return -1

def recNewF(self, layer, nFri, time): #keep track of a new friend added
    if not time in self.FriendHist: #it's a dict
        self.aggF(time) #copy the dict
    if not layer in self.FriendHist[time]:
        self.FriendHist[time][layer] = [] #it's a list
    self.FriendHist[time][layer].append(nFri)

def aggF(self, time): #keep list of friend up to date with clock but without changes
    if not time-1 in self.FriendHist:
        self.FriendHist[time-1] = {}
    self.FriendHist[time] = dict(self.FriendHist[time-1])

def delOldF(self, layer, nFri, time): #remove a friend
    if time in self.FriendHist:
        if not layer in self.FriendHist[time]:
            return -1 #if layer not present raise error
        if not nFri in self.FriendHist[time][layer]:
            return -1 #if nFri not present rise error
        self.FriendHist[time][layer].remove(nFri) #else remove selected from list
```

```
        else:
            if time-1 in self.FriendHist:
                self.FriendHist[time] = dict(self.FriendHist[time-1])
                 # tries to rebuilt actual FriendHist[time] dict
                if not layer in self.FriendHist[time]:
                    return -1 #if layer not present raise error
                if not nFri in self.FriendHist[time][layer]:
                    return -1 # if nFri not present, rise error
                self.FriendHist[time][layer].remove(nFri)
            else:
                return -1

    def recNewI(self, nFri, time, pBel, vBel, layer): #vBel it's a value in pos pBel in vector VarBel
        if not time in self.InfluHist: #InfluHist is a dict of dict
            self.InfluHist[time] = {}
        if not layer in self.InfluHist[time]:
            self.InfluHist[time][layer] = []
        self.InfluHist[time][layer].append([pBel,vBel, nFri]) #InfluHist[time][nFri] is a list
```

7.1.1 doVoteA

Here is the algorithm for voting on a issue with the tool numpy.inner() as a function of Node.py .

```
def doVoteA(self, issue):
        import numpy as np
        if len(issue)==(self.nFixBel+self.nVarBel):
            tT=list((self.VarBel+self.FixBel)/(np.linalg.norm(self.VarBel+self.FixBel)))
            tI=issue/np.linalg.norm(issue)
            self.Vote = np.inner(tT,tI)
        else:
            return(-1)
```

7.2 NetworkTool.py

Here a series of generic tools for analysis and modification of the network.

```
def NodeAlike(NodeA, NodeB, nF, dF, nV, dV):
#nF fix belief in distance pF, nV var belief in common in distance pV
    import numpy as np
    alike = []
    posF = []
    posV = []
    '''similarity in fixed belief'''
    if not NodeA.getLenFixB() == NodeB.getLenFixB(): #chek if node have same list dimension
        return -1
```

```
    if nF > NodeA.getLenFixB():
        return -1
    if not NodeA.getLenFixB() == 0: #if list empty skip
        k = 0
        while k < nF:
            temp = np.random.randint(0,NodeA.getLenFixB())
            if not temp in posF: #this avoid to draw twice the same position
                posF.append(temp)
                k = k+1
    for i in posF:
        if abs(NodeA.getOneFixB(i) - NodeB.getOneFixB(i)) <= dF:
            alike.append(1)
        else:
            alike.append(0) #if too different add 0 in alike
    '''similarity in variable belief'''
    if not NodeA.getLenVarB() == NodeB.getLenVarB(): #check if node have same list dimension
        return -1
    if nV > NodeA.getLenVarB():
        return -1
    if not NodeA.getLenVarB() == 0: #if list empty skip
        k = 0
        while k < nV:
            temp = np.random.randint(0,NodeA.getLenVarB())
            if not temp in posV: #this avoid to draw twice the same position
                posV.append(temp)
                k = k+1
    for i in posV:
        if abs(NodeA.getOneVarB(i) - NodeB.getOneVarB(i)) <= dV:
            alike.append(1)
        else:
            alike.append(0) #if too different add 0 in alike
    '''if at least one parameter too different return 0'''
    if 0 in alike:
        return 0
    else:
        return 1 #if no problem return 1

def NodePosAlike(NodeA, NodeB, listPF, dF, listPV, dV):
#listPF and listPV are list of position to look at in belief for similarity
    import numpy as np
    alike = []
    '''similarity in fixed belief'''
    if not NodeA.getLenFixB() == NodeB.getLenFixB(): #chek if node have same list dimension
        return -1
    if not NodeA.getLenFixB() == 0: #if list empty skip
        for i in listPF:
            if abs(NodeA.getOneFixB(i) - NodeB.getOneFixB(i)) <= dF:
                alike.append(1)
            else:
                alike.append(0) #if too different add 0 in alike
```

```
    '''similarity in variable belief'''
    if not NodeA.getLenVarB() == NodeB.getLenVarB(): #chek if node have same list dimension
        return -1
    if not NodeA.getLenVarB() == 0: #if list empty skip
        for i in listPV:
            if abs(NodeA.getOneVarB(i) - NodeB.getOneVarB(i)) <= dV:
                alike.append(1)
            else:
                alike.append(0) #if too different add 0 in alike
    '''if at least one parameter too different return 0'''
    #print(alike) option for verbose
    if 0 in alike:
        return 0
    else:
        return 1 #if no problem return 1


def NodeInfluRand(NodeA, NodeB, nV, pV): #A influence B on nV belief with prob pV
    import numpy as np
    posV = []
    if not NodeA.getLenVarB() == NodeB.getLenVarB():
        return -1
    if nV >= NodeA.getLenVarB():
        return -1
    for i in range(nV):
        posV.append(np.random.randint(0,NodeA.getLenFixB()))
    for i in posV:
        if np.random.random() < pV: #that's a prob
            NodeB.setOneVarB(NodeA.getOneVarB(i))

def NodeBoundTwins(NodeA, NodeB, bpos, pV, dV, rate, time, layer, r=0):
#node, node, pos of belief to consider, random prob, distance of belief, change rate
#this simply process both nodes
    import numpy as np
    if not NodeA.getLenVarB() == NodeB.getLenVarB():
        return -1
    pos = np.random.choice(bpos) #shuffle list of belief
    if abs(NodeA.getOneVarB(pos)-NodeB.getOneVarB(pos)) < dV: #check distance between beliefs
        if np.random.random()<pV: #check random
            NodeB.setOneVarB(pos, NodeB.getOneVarB(pos)+rate*(NodeA.getOneVarB(pos)-NodeB.getOneVarB(pos)))
            #correct value of belief
            NodeA.setOneVarB(pos, NodeA.getOneVarB(pos)+rate*(NodeB.getOneVarB(pos)-NodeA.getOneVarB(pos)))
            '''to keep track'''
            if r == 1:
                NodeB.recNewI(NodeA.getNum(), time, pos, NodeA.getOneVarB(pos), layer)
                NodeA.recNewI(NodeA.getNum(), time, pos, NodeB.getOneVarB(pos), layer)
    del pos

def NodeBoundDrift(NodeA, NodeB, bpos, pV, dV, rate, time, layer, r=0):
#experimental to see different way to converge
```

```
    import numpy as np
    if not NodeA.getLenVarB() == NodeB.getLenVarB():
        return -1
    pos = np.random.choice(bpos) #shuffle list of belief
    if abs(NodeA.getOneVarB(pos)-NodeB.getOneVarB(pos)) < dV: #check distance between beliefs
        if np.random.random()<pV: #check random
            NodeB.setOneVarB(pos, NodeB.getOneVarB(pos)+rate*(NodeA.getOneVarB(pos)-NodeB.getOneVarB(pos)))
            #correct value of belief
            '''to keep track'''
            if r == 1:
                NodeB.recNewI(NodeA.getNum(), time, pos, NodeA.getOneVarB(pos), layer)
    else:
        if np.random.random()<0.01*pV: #check random
            NodeB.setOneVarB(pos, NodeB.getOneVarB(pos)-rate*(NodeA.getOneVarB(pos)+NodeB.getOneVarB(pos)))
            #correct value of belief
            '''to keep track'''
            if r == 1:
                NodeB.recNewI(NodeA.getNum(), time, pos, -1*NodeA.getOneVarB(pos), layer)
    del pos

def CheckEdge(MG,Node,Candidate,Layer):
    flag = 0
    tG = ExtractLayer(MG,Layer)
    if Candidate==Node: #check two nodes aren't the same
        flag = 1
    if (Candidate, Node) in tG.edges(): #if edges don't already in layer
        flag = 1
    if (Node, Candidate) in tG.edges(): #if edges don't already in layer
        flag = 1
    tG.clear()
    return flag

def ExtractLayer(MG, Layer): #return a single layer
    import networkx as nx
    temp = nx.Graph()
    for i in MG.edges(data='layer'):
        if i[2]==Layer:
            temp.add_edge(i[0],i[1])
    return temp
    del temp

def DrawLayer(MG, Nodes, Pos, nLayer): #draw representation of each layer
    import networkx as nx
    import matplotlib.pyplot as plt
    fig = plt.figure(figsize=(15,nLayer*3))
    axt=[]
    color = ['r','g','b','c','m','y', 'r','g','b','c','m','y', 'r','g','b','c','m','y']
    Label = {}
    for i in range(Nodes):
        Label[MG.nodes()[i]]=i
```

```
    mxt = fig.add_subplot(nLayer+1,1,1)
    mxt.set_title('complete graph')
    nx.draw(MG,ax=mxt, pos=Pos, labels=Label, node_size = 100)
    for i in range(nLayer):
        axt.append(fig.add_subplot(nLayer+1,1,i+2))
        axt[i].set_title(str('Layer '+str(i)))
        nx.draw(ExtractLayer(MG,i),ax=axt[i], edge_color=color[i], pos=Pos, labels=Label, node_size = 150)
    return plt.show()

def DrawLayerVote(MG, Nodes, Pos, nLayer, issue):
    import networkx as nx
    import matplotlib.pyplot as plt
    col = []
    for i in MG.nodes():
        col.append(i.getVote())
    fig = plt.figure(figsize=(15,nLayer*3))
    axt=[]
    color = ['r','g','b','c','m','y', 'r','g','b','c','m','y', 'r','g','b','c','m','y']
    Label = {}
    for i in range(Nodes):
        Label[MG.nodes()[i]]=i
    mxt = fig.add_subplot(nLayer+1,1,1)
    mxt.set_title('complete graph')
    nx.draw(MG,ax=mxt, pos=Pos, labels=Label, node_size = 100, node_color = col, cmap = 'bwr')
    for i in range(nLayer):
        col=[]
        G=ExtractLayer(MG,i)
        for j in G.nodes():
            col.append(j.getVote())
        axt.append(fig.add_subplot(nLayer+1,1,i+2))
        axt[i].set_title(str('Layer '+str(i)))
        nx.draw(G,ax=axt[i], edge_color=color[i], pos=Pos, labels=Label, node_size = 150, node_color=col,
         cmap='bwr')
    return plt.show()

def PlotLayer(MG,nLayer): #plot degree distribution for each layer
    import networkx as nx
    import matplotlib.pyplot as plt
    import numpy as np
    from scipy import stats
    from sklearn.metrics import mean_squared_error
    from math import log
    fig = plt.figure(figsize=(10,nLayer*6.5))
    color = ['r','g','b','c','m','y', 'r','g','b','c','m','y', 'r','g','b','c','m','y']
    degree_sequence=sorted(nx.degree(MG).values(),reverse=True)
    log_degree_sequence=[]
    for i in degree_sequence:
        log_degree_sequence.append(float(i))
    ax = fig.add_subplot(nLayer+1,1,1)
    ax.set_title("Degree rank Complete Graph")
```

```
    x=[]
    for i in range(len(degree_sequence)):
        x.append(float(i+1))
    fit = np.polyfit(x, log_degree_sequence,1)
    fit_fn = np.poly1d(fit)
    print ("Complete Graph; entry:",len(x), "; mean squared error to log-fit:", mean_squared_error(fit_fn(x),
     degree_sequence))
    plt.loglog(degree_sequence,'b-',marker='o')
    ax.set_ylabel("degree")
    ax.set_xlabel("rank")
    for i in range(nLayer):
        degree_sequence=sorted(nx.degree(ExtractLayer(MG,i)).values(),reverse=True)
        log_degree_sequence=[]
        for i in degree_sequence:
            log_degree_sequence.append(float(i))
        kurt = stats.kurtosis(degree_sequence)
        ax=fig.add_subplot(nLayer+1,1,i+2)
        ax.set_title(str('Layer '+str(i)+', Kurtosis: '+str(round(kurt,2))))
        plt.loglog(degree_sequence,color[i],marker='o')
        x=[]
        for i in range(len(degree_sequence)):
            x.append(float(i+1))
        fit = np.polyfit(x,log_degree_sequence,1)
        fit_fn = np.poly1d(fit)
        print ("Layer:", i,"; entry:",len(x), "; mean squared error to log-fit:", mean_squared_error(fit_fn(x),
         degree_sequence))

        plt.ylabel("degree")
        plt.xlabel("rank")

    return fig.show()

def BaseAnalisys(MG,nLayer):
    import networkx as nx
    import numpy as np
    output=[]
    print('complete assortativity ',nx.degree_assortativity_coefficient(MG))
    output.append([MG.number_of_nodes(),nx.degree_assortativity_coefficient(MG),
     np.average(list(nx.degree(MG).values()))])
    print()
    for i in range(nLayer):
        G=ExtractLayer(MG,i)
        print('Layer',i)
        print('assortativity ',nx.degree_assortativity_coefficient(G))
        print('clustering ',nx.average_clustering(G))
        print('averagedegree ',np.average(list(nx.degree(G).values())))
        print('nodes ', G.number_of_nodes())
        print()
        output.append([i,G.number_of_nodes(),nx.degree_assortativity_coefficient(G),
         nx.average_clustering(G),np.average(list(nx.degree(G).values()))])
```

```
    return(output)

def LayerAnalisys(MG, Layer):
    import networkx as nx
    import numpy as np
    temp = []
    temp.append(Layer)
    if ExtractLayer(MG,Layer).number_of_edges() != 0: #trying to avoid error for empty Layer
        temp.append(nx.degree_pearson_correlation_coefficient(ExtractLayer(MG,Layer)))
                #temp.append(nx.degree_assortativity_coefficient(ExtractLayer(MG,Layer)))
        temp.append(nx.average_clustering(ExtractLayer(MG,Layer)))
        temp.append(np.average(list(nx.degree(ExtractLayer(MG,Layer)).values())))
    else:
        temp.append(float('nan'))
        temp.append(float('nan'))
        temp.append(float('nan'))
        print('nan error! layer: ', Layer)
    return temp

def NetAnalisys(MG):
    import networkx as nx
    import numpy as np
    temp = []
    temp.append(nx.degree_assortativity_coefficient(MG))
    temp.append(np.average(list(nx.degree(MG).values())))
    return temp

def ShowDoubleEdges(MG):
    import networkx as nx
    for i in MG.edges(data='layer'):
        for j in range(Layer):
            if (i[0],i[1]) in ExtractLayer(MG,j).edges() or (i[1],i[0]) in ExtractLayer(MG,j).edges():
                if i[2]!=j:
                    print('layer',i[2],'and ',j,' edge ',i)

def GullDistr(MG, nLayer):
    #analysis of gullible distribution, should be uniform
    import matplotlib.plot as plt
    avg=[]
    for j in range(nLayer):
        gul=[]
        for i in ExtractLayer(MG, j).nodes():
            gul.append(i.getGull())
        avg.append(np.mean(gul))
        plt.hist(gul, 10)
    print('Average: ', np.mean(avg))
    return plt.show()

def DegCorrAll(MG, bpos, nLayer): #calculate correlation of degree centrality for nodes in different layer
    import numpy as np
```

```
    import networkx as nx
    val=[]
    val.append(nx.degree_centrality(MG))
    for i in range(nLayer):
        G=ExtractLayer(MG, i)
        val.append(nx.degree_centrality(G))
    print('Number of layers: ', len(val))
    matr=[]
    matr.append([])
    for n in MG.nodes():
        matr[0].append(val[0][n])
    for l in range(nLayer):
        matr.append([])
        l=l+1
        for n in MG.nodes():
            if n in val[l]:
                matr[l].append(val[l][n])
            else:
                matr[l].append(0)
    #print(shape(matr))
    pears=[]
    print('Here a list of couples of layers with positive Pearson coefficient regarding degree centrality')
    for i in range(len(val)):
        for j in range(len(val)):
            if i!=0 and j!=0 and i!=j: #calc the pearson coeff of deg_centr only between different layers
                pears.append(np.corrcoef(matr[i],matr[j])[0,1])
                if (np.corrcoef(matr[i],matr[j])[0,1])>0:
                    print("check:",i,":",j," bpos",bpos[i],":",bpos[j]," coeff",
                     np.corrcoef(matr[i],matr[j])[0,1])
    print()
    counter=sum(1 if x>0 else 0 for x in pears) #how many pos coeff
    print('Number of Pearson coefficient calculated: ',len(pears))
    print('Number of positive coefficient: ', counter)

def GephiSearch(MG, Label):
    import matplotlib.pyplot as plt
    for nn in MG.nodes():
        if MG.node[nn]['Label']==Label:
            print(MG.node[nn])
            zz=nn
            print("found")
    print('Active layers: ', zz.getActiveLayer())
    print('Total neighbors: ', len(MG.neighbors(zz)))
    print('Neighbors per layer:')
    for i in zz.getActiveLayer():
        print('Neigh. ',len(ExtractLayer(MG, i).neighbors(zz)), 'layer ',i)
    print('Vote: ', zz.getVote(), ' Gullible: ',zz.getGull())
    print('Layer in wich was influenced: ')
    ix=[]
    for i in zz.InfluHist:
```

```
        for j in zz.InfluHist[i]:
            ix.append(j)
    plt.hist(ix, 7)
    print(ix)
    print('List of influence: ', zz.InfluHist)
    return plt.show()
```

7.2.1 NodeBoundOne

Here the algorithm that implement the Bounded Confidence Model

```
def NodeBoundOne(NodeA, NodeB, bpos, pV, dV, rate, time, layer, r=0):
#node, node, pos of belief to consider, random prob, distance of belief, change rate
    import numpy as np
    if not NodeA.getLenVarB() == NodeB.getLenVarB():
        return -1
    pos = np.random.choice(bpos) #shuffle list of belief
    if abs(NodeA.getOneVarB(pos)-NodeB.getOneVarB(pos)) < dV: #check distance between beliefs
        if np.random.random()<pV: #check random
            NodeB.setOneVarB(pos, NodeB.getOneVarB(pos)+rate*(NodeA.getOneVarB(pos)-NodeB.getOneVarB(pos)))
            #correct value of belief
            '''to keep track'''
            if r == 1:
                NodeB.recNewI(NodeA.getNum(), time, pos, NodeA.getOneVarB(pos), layer)
    del pos
```

7.2.2 EvolveInfluR

This algorithm performs a series of iteration of the diffusion process of ideas among the Agents in the network

```
def EvolveInfluR(MG, iteration, cbpos, pV, dV, TheTime, nLayer): #evolve and influence the beliefs
    import numpy as np
    print('Evolving for ',iteration)
    G=[]
    TheTime = TheTime*100
    for lay in range(nLayer):
        G.append(ExtractLayer(MG, lay))
    for kk in range(iteration):
        if kk - 500*(kk//500) == 0 :
            #print(kk, TheTime)
            TheTime = TheTime+1
        for node in MG.nodes():
            try:
                lay = np.random.choice(node.getActiveLayer()) #choose a layer in wich the node is active
                g = G[lay]
                candidate = np.random.choice(g.neighbors(node)) #choose a random nighbor
                NodeBoundOne(candidate, node, cbpos[lay], pV, dV, node.getGull(), TheTime, lay)
```

```
                NodeBoundOne(node, candidate, cbpos[lay], pV, dV, candidate.getGull(), TheTime, lay)
            except:
                pass
    del G
```

7.3 GenerativeTool.py

This file contains all the tools made for the creation of the network.

```
def AttachCBRandomLayer(MG, Node, nFriends, nLayer, llPF, listDF, llPV, listDV, time):
#listlistPF is a list of list
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    temp = []
    for n in MG.nodes():
        temp.append(n)
    i=0
    while len(temp) > 0 and i < nFriends: #if still there are nodes and not enough friends
        np.random.shuffle(temp) #shuffle list of friend
        Candidate = temp.pop() #choose random node removing from temp list
        tlay = np.random.randint(nLayer)
        tdF = listDF[tlay] #from the list extract the needed one
        tdV = listDV[tlay]
        listPF = llPF[tlay]
        listPV = llPV[tlay]
        if nt.CheckEdge(MG, Node, Candidate, tlay) == 0:
            if nt.NodePosAlike(Node, Candidate, listPF, tdF, listV, dV) == 1:
                MG.add_edge(Node, Candidate, layer = tlay) #if positive add edge in layer
                Node.setNewLayer(tlay)
                Candidate.setNewLayer(tlay)
                '''to keep track'''
                Node.aggF(time)
                Candidate.aggF(time)
                Node.recNewF(tlay, Candidate.getNum(), time)
                Candidate.recNewF(tlay, Node.getNum(), time)
                i=i+1

def AttachCBAnyLayer(MG, NodeA, nFriends, llPF, listDF, llPV, listDV, time): #listlistPF is a list of list
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    i=0
    temp = MG.nodes()[:]
    while i < nFriends and len(temp) > 0 : #if still there are nodes and not enough friends
        Candidate = temp.pop() #choose random node removing from temp list
        if len(Candidate.getActiveLayer()) != 0:
            tlay = np.random.choice(Candidate.getActiveLayer()) #layer in wich the node is active
```

```
            if nt.CheckEdge(MG, NodeA, Candidate, tlay) == 0:
                tdF = listDF[tlay] #from the list extract the needed one
                tdV = listDV[tlay]
                listPF = llPF[tlay]
                listPV = llPV[tlay]
                if nt.NodePosAlike(NodeA, Candidate, listPF, tdF, listPV, tdV) == 1:
                    MG.add_edge(NodeA, Candidate, layer = tlay) #if positive add edge in layer
                    NodeA.setNewLayer(tlay)
                    Candidate.setNewLayer(tlay)
                    '''to keep track'''
                    NodeA.aggF(time)
                    Candidate.aggF(time)
                    NodeA.recNewF(tlay, Candidate.getNum(), time)
                    Candidate.recNewF(tlay, NodeA.getNum(), time)
                    i=i+1
    del temp

def AttachFFAnyLayer(MG, NodeA, nFriends, llPF, listDF, llPV, listDV, time):
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    temp = []
    for m in nx.all_neighbors(MG, NodeA):
        for n in nx.all_neighbors(MG, m): #create a list of neighbors of neighbors
            temp.append(n)
    i=0
    while len(temp) > 0 and i < nFriends: #if still there are nodes and not enough friends
        np.random.shuffle(temp) #shuffle list of friend
        Candidate = temp.pop() #choose random node removing from temp list
        if len(Candidate.getActiveLayer()) != 0:
            tlay = np.random.choice(Candidate.getActiveLayer())
            if nt.CheckEdge(MG, NodeA, Candidate, tlay) == 0:
                tdF = listDF[tlay] #from the list extract the needed one
                tdV = listDV[tlay]
                listPF = llPF[tlay]
                listPV = llPV[tlay]
                if nt.NodePosAlike(NodeA, Candidate, listPF, tdF, listPV, tdV) == 1:
                    MG.add_edge(NodeA, Candidate, layer = tlay) #if positive add edge in layer
                    NodeA.setNewLayer(tlay)
                    '''to keep track'''
                    NodeA.aggF(time)
                    Candidate.aggF(time)
                    NodeA.recNewF(tlay, Candidate.getNum(), time)
                    Candidate.recNewF(tlay, NodeA.getNum(), time)
                    i=i+1
    if i < nFriends:
        AttachCBAnyLayer(MG, NodeA, nFriends-i, llPF, listDF, llPV, listDV, time)
    del temp

def  AttachFFRandomLayer(MG, Node, nFriends, nLayer, llPF, listDF, llPV, listDV, time):
```

```
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    temp = []
    for m in nx.all_neighbors(MG, Node):
        for n in nx.all_neighbors(MG, m): #create a list of neighbors of neighbors
            temp.append(n)
    i=0
    while len(temp) > 0 and i < nFriends: #if still there are nodes and not enough friends
        np.random.shuffle(temp) #shuffle list of friend
        Candidate = temp.pop() #choose random node removing from temp list
        tlay = np.random.randint(nLayer)
        listPF = llPF[tlay]
        listPV = llPV[tlay]
        tdF = listDF[tlay]
        tdV = listDV[tlay]
        if nt.CheckEdge(MG, Node, Candidate, tlay) == 0:
            if nt.NodePosAlike(Node, Candidate, listPF, tdF, lisPV, tdV) == 1:
                MG.add_edge(Node, Candidate, layer = tlay) #if positive add edge in layer
                Node.setNewLayer(tlay)
                Candidate.setNewLayer(tlay)
                '''to keep track'''
                Node.aggF(time)
                Candidate.aggF(time)
                Node.recNewF(tlay, Candidate.getNum(), time)
                Candidate.recNewF(tlay, Node.getNum(), time)
                i=i+1
    #if i>0: print(i) #chek if work
    if i < nFriends:
        tlay = np.random.randint(nLayer)
        listPF = llPF[tlay]
        listPV = llPV[tlay]
        tdF = listDF[tlay]
        tdV = listDV[tlay]
        AttachChoosenBelief(MG, Node, nFriends-i, tlay, listPF, tdF, listPV, dV, time)

def RandomLink(MG, Node, nLayer, time):
    import numpy as np
    from NetworkTool import CheckEdge
    flag = 1
    while flag == 1:
        tlay = np.random.randint(nLayer)
        Candidate = np.random.choice(MG.nodes())
        flag = CheckEdge(MG, Node, Candidate, tlay)
    else:
        MG.add_edge(Node, Candidate, layer = tlay)
        Node.setNewLayer(tlay)
        Candidate.setNewLayer(tlay)
        '''to keep track'''
        Node.aggF(time)
```

```
        Candidate.aggF(time)
        Node.recNewF(tlay, Candidate.getNum(), time)
        Candidate.recNewF(tlay, Node.getNum(), time)
```

7.3.1 AttachForBelief

Take as input the network, a chosen node, the number of friend we want him to have, the layer in which the node is supposed to create new links, the number of fixed and variable belief to check and the maximum distance accepted for those value while confronting with another node.

```
def AttachForBelief(MG, Node, nFriends, Layer, nF, dF, nV, dV, time): #For Node in MG try to add nFriend
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    temp = []
    for n in MG.nodes():
        temp.append(n)
    i = 0
    while len(temp) > 0 and i < nFriends: #if still there are nodes and not enough friends
        np.random.shuffle(temp) #shuffle list of friend
        Candidate = temp.pop() #choose random node removing from temp list
        if nt.NodeAlike(Node, Candidate, nF, dF, nV, dV) == 1 and nt.CheckEdge(MG,Node,Candidate,Layer) == 0:
         #check if alike and not connected
            MG.add_edge(Node, Candidate, layer = Layer) #if positive add edge in layer
            Node.setNewLayer(Layer)
            Candidate.setNewLayer(Layer)
            '''to keep track'''
            Node.aggF(time)
            Candidate.aggF(time)
            Node.recNewF(Layer, Candidate.getNum(), time)
            Candidate.recNewF(Layer, Node.getNum(), time)
            i = i+1
```

7.3.2 AttachChoosenBelief

Take as input the network, a chosen node, the number of friend we want him to have, the layer in which the node is supposed to create new links, a list of fixed and variable belief to check and the maximum distance accepted for those value while confronting with another node.

```
def AttachChoosenBelief(MG, Node, nFriends, Layer, listPF, dF, listPV, dV, time): #
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    temp = []
    for n in MG.nodes():
        temp.append(n)
```

```
    i=0
    while len(temp) > 0 and i < nFriends: #if still there are nodes and not enough friends
        np.random.shuffle(temp) #shuffle list of friend
        Candidate = temp.pop() #choose random node removing from temp list
        if nt.CheckEdge(MG, Node, Candidate, Layer) == 0:
            if nt.NodePosAlike(Node, Candidate, listPF, dF, listPV, dV) == 1:
                MG.add_edge(Node, Candidate, layer = Layer) #if positive add edge in layer
                Node.setNewLayer(Layer)
                Candidate.setNewLayer(Layer)
                '''to keep track'''
                Node.aggF(time)
                Candidate.aggF(time)
                Node.recNewF(Layer, Candidate.getNum(), time)
                Candidate.recNewF(Layer, Node.getNum(), time)
                i=i+1
```

7.3.3 AttachFriendFirst

Same as before but search between friends of friends first, then goes for all nodes in the network.

```
def AttachFriendFirst(MG, Node, nFriends, Layer, listPF, dF, listPV, dV, time):
    import numpy as np
    import networkx as nx
    import NetworkTool as nt
    temp = []
    for m in nx.all_neighbors(MG, Node):
        for n in nx.all_neighbors(MG, m): #create a list of neighbors of neighbors
            temp.append(n)
    i=0
    while len(temp) > 0 and i < nFriends: #if still there are nodes and not enough friends
        np.random.shuffle(temp) #shuffle list of friend
        Candidate = temp.pop() #choose random node removing from temp list
        if nt.CheckEdge(MG, Node, Candidate, Layer) == 0:
            if nt.NodePosAlike(Node, Candidate, listPF, dF, listPV, dV) == 1:
                MG.add_edge(Node, Candidate, layer = Layer) #if positive add edge in layer
                Node.setNewLayer(Layer)
                Candidate.setNewLayer(Layer)
                '''to keep track'''
                Node.aggF(time)
                Candidate.aggF(time)
                Node.recNewF(Layer, Candidate.getNum(), time)
                Candidate.recNewF(Layer, Node.getNum(), time)
                i=i+1
    #if i>0: print(i) #chek if work
    if i < nFriends:
        AttachChoosenBelief(MG, Node, nFriends-i, Layer, listPF, dF, listPV, dV, time)
```

7.4 VoteTool

Here a series of tools made for the voting process and analysis.

```
def AllVoteA(MG, issue):
    for nn in MG.nodes():
            nn.doVoteA(issue)

def SocialDesirability(NodeA, MG, md): #if too distant from neighbors lie
    import numpy as np
    vv=[]
    for nn in MG.neighbors(NodeA):
        vv.append(nn.getVote())
    if np.mean(vv)*NodeA.getVote()<0 and abs(np.mean(vv)*NodeA.getVote())>md:
        NodeA.setVote(np.mean(vv))
    del vv

def ConformUncertVote(NodeA, MG, bd, issue): #conform to neighbors if uncertain
    import numpy as np
    if abs(NodeA.getVote()) < bd:
        vv=[]
        for nn in MG.neighbors(NodeA):
            vv.append(nn.getVote())
        NodeA.setVote(np.mean(vv))
        del vv

def DifformUncertVote(NodeA, MG, bd, issue): #difform to neighbors if uncertain
    import numpy as np
    if abs(NodeA.getVote()) < bd:
        vv=[]
        for nn in MG.neighbors(NodeA):
            vv.append(nn.getVote())
        NodeA.setVote(-1*np.mean(vv))
        del vv

def PollDesirability(MG, listLayer,listCall, listMd, issue, bd):
    G=[]
    iteration = len(listLayer)
    for tlay in listLayer:
        G.append(ExtractLayer(MG,tlay))
    Interviews=[]
    for i in iteration:
        for j in range(listCall[i]):
            Interviews[i].append(np.random.choice(G[i].nodes()))
        for n in Interviews[i]:
            SocialDesirability(n, G[i], listMd[i])
    for i in iteration:
        VoteAnalysis(G[i], issue, bd, 1)

def FixDistPrnt(MG, plot):
```

```
    from scipy import stats
    import numpy as np
    import matplotlib.pyplot as plt
    vv = []
    for nn in MG.nodes():
        vv.append(BelFixDistance(nn, MG))
    kur=round(stats.kurtosis(vv),2)
    skew=round(stats.skew(vv),2)
    if plot == 1:
        print('avg dist ', np.mean(vv))
        plt.xlabel('Distance')
        plt.ylabel('# of nodes')
        print('kurtosis',kur)
        print('skewness',skew)
        plt.title('kurtosis: '+str(kur)+', skewness: '+str(skew))
        plt.hist(vv, bins=100)
        plt.show()
    return([np.mean(vv), kur, skew])

def VarDistPrnt(MG, plot):
    from scipy import stats
    import numpy as np
    import matplotlib.pyplot as plt
    vv = []
    for nn in MG.nodes():
        vv.append(BelVarDistance(nn, MG))
    kur=round(stats.kurtosis(vv),2)
    skew=round(stats.skew(vv),2)
    if plot == 1:
        print('avg dist ', np.mean(vv))
        plt.xlabel('Distance')
        plt.ylabel('# of nodes')
        print('kurtosis',kur)
        print('skewness',skew)
        plt.title('kurtosis: '+str(kur)+', skewness: '+str(skew))
        plt.hist(vv, bins=100)
        plt.show()
    return([np.mean(vv), kur, skew])

def VoteAnalysis(MG, issue, bd, do):
    import matplotlib.pyplot as plt
    y=0
    n=0
    a=0
    x=[]
    print(issue)#,' ',len(issue))
    if do == 1:
        for NN in MG.nodes():
            NN.doVoteA(issue)
            x.append(NN.getVote())                          #watch type of vote!
```

```
    else:
        for NN in MG.nodes():
            x.append(NN.getVote())
    for i in x:
        if i>bd:
            y=y+1
        if i<(-1*bd):
            n=n+1
        if i<bd and i>(-1*bd):
            a=a+1
    aa=100*a/len(x)
    yy=100*y/len(x)
    nn=100*n/len(x)
    print('astenuti: ',a,' ',aa,'%, favorevoli: ',y,' ',yy,'%, contrari: ',n,' ',nn,'%')
    print()
    if y>n:
        print('Passato')
    else:
        print('Respinto')
    fig = plt.figure(figsize=(15,5))
    ax1 = fig.add_subplot(1,2,1)
    labels = 'Favorevoli', 'Contrari', 'Astenuti'
    sizes = [y, n, a]
    explode = (0.1, 0.1, 0.1)
    ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
            shadow=True, startangle=90)
    ax1.axis('equal')
    ax2 = fig.add_subplot(1,2,2)
    ax2.hist(x,bins=60)
    ax2.plot((bd, bd), (0, 20), 'r-')
    ax2.plot((-1*bd, -1*bd), (0, 20), 'r-')
    fig.savefig('VoteAnalysis.png',orientation='landscape',bbox_inches='tight',dpi='figure')
    plt.show()
    return [y,n,a]

def VoteAnalysisLayer(MG, issue, bd, nLayer):
    from NetworkTool import ExtractLayer
    for layer in range(nLayer):
        G = ExtractLayer(MG, layer)
        print('Analisys of Layer: ', layer)
        VoteAnalysis(G, issue, bd)
        print('------------------------------------------------------------')


def VoteALayerLight(MG, issue, bd, do, nLayer):
    import matplotlib.pyplot as plt
    from NetworkTool import ExtractLayer
    print(issue)#,' ',len(issue))
    for layer in range(nLayer):
        G = ExtractLayer(MG, layer)
```

```
        print(G.number_of_nodes())
        x=[]
        y=0
        n=0
        a=0
        if do == 1:
            for NN in MG.nodes():
                NN.doVoteA(issue)                          #watch type of vote!
        for NN in G.nodes():
            x.append(NN.getVote())
        for i in x:
            if i>bd:
                y=y+1
            if i<(-1*bd):
                n=n+1
            if i<bd and i>(-1*bd):
                a=a+1
        fig = plt.figure(figsize=(15,5))
        ax1 = fig.add_subplot(1,2,1)
        labels = 'Favorevoli', 'Contrari', 'Astenuti'
        sizes = [y, n, a]
        explode = (0.1, 0.1, 0.1)
        ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
                shadow=True, startangle=90)
        ax1.axis('equal')
        ax2 = fig.add_subplot(1,2,2)
        ax2.hist(x,bins=60,)
        ax2.plot((bd, bd), (0, 20), 'r-')
        ax2.plot((-1*bd, -1*bd), (0, 20), 'r-')
        ax1.set_title('layer: '+str(layer))
        ax2.set_title('layer: '+str(layer))
    #fig.savefig('LayerVoteA.png',orientation='landscape',bbox_inches='tight',dpi='figure')
    plt.show()

def VoteDist(MG, plot):
    from scipy import stats
    import numpy as np
    import matplotlib.pyplot as plt
    vv = []
    for nn in MG.nodes():
        vv.append(nn.getVote())
    kur=round(stats.kurtosis(vv),2)
    skew=round(stats.skew(vv),2)
    if plot == 1:
        plt.xlabel('Vote')
        plt.ylabel('# of nodes')
        print('kurtosis',kur)
        print('skewness',skew)
        plt.title('kurtosis: '+str(kur)+', skewness: '+str(skew))
        plt.hist(vv, bins=60)
```

```
        plt.show()
    return([np.mean(vv), kur, skew])



def PollLayers(MG, Call1,Call2, bd, issue, lay1,lay2, socDes):
    import numpy as np
    import matplotlib.pyplot as plt
    from NetworkTool import ExtractLayer
    G1=ExtractLayer(MG,lay1)
    G2=ExtractLayer(MG,lay2)
    Interviews1=[]
    Interviews2=[]
    vote1=[]
    vote2=[]
    print(G1.number_of_nodes())
    print(G2.number_of_nodes())
    VoteAnalysis(G1, issue, bd, 1)
    VoteAnalysis(G2, issue, bd, 1)
    for i in range(Call1):
        Interviews1.append(np.random.choice(G1.nodes()))
    for i in range(Call2):
        Interviews2.append(np.random.choice(G2.nodes()))
    if socDes!= 0:
        for n in Interviews1:
            SocialDesirability(n, G1, socDes)
            vote1.append(n.getVote())
        for n in Interviews2:
            SocialDesirability(n, G2, socDes)
            vote2.append(n.getVote())
    y1,n1,a1,y2,n2,a2 = 0,0,0,0,0,0
    for i in vote1:
        if i>bd:
            y1=y1+1
        if i<(-1*bd):
            n1=n1+1
        if i<bd and i>(-1*bd):
            a1=a1+1
    for i in vote2:
        if i>bd:
            y2=y2+1
        if i<(-1*bd):
            n2=n2+1
        if i<bd and i>(-1*bd):
            a2=a2+1
    fig = plt.figure(figsize=(15,5))
    ax1 = fig.add_subplot(1,2,1)
    ax1.hist(vote1, 60)
    ax1.plot((bd, bd), (0, 10), 'r-')
    ax1.plot((-1*bd, -1*bd), (0, 10), 'r-')
```

```
    ax2 = fig.add_subplot(1,2,2)
    ax2.hist(vote2, 60)
    ax2.plot((bd, bd), (0, 10), 'r-')
    ax2.plot((-1*bd, -1*bd), (0, 10), 'r-')
    ax1.set_title('layer: '+str(lay1)+'; no: '+str(n1)+'; yes: '+str(y1))
    ax2.set_title('layer: '+str(lay2)+'; no: '+str(n2)+'; yes: '+str(y2))
    print('avg1: ', np.mean(vote1),'; std: ', np.std(vote1),'; avg2: ', np.mean(vote2),'; std: ', np.std(vote2))
    plt.show()
    return([n1,y1,a1,n2,y2,a2])

def ChiFreq(MG, bins, layer): #difference in vote distribution as chisq
    from NetworkTool import ExtractLayer
    G = ExtractLayer(MG, layer)
    x = []
    y = []
    xHist = []
    yHist = []
    for nn in MG.nodes():
        x.append(nn.getVote())
    for nn in G.nodes():
        y.append(nn.getVote())
    if max(x) > max(y):
        Max=max(x)
    else:
        Max=max(y)
    if min(x) < min(y):
        Min=min(x)
    else:
        Min=min(y)
    Span = Max - Min
    Size = Span/bins
    #print('span',Span,'size',Size,'bins',bins,'min-max', Min,Max)#-------------------
    for i in range(bins+1):
        xHist.append(0)
    for i in x:
        pos = int((i-Min)//Size)
        if i-Min >= Size*pos and i-Min <= Size*(pos+1):
            xHist[pos]=xHist[pos]+1
        else:
            print('error x ',i)#-------------------
    for i in range(bins+1):
        yHist.append(0)
    for i in y:
        pos = int((i-Min)//Size)
        if i-Min >= Size*pos and i-Min <= Size*(pos+1):
            yHist[pos]=yHist[pos]+1
        else:
            print('error y ',i)#-------------------
    for i in range(bins+1):
        xHist[i]=xHist[i]
```

```
        yHist[i]=yHist[i]
    Chi=0
    Df=0
    for b in range(bins+1):
        if xHist[b] != 0 or yHist[b] != 0:
            Chi=Chi+(((xHist[b]-yHist[b])**2)/(xHist[b]+yHist[b]))#numerical recipes in c 0-521-43108-5 pag 622
        else:
            Df=Df+1
    print(Df)
    return([Chi,bins-Df, xHist,yHist, MG.number_of_nodes(), G.number_of_nodes()])
```

7.4.1 BelVarDistance

Calculation of distances in the array of variable beliefs between two nodes.

```
def BelVarDistance(NodeA, MG):
    import numpy as np
    vv = []
    for nn in MG.neighbors(NodeA):
        vv.append(np.linalg.norm(np.array(NodeA.getVarB())-np.array(nn.getVarB())))
    return np.mean(vv)
```

7.4.2 BelFixDistance

Calculation of distances in the array of fixed beliefs between two nodes.

```
def BelFixDistance(NodeA, MG):
    import numpy as np
    vv = []
    for nn in MG.neighbors(NodeA):
        vv.append(np.linalg.norm(np.array(nn.getFixB())-np.array(NodeA.getFixB())))
    return np.mean(vv)
```

Bibliography

Bargigli, L., Di Iasio, G., Infante, L., Lillo, F., and Pierobon, F. (2015). The multiplex structure of interbank networks. *Quantitative Finance*, 15(4):673–691.

Battiston, F., Nicosia, V., and Latora, V. (2014). Structural measures for multiplex networks. *Physical Review E*, 89(3):032804.

Boccaletti, S., Bianconi, G., Criado, R., Del Genio, C. I., Gómez-Gardenes, J., Romance, M., Sendina-Nadal, I., Wang, Z., and Zanin, M. (2014). The structure and dynamics of multilayer networks. *Physics Reports*, 544(1):1–122.

Costa, L. d. F., Rodrigues, F. A., Travieso, G., and Villas Boas, P. R. (2007). Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242.

Criado, R., Flores, J., García del Amo, A., Gómez-Gardeñes, J., and Romance, M. (2012). A mathematical model for networks with structures in the mesoscale. *International Journal of Computer Mathematics*, 89(3):291–309.

Deffuant, G., Neau, D., Amblard, F., and Weisbuch, G. (2000). Mixing beliefs among interacting agents. *Advances in Complex Systems*, 3(01n04):87–98.

Diakonova, M., Nicosia, V., Latora, V., and Miguel, M. S. (2016). Irreducibility of multilayer network dynamics: The case of the voter model. *New Journal of Physics*, 18(2):1–10.

Lewis, K., Kaufman, J., Gonzalez, M., Wimmer, A., and Christakis, N. (2008). Tastes, ties, and time: A new social network dataset using facebook. com. *Social networks*, 30(4):330–342.

Magnani, M. and Rossi, L. (2013). Formation of multiple networks. In *International Conference on Social Computing, Behavioral-Cultural Modeling, and Prediction*, pages 257–264. Springer.

Natale, P. (2004). *Il sondaggio*. GLF editori Laterza.

Natale, P. (2009). *Attenti al sondaggio!* GLF editori Laterza.

Nicosia, V. and Latora, V. (2015). Measuring and modeling correlations in multiplex networks. *Physical Review E*, 92(3):032805.

Press, W., Teukolsky, S., and Vetterling, W. (1992). B. flannery, numerical recipes in c.

Quattrociocchi, W., Caldarelli, G., and Scala, A. (2014). Opinion dynamics on interacting networks: media competition and social influence. *Scientific Reports*, 4:4938.

Schelling, T. C. (1969). Models of segregation. *The American Economic Review*, 59(2):488–493.

Solá, L., Romance, M., Criado, R., Flores, J., García del Amo, A., and Boccaletti, S. (2013). Eigenvector centrality of nodes in multiplex networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 23(3):033131.

Ringraziamenti

Innanzitutto un grazie, con tutto il mio cuore, va ai miei genitori.

Il loro supporto, confronto, insegnamento mi ha non solo guidato nel mio lungo percorso formativo, soprattutto mi ha reso la persona curiosa, motivata e capace che sono oggi.

Ringrazio i miei nonni, Rita e Vincenzo, che ogni giorno si prendono cura di me, passandomi i loro valori, le loro storie e la loro passione per la vita.

A Luciana ed Oreste il mio ricordo ed il mio pensiero, sicuro sarebbero fieri di poter gioire con me di questo risultato che mi hanno aiutato a raggiungere, con la loro forza e la loro gioia.

Ai miei compagni della forte esperienza politica che ho potuto vivere in università l'incoraggiamento a non smettere mai, contro ogni ingiustizia, di lottare per i loro ideali, continuando a formarsi, dubitare, ragionare e vivere ogni aspetto della vita nella costante sfida di costruire un mondo migliore.

Alle mie amiche di vecchia data l'onore di avermi reso un uomo migliore, l'onere di essersi prese cura della parte più fragile di me.

Ringrazio i miei amici di sempre con cui così tanto ho condiviso, dalle esperienze più ardite e appaganti alle più quotidiane e felici, ringrazio i nuovi, che hanno aperto i miei orizzonti e mi hanno fatto capire quanto vasto e grande sia il mondo.

Un grazie speciale al mio relatore, la cui presenza costante e gli stimoli arguti hanno permesso a questo lavoro di ricerca di crescere, nonché di insegnarmi metodo e passione per la ricerca.

Abbiamo davanti a noi il mondo, un mondo che si troverà ad affrontare sfide sempre più complesse, spetterà a noi superarle nel più creativo e appassionato dei modi.