Predicting Influence of an Agent in a Complex Network

Bachelor thesis CKI (7.5 ECTS)
25 august 2014

Rogier van Dinther

Universiteit Utrecht

Supervisor: Dr. G.A.W. Vreeswijk
Reviewer: V. van Oostrom

Table of Contents

1. Introduction

Research on agents, and in particular Multi-Agent Systems (MAS), is an important part of the field of AI. Robot football, a rather famous and visible example of AI research, if somewhat popularised, is an example of this. The yearly International Conference on Autonomous Agents and Multi-Agent Systems (the AAMAS for short) is dedicated entirely to this field of research. An almost defining characteristic of these systems is that they are decentralised. All agents have equal authority. If the actions of all agents were orchestrated centrally, then they wouldn't really be multiple agents at all, but rather one big one. Given this lack of central control, especially as MAS's get bigger, a challenge of working with them is to still have ways to oversee and control the system as a whole.

With so many agents and no central control, it is good practice for agents to establish conventions between them to cooperate better. One way to control the MAS as a whole is to control these conventions. A researcher could choose these conventions directly, but this may be problematic: "Generating conventions offline is difficult, due to limited knowledge of society characteristics, time variance, and computational expense. Such conventions also lack robustness to evolving populations and environments." (Franks e.a., p. 448) In other words, they won't know the system well enough to make an informed decision, and any choice won't take into consideration that the system might change. Avoiding these problems and letting agents decide their own conventions leaves no control at all, while they might like to have a say in the matter. Perhaps we want to study the effect of different conventions on the MAS, or perhaps there is some outside factor like a hardware limitation that makes certain conventions preferable. A middle way is to choose so-called 'Influencer Agents' (IAs), where that agent's preference for a certain convention is dictated. The idea is that with these agents supporting it, the convention is more likely to (at least partially) be chosen by the system as a whole, while also avoiding the problems named above. But where do we place these IAs to be most effective?

The interaction in many MAS's can be abstracted to a network graph of agents, where edges represent the ability for two agents to interact. Nodes in a network have a certain 'influence', or 'network value'. What this means exactly depends on what exact measure is chosen, but at its most basic level, a more influential node is better able to spread information and behaviour through the network than a less influential one. In the same way, a more influential agent in the network is more likely to get its way when it comes to the conventions it wants. Thus, a good place for an IA would be with the most influential agent(s). Unfortunately, it's not simply a matter of mapping every individual agent to their influence through some function and then picking the k agents with the highest scores. 'Influence' in this sense is an agent/node's ability to influence others, so one agent's influence can only be defined in terms of its effect on the whole system. Ideally, a subset should be found that, as a set, most effectively spreads the chosen convention, but solving that so-called *influence maximisation problem* is generally NP-hard (Franks e.a., p. 448). In practice, it may work with small networks, but for real-world examples, with agents numbering in the thousands, this is not feasible.

In their paper *Learning influence in complex social networks*, Franks e.a. propose a method to find approximate solutions to the *influence maximisation problem* in feasible time by taking a sample of all agents and using them to build a model for predicting influence for any agent based on information about its location in the network. I implemented this method in a somewhat changed form. In this paper, using this implementation, I'm going to test how accurate the predictions are that such a model produces[1]. This paper is intended as a report explaining the workings of the program and discussing the results it produces. For a full understanding, it may help to look at this program and try it out.

1.1. The research question

A main part of Franks e.a.'s method is that it avoids doing a full calculation on all agents in a usually large network, and instead builds a model with a subset of these agents to make predictions. This is only a valid approach if the predictions that are made are close to the real values as given by a full calculation. Specifically, the question this paper will try to answer is: "Given a network of agents representing a multi-agent system, can Franks e.a.'s method produce a model that makes predictions about influence values of agents in this network that approximate the influence values given by an actual calculation with acceptable accuracy?"

1.2. Relation to AI

Franks e.a.'s method essentially uses supervised machine learning, an important concept in AI. A training set of agents are fed to the model with their input (information about network location) and desired output (influence value), and from that the model infers a function that predicts outputs given input. The training set is sure to be noisy, so any produced model will have to deal with that correctly to be accurate.

Furthermore, the eventual goal is to make an educated guess at effective places to place IA's, which in turn helps researchers of multi-agent systems to control the choice of conventions in these systems. In that way it contributes to the broader field of AI.

1.3. Overview

First, I explain in concept the method that Franks e.a. propose as well as other algorithm used (Section 2). Then I get into more technical detail about the exact implementation of these algorithms (Section 3). The results of the experiments and conclusions drawn from them are discussed in Section 5. For a full understanding, it may help to look at the program that was used to do these experiments and try it out.

[1] The source code of this program, as well as the results, can be found at: https://github.com/abackwood/Predicting-Influence-of-an-Agent-in-a-Complex-Network

2. The Algorithms in Concept

In this section I will describe the algorithms that I used for this experiment, starting with the general algorithm for predicting agent influence as proposed by Franks e.a., then the problem of *language coordination* that takes the role of 'influence model' here, and finally the random network generator that I used to mimic sampled subgraphs. For each, I will also discuss how I interpreted ambiguous definitions, and changes that I made and why. I make brief notes about the complexity of individual operations, but only discuss the complexity of the adapted method as a whole in section 2.4. It is easier to do this last, when we have all the information.

2.1. Franks e.a.'s algorithm

The algorithm goes through five high-level steps (Franks e.a., p. 448). I will discuss each in more detail below:

1. Take a subnetwork G_s of the original network G.
2. From G_s, take a random set of nodes $S \subset G_s$, of a predetermined size.
3. Choose a measure of influence and influence propagation (i.e. what is influence in this context, and how does a node actually influence other nodes with it?).
4. Obtain the influence for each node in S by running multiple simulations of the model.
5. Calculate 14 given "location metrics" for each node in G_s, and build a model to predict influence based on these 14 values for any node in the network, based on the nodes in S.

Most real world applications will deal with very large networks. The topology might not even be fully known. A network of a more practical size, of which the topology is fully known, is required for step 5. Therefore, we sample a subnetwork G_s. Ideally, G_s is a good representation of G. Franks e.a. propose three methods for sampling: Snowball Sampling (SNS), Metropolis-Hastings Random Walk (MHRW) and MHRW with Delayed Acceptance (MHRWDA) (Franks e.a., p. 450).

SNS starts with a randomly chosen node from a seed set and then samples neighbours at random from already sampled nodes, much like breadth-first search. MHRW and MHRWDA perform a random walk with biased transition probabilities. Delayed Acceptance also adjusts this bias to reduce the chance of backtracking. (Franks e.a., p. 450). Each comes with their own trade-off. "SNS [is] biased towards high node degrees, but SNS can produce good coverage of the local area around the start node. It is subject to greater variation between samples but may be useful for ensuring that a wide variety of structural properties are tested. MHRW and MHRWDA converge towards the node degree distribution exhibited in the full network, but there are no guarantees about the reproduction of any other metrics or structural properties." (Franks e.a., p. 450). Being that they are essentially graph search algorithms, all three algorithms have a complexity $O(n)$ with n the size of the desired subnetwork G_s, which is at worst the size of the original network G, though that would defeat the purpose of sampling.

There are notable differences between the setup Frank e.a. use and the setup I used. Franks e.a. sample 45 subnetworks of 1000 nodes each from a real-life network. Because I have no

access to large real-life networks and I am more interested in a proof of concept than actual results, I use randomly generated networks of 200-1000 nodes, rather than sampled subnetworks, so the choice of sampling algorithm is outside the scope of this paper. An explanation of the generation algorithm follows in section 2.3.

Picking a random subset of nodes from all nodes in G_s is straightforward, and has complexity *O(n)*. Seeing as I am more interested in the effectiveness of the method than saving time, here too I move away from Franks e.a.'s algorithm. After calculating the influence and location metrics of each node in G_s, I choose K subsets of nodes and build models with those subsets, emulating only having partial information. By comparing the predicted values against the real values, the effectiveness of the models can be tested. Repeating the process K times avoids errors brought on by chance.

A model for influence and influence propagation has to come up with a value for influence that can be ranked against others and is useful in step 5 for building a predictive model. An integer or real number would make sense, but is not strictly required. It also stands to reason that whatever is propagated, agents should by definition be more successful at propagating it as they rank higher on the influence scale. Beyond these constraints, a simple model is preferable over a complex one. Calculating the influence is easily the most time-consuming part of the algorithm (Franks e.a. iterate 50.000 times over the same procedure), a model that is fast to calculate will go a long way towards making the whole algorithm more efficient. Franks e.a. use a version of *language coordination*. Because I am testing their algorithm I would like to stay as close to their methods as I can, so I also use *language coordination*, which I'll explain in section 2.2.

A location metric is a way to quantify the position of a node in a network. There are many such metrics, and existing ones can be morphed to make new ones (look at *closeness centrality* below for an example of this). Franks e.a. have chosen 14 likely candidates to be predictors of influence for a node. These metrics are calculated for every node in G_s and the resulting 14 scores are later used as input for building the predictive model. Below is a quick overview of the chosen 14.

Let *n* be the node we're looking at, *N* the set of *n*'s neighbours, and *E* the set of edges in G_s:

Degree: the number of neighbours. "Intuitively the more individuals a node can communicate with the more it is able to influence." (Franks e.a., p. 449)

Local Clustering Coefficient (LCC): This is the proportion of neighbours of n who are also neighbours of each other.

Let $NN = \{ (n1, n2) \mid n1 \in N, n2 \in N, (n1, n2) \in E \}$.

I have interpreted this metric as $|NN| / (|N| * (|N|-1))$.

An exception occurs if $|N| = 1$. This leads to LCC = 0/0, which is undefined. I have defined LCC to be 0 in that case.

Lowest, Highest and Average Edge Embeddedness (L/H/AEE): This is the number of neighbours n shares with one of its neighbours. In other words, the size of EE = { n1 | n1 ∈ N, (n2, n1) ∈ E, n2 ∈ N }. As the name suggests, these three metrics are respectively the lowest, highest and average Edge Embeddedness of n with a neighbour.

Lowest, Highest and Average Edge Overlap (L/H/AEO): This is similar to Edge Embeddedness, only now we take the fraction of the total neighbours of both nodes n and n1 that they share. I have interpreted the total to be the sum of the degree of n and the neighbour in question, let that be T. At best, n and n1 share all their neighbours. Then T = 2*|N| and |EE| = |N|. To answer to the intuition that in that case the Edge Overlap is 1, we double |EE|. Thus, Edge Overlap is given by (2 * |EE|) / T.

Average Shortest Path Length (ASPL): This is self-explanatory: the average length of shortest paths from n to all nodes. I have interpreted this to include n itself. Intuitively, if the distance between n and other nodes is relatively short, n would tend to be influential.

Average Neighbour Degree (AND): As the name says, this is the average degree of n's neighbours.

Closeness Centrality (CC): This is the reciprocal of ASPL: 1 / ASPL.

Betweenness Centrality (BC): This is the fraction of all shortest paths in the network that include node n. As Franks e.a. point out: "a node with high betweenness is intuitively influential by virtue of being a conduit for more information flows." (Franks e.a., p. 449).

Eigenvector Centrality (EC) & Hyperlink-induced Topic Search (HITS): EC determines how much time a random walk across the network spends at node n, and HITS tries to "determine hubs and authorities in a network, where a hub is a node that links to many authorities, and an authority is a node that is linked to by many hubs." (Franks e.a., p. 449). These last two metrics seem to be intended for large networks, not the relatively small networks that I work with, so I am not going to use them for my own experiment.

With these 12 values and the influence calculated for the nodes in S, we can apply multiple linear regression to infer a linear function. Given the 12 location metrics for a node, we can then predict the influence of that node without having to test it. As said above, what I'm interested in is the difference between the real influence and the prediction, so we'll be calculating the influence of every node but only base the predictive model on a few.

2.2. Language Coordination

Franks e.a. use an adaptation of the *language coordination* problem as defined by Salazar e.a. as their influence model (N. Salazar-Ramirez e.a., in: Franks e.a., p. 447). Remember that we require a measure of influence and a means by which influence spreads.

In this problem, each agent has a lexicon of 10 words mapped to 10 concepts, representing the “language” this agent speaks. At the start, this lexicon is generated at random. Now, the MAS goes through a set number of rounds *t*. Each round, an agent does three things:

1. It sends one randomly chosen mapping to a randomly chosen neighbour. If this neighbour has the same mapping (speaks the same language, as it were) the communication is considered successful, otherwise a failure.
2. There is a small chance (p = 0.01) that it sends part of its lexicon to all neighbours, along with its *communicative efficacy*, which is to say: the number of successful communications in the last 20 rounds.
3. There is a small chance (p = 0.01) that it updates its lexicon with the received partial lexicon with the highest *communicative efficacy*, using a two-point crossover.

The dominant lexicon *L’* is the lexicon that most agents share after all rounds are done. Let *L* be the agent’s original lexicon before round 1. Then that agent’s influence is defined as $|L \cap L'| / |L \cup L'|$. Thus, an agent that has no mapping in common with the dominant lexicon has an influence of 0, one that exactly matches the dominant lexicon has an influence of 1, and any middle ground has influence somewhere in the [0,1] range. This whole routine is repeated a set number of times *r*. The final influence of an agent is the average of its *r* influence scores.

2.3. Network generation

Franks e.a. call for the sampling of a larger network to obtain a subnetwork of a practical size. As said, I will simulate this with randomly generated networks. To mimic a sampled subnetwork, I use an adaptation of the random network generation algorithm by Bayati e.a.

1. Given an N = number of nodes, and MAX = highest degree one node can have, we generate a list of integers D, of size N, with each value randomly chosen between 1 and MAX. This represents the desired degree for each node.
2. Pick two nodes i and j at random so that $i \neq j$, $D[i] > 0$ and $D[j] > 0$, and (i, j) is not already an edge in the graph. Put (i, j) as an edge in the graph and subtract 1 from both D[i] and D[j].
3. Repeat step 2 until no more edges can be added.
4. If some value in D is still higher than 0 or the graph is partitioned, start over from step 1. Else output the graph.

Bayati don’t explicitly include an upper limit MAX in their definition of the algorithm. This is a property of my implementation. It gives us some control over the shape of the network. A low MAX will tend to produce a larger degree of separation between nodes, and higher MAX lower degree of separation.

The condition that the graph must not be partitioned in step 4 is another addition on my part. The problem of *language coordination* and the underlying idea of influence propagation only seem to make sense in a network where information from one node can in principle reach every other node.

2.4. Complexity of the Adapted Algorithm

If we look back at the step-by-step summary of Franks e.a.'s algorithm, sampling a subnetwork has been replaced by randomly generating a network. Step 1 in doing that involves generating a list of size N and thus has complexity O(N). Since choosing a random number and reading a value from an array are O(1) operations, the most complex operation in step 2 is checking that a given edge does not already exist. In the worst case, the list of edges has size ½*N*MAX - 1, making the complexity of this step O(N*MAX). Step 3 repeats step 2 for every edge added, worst case ½*N*MAX times. The complexity of steps 1-3 then is O(N) + ½*N*MAX*O(N*MAX) = $O(N^2*MAX^2)$. We can't give a complexity over step 4 other than to say that the probability that a good graph is eventually generated approaches 1.

As discussed, choosing a random subset of nodes is straightforward and can be done in linear time, but we do it K times, leading to a complexity for this step of O(K*N). More important is the complexity of the influence model. The most complex operation in one iteration of the *language coordination problem* is when a broadcast happens, being O(E), with E the set of edges in the graph. However, we know the maximum degree of any one node, so we can adjust that to O(MAX). This operation could happen for every agent, so the complexity for a whole iteration is O(N*MAX). We iterate a set number of times *t*, and run the whole simulation *r* times. By my logic, the final complexity in the context of the bigger algorithm comes out to be O(N*MAX*t*r).

The calculation of a location metric generally has complexity O(N), or O(N*MAX) if it involves looking at neighbours. Exceptions are the ASPL and BC location metrics. Both require knowledge of all shortest paths in the network. Finding those takes $O(N^2)$ time. To avoid expensive double work, all shortest paths are calculated and stored beforehand using breadth-first search from each node. Nonetheless, the final complexity of the calculation of location metrics is $O(N^2)$.

Putting together the above, the complexity of the whole is $O(N^2MAX^2)$ + O(N*MAX*t*r) + O(K*N) + $O(N^2)$. If we assume *t, r* and *K* are constant and just focus on the parameters that define the network, this can be abbreviated to $O(N^2MAX^2)$ + $O(N^2)$.

3. A Technical Walkthrough

In this section I go into how the different algorithms introduced in section 2 work together to form a three-phase pipeline, and some specifics of their implementation. I will focus on details that are not implicit in the algorithms, as for the most part the implementation is straightforward. The network generation and calculation of the location metrics is written in Java. Running Language Coordination on the network happens in NetLogo. Lastly, applying linear regression to the results happens in Java again.

3.1. Network Generation and Calculation of Location Metrics

The network generation phase is written in Java because it allows for complex objects (contrary to NetLogo), making it very useful in building a graph and exporting the appropriate information to a file. In the same program, we also calculate the location metrics because those require only the topology that we just created and, again, are easier to code in Java.

In practice creating a network for N = 1000 and MAX = 20 takes about half a minute including writing the results to two files:

1. A *"Graph_N_NLinput.txt"* file with NetLogo syntax that builds the network. The contents of this file need to be copied into the "setup-topology" procedure in the NetLogo model. Essentially, it creates N turtles and then creates the links between them using the "create-links-with" procedure. Since the links are undirected, every link is created twice. According to its definition, NetLogo ignores the command to make links that already exist, so this does not lead to problems..
2. A *"Graph_N_LMinput.txt*" file with the location metrics for each node, to be used in the third phase. This file is automatically read and parsed by that program, provided it is in the same directory.

3.2. Running the Language Coordination Algorithm

This phase happens in NetLogo. This way, we can tweak parameters easily (e.g. the chance that a node updates, or the number of iterations). Agent lexicons are set to a random permutation of the list {1..10} at the start. After running a set number of iterations a set number of times, the influence for each node and the settings of the parameters are exported into *"Graph_N_influence.txt"*.

As an extra feature, the dominant lexicon and influence of each node is calculated every 1000 iterations. The color of an agent is scaled along the influence value. Lighter color is higher influence. There is an option *show_lexicon* to display a list of all different lexicons and their frequency. This is only to give the viewer some feedback about what's going on and serves no functional purpose.

3.3. Building the Influence Model

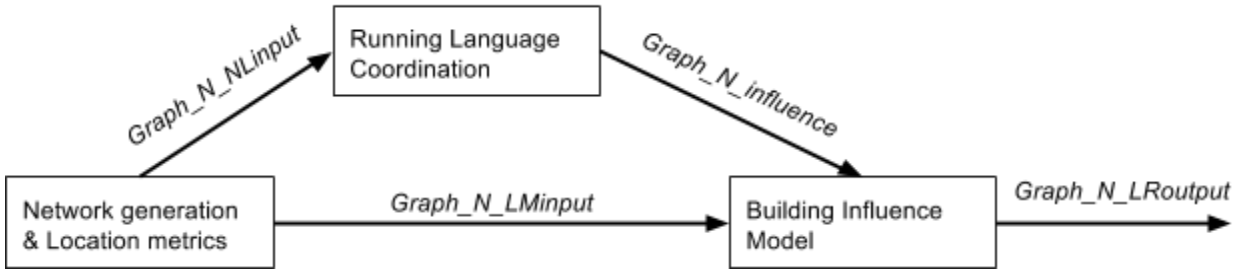
This phase is performed in Java. Location metrics and influence are read from the files produced in the previous phases and combined into a matrix. The following is repeated K times

We take a random sample of S nodes and perform Ordinary Least Squares (OLS) multiple linear regression with the 12 location metrics as independent variables, and influence as dependent variable. For this, I've used the Apache Commons library for Java.[2] We apply the linear function that results from this on all nodes and calculate the average deviation from the real influence values. If this is low enough, that would indicate it is indeed possible to reliably predict influence from a sample of agents/nodes, rather than naively test them all.

The Apache package requires that the input matrix containing the X values (the location metrics) for OLS multiple linear regression be *non-singular* (i.e. it has an inverse). Matrix singularity has nothing to do with the data, and is purely a property of the matrix itself. A singular matrix can easily be made non-singular by changing the value of one entry just a little (say, 0.001) (Cook, 2012). However, this makes any result derived using that matrix unreliable (and in fact this seems to be the case; see Section 5). As a middle ground, we first try to choose a different subset (and thus different matrix). If we've done this 10 times and are still unable to find a non-singular matrix we apply the crude solution above.

The results and all parameters are printed to a file *Graph_N_LRoutput.txt*.

Below is a representation of the entire pipeline, with processes and files.



[2] To be precise, the "org.apache.commons.math3.stat.regression.*" package, and its OLSMultipleLinearRegression class in particular.

4. Discussion

In retrospect, it would have been easier to also program Language Coordination in Java, and have the whole pipeline be in the same language. Potentially, this would have cut out the need for any file I/O by putting it all in one execution, though I would likely still have left the network generation as a separate program so I could experiment with the parameters there until I was satisfied with the resulting network, and only then use that network for the other phases. By the time I discovered this, rewriting it would have taken more work than creating the rest of the pipeline around it, which is what I did.

The definition of the ‘dominant lexicon’ was ambiguous. Franks e.a. only say this about it: “We define the dominant lexicon as the one that is shared by the highest number of agents.” (Franks e.a., p. 451). To me, this leaves room for two interpretations: 1) the literal sequence of mappings as a whole that is shared by most agents, or 2) the compound lexicon of the most occurring mappings per concept. I chose the first one because it seemed counterintuitive to me that the dominant lexicon might be one that no agent actually has, which is true for the second interpretation. However, following this second interpretation might have led to other results. This would have to be tested.

Another note about the lexicons was that different words could be mapped to the same concept (or in implementation-specific terms: a number could occur more than once). Franks e.a. make no note of if this is intended or not, but it is a consequence of their version of Language Coordination. Using two-point crossover for two random permutations of list {1..10} will almost always produce a list where some numbers occur more than once. Therefore, I assume this was what Franks e.a. had in mind, but I may be wrong.

By using a randomly generated network, rather than a “natural” one, the results might be skewed a certain way. Every random network generator has a bias of some sort. Franks e.a. note that synthetic networks “tend to be poor models of real-world networks.” (Franks e.a., p. 450). Because I had no access to a suitably large network and using one was arguably beyond the scope of this experiment, I made use of an artificial one, but Franks e.a.’s objection is duly noted.

5. Conclusion

The last phase in the pipeline produces a file like the following[3]:

N = 200
MAX = 20
Runs = 20
Iterations = 50000
P_broadcast = 0.05
P_update = 0.05

S = 20
K = 500

1: 0.25372087734714716 + 5.136482300939562E-4*Degree + -0.1645013104937723*LCC + 1.6210005528103256E14*LEE + -0.026309596772714158*HEE + 0.101685012659973*AEE + -2.350450810331084E15*LEO + 0.2518341951624404*HEO + -0.6807649683877679*AEO + -0.028448828734276975*ASPL + -0.003296042441092845*AND + -0.04598653589172952*CC + -2.7469984788639774*BC
Mean deviation: 3.087620350765887E11

2: -0.5276719131352765 + 3.59925002587493E-4*Degree + -0.3643959792874145*LCC + 8.179825395836522E13*LEE + 0.015105723769568712*HEE + -0.04800628855799214*AEE + -1.1860746868147692E15*LEO + -0.4273837525144249*HEO + 1.9551231109339726*AEO + 0.0838741789192411*ASPL + 0.0025236510815174456*AND + 0.8853808335319295*CC + -0.1498560604346686*BC
Mean deviation: 1.5580621064015915E11

…

500: -3.627739658904719 + -0.005742459140287733*Degree + -0.6822711045945867*LCC + -21.015028084971302*LEE + 0.01854888403502706*HEE + -0.1744884397573462*AEE + 222.9911865808991*LEO + -0.2138132496708975*HEO + 3.0607395172658767*AEO + 0.4551682940576456*ASPL + 0.005016634369275013*AND + 7.232733002744823*CC + 1.0436033744200923*BC
Mean deviation: 0.05055787058341176

Average deviation over all samples: 4.85422886461739E10
- without out of bounds deviations (413): 0.10841543871731572

The first lines state the parameters of the pipeline. After that follows, for every sample, a representation of the linear function that is the predictive model and its mean deviation from the real values. At the bottom is the average over all these values. Sometimes models will occur with deviations far above 1. Since influence will always have a value in the range [0,1], a mean deviation of more than 1 is meaningless. Therefore, there is a second value that is the same average but with these nonsensical models excluded, with the number of samples it's based on (413 in this case) given. These "out of bounds" cases are most likely caused by our treatment of singular matrices described in Section 3.3. Larger networks have less of a problem with singular matrices, and also less often produce nonsensical models.

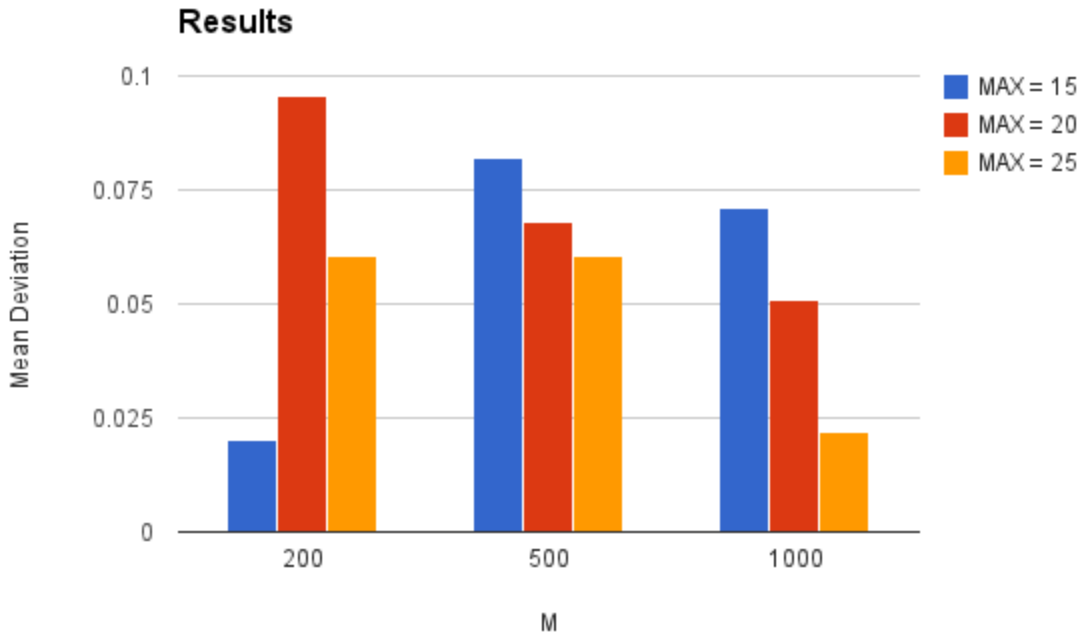
[3] The full result files can be found at:
https://github.com/abackwood/Predicting-Influence-of-an-Agent-in-a-Complex-Network

Another thing to note is that the parameters *p_broadcast* and *p_update* in the example are set to 0.05, while Franks e.a. define them to be 0.01. I discovered that with 0.01 the lexicons didn't converge to one dominant lexicon. If in a network of 100 agents the "dominant" lexicon is shared by 5 agents, that's not a very useful result. With the probabilities increased to 0.05 the network did converge by the end of a run. The dominant lexicon would typically have 50-75 agents backing it up, making it more credibly the dominant one. That's why I used those parameters. The difference may be caused by the differences in network size. My used networks are smaller, so the absolute number of broadcasts and updates is also proportionally smaller. For influence to spread these things need to happen, so a larger probability is necessary to compensate.

The only two parameters that I changed were the ones for the network generator, N and the maximum degree, while keeping all else equal. For each set of parameters, the pipeline was run twice for two different networks. In the third phase of each run I made sure to have at least 300 "good" samples to base the mean deviation on.

N	MAX	Mean Deviation	Samples	Average
200	15	0.02019006474	500	0.02015035588
		0.02011064701	500	
200	20	0.1084154387	413	0.09565569063
		0.08289594255	445	
200	25	0.02342103659	323	0.06061394215
		0.09780684771	540	
500	15	0.07894498653	462	0.08191475236
		0.0848845182	444	
500	20	0.06452971754	449	0.06817253894
		0.07181536035	438	
500	25	0.07244831444	414	0.06062804754
		0.04880778065	473	

1000	15	0.08146025825	412	0.07085197497
		0.06024369168	413	
1000	20	0.05083183485	474	0.05107963569
		0.05132743653	449	
1000	25	0.02299401061	500	0.02192379828
		0.02085358595	496	



The question this paper is trying to answer is: “Given a network of agents representing a multi-agent system, can Franks e.a.’s method produce a model that makes predictions about influence values of agents in this network that approximate the influence values given by an actual calculation with acceptable accuracy?” In the table and graph above, a low mean deviation indicates that the models produced for that network can accurately predict the real values for influence. Conversely, a higher mean deviation means these predictions are less accurate.

A pattern almost emerges. The outlier is N = 200. The value for MAX = 15 there really doesn’t fit in with the rest of the results. Based on the other data, we would expect it to be much higher than “N = 200, K = 20”, literally off the chart. Given that it’s also by far the lowest value where “MAX = 15” is concerned, we may conclude that something went very wrong in calculating it. If we

dismiss that value and assume that in reality it follows a similar trend as the “MAX = 15” values for the other values of N then it seems that both N and MAX are inversely proportional to the mean deviation in predicting the influence in that network. Lower N and/or MAX means a lower mean deviation, and thus a more accurate prediction.

In the case of MAX that could well be. It stands to reason that MAX, being the maximum allowed degree of one node, has an effect on the expected distance between any two nodes. A higher MAX value would tend to cause paths to be shorter. According to the data, denser networks are more “predictable”. However, for N an inverse relation is a rather strange outcome. Since this method relies on sampling to make predictions, and the sample size is constant, we might have expected a higher N to lead to lower accuracy, i.e. higher mean deviation, as the sample is smaller compared to the true number of agents. One possibility is that, if such an inverse relationship even exists between N and the mean deviation, it is asymptotic and rapidly flattens out. Another possibility is that the interplay between N and MAX in generating a network is not quite so simple that you can change one and not the other without creating a very different kind of network. Then “N = 500, MAX = 25” leads to a completely different kind of network than “N = 1000, MAX = 25” does, and one that is apparently a good deal harder to predict influence values for.

Because of all this, it’s hard to make hard conclusions about the effect of network size and topology on the accuracy of influence prediction with Franks e.a.’s method. What we do see is that all mean deviations are below 0.1. For a value in the range [0,1], a deviation of less than 0.1 is really good. Assuming an equal distribution of values, a fully random guess would have an average deviation of 0.5. A high overall accuracy like that definitely speaks for the effectiveness of Franks e.a.’s algorithm. I highly doubt that the accuracy increases with network size as is implied in the data above, but if these results scale to networks of real-life size (in the order of thousands of agents), then it is probably a very useful aid in choosing good positions for placing Influencer Agents to control convention choice in a decentralised, online, way. As said in the introduction, being able to control the behaviour and conventions of a MAS can aid research involving these systems.

Further research could investigate if these results do indeed scale to more realistic-size networks and what kind of relation, if any, there is between size and accuracy. Another thing to look at would be the way I’ve used random network generation to mimic sampling a real network. Any randomly generated network has a bias. Perhaps the results are different if a large static (possibly itself randomly generated) network is sampled using one of the sampling algorithms proposed by Franks e.a.

Appendix A: Glossary

- **IA:** Influencer Agent; an agent whose convention preference is dictated from outside the system.
- **Influence:** A measure of how likely an agent is to ‘get its way’ in what conventions are chosen.
- **Language Coordination:** A problem originally defined by Salazar e.a.; adapted by Franks e.a. to serve as influence model.
- **MAS:** Multi-Agent System; here, considered equivalent to a network of agents
- **Two-point crossover:** A method to produce a ‘child’ of two lists of equal size, so that the child has features of both lists. A random startpoint and endpoint is chosen and the lists exchange the sublist defined by that start- and endpoint. The resulting list has the same size as its parents.

Appendix B: Works Cited

Franks, Henry, Nathan Griffiths and Sarabjot Singh Anand. "*Learning influence in complex social networks." Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems* 6-10 May 2013: 447-454. Print.

Bayati, Mohsen, Jeong Han Kim, Amin Saberi. "*A Sequential Algorithm for Generating Random Graphs.*" *Algorithmica* Dec. 2010: 860-910. Print.

Cook, John. "Making a singular matrix non-singular". *The Endeavour*. 13 June 2012. Web. 16 March 2014. http://www.johndcook.com/blog/2012/06/13/matrix-condition-number/