Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Opinion formation by ”employed agents” in social networks

Stefan Brugger & Christoph Schwirzer

Zurich
May 2011

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Stefan Brugger   Christoph Schwirzer

Contents

1 Individual contributions **4**

2 Introduction and Motivations **4**

3 Description of the Model **6**
3.1 Bounded confidence model 6
3.2 Informed agents extension 6
3.3 Our proposal 7

4 Implementation **9**
4.1 Model for graphs and generation 9
4.1.1 Real world graphs 9
4.1.2 Random graphs 10
4.1.3 Generation of the graphs 12
4.2 Statistics on the graphs 13
4.3 Implementation of employed agent model 14

5 Simulation Results and Discussion **18**
5.1 Experimental setup 18
5.2 Influence of the parameters μ and u 19
5.3 Influence of the number of employed agents 23
5.4 Impact of the selection strategy for employed agents 31

6 Summary and Outlook **37**

7 Appendix **39**
7.1 References 39
7.2 List of Figures 40
7.3 List of Tables 42
7.4 Source code 43
7.4.1 Simulation 43
7.4.2 Generation 49
7.4.3 Statistics 52
7.4.4 Utility 55

1 Individual contributions

The entire report was created in a cooperative manner.

2 Introduction and Motivations

Social networks play an important role in many people's life. Especially in the last years social online communities like Facebook increased in size rapidly. In such a network, people have up to thousands of friends and opinions can spread quickly. By stating your opinion publicly, you influence and eventually change the opinions of people connected to you. For that reason social online networks became highly interesting for many applications in economy and politics. For instance, this notably could be seen by the 2008's American presidential election, where campaigns for the candidates were extended to social networks. For future elections, such networks might play an even more important role than today.

In this report, we like to study, how such social online networks can be modelled and how opinion formation can be influenced. We studied two research articles about the subject opinion formation.
In "*Minorities in a Model for Opinion Formation*" [LZ04] a continuous model for opinion formation was explored to investigate how the final opinion distribution depends on the choice of different model parameters. The model was presented in [WN02] and is referred as "*bounded confidence model*". In "*Opinion Formation by Informed Agents*" [AA10] a continuous and network-based model for opinion formation with *informed agents* was proposed. Informed agents are common people, indistinguishable from usual agents. They try to shift the public opinion towards a desired direction. Therefore they pretend to have an opinion similar to their neighbors' opinions to gradually change the opinion of the individuals in their neighborhood. In [AA10] the influence of informed agents in different network structures and parameter settings was analyzed.

In this report, we consider a continuous and network-based model for opinion formation with *employed agents*. It is based on the bounded confidence model described in [LZ04]. Employed agents are entities in a network, which are hired by some party to advance a particular view and to argue for that view in their neighborhood. We like to investigate the influence of employed agents on opinion formation in different network structures. In particular, we like to run simulations on real world friendship graphs from the Facebook networks of several American universities. Moreover, we want to run the same simulations on generated graphs with similar characteristics

to explore to which extend generated graphs can be used to approximate real world friendship networks.
Finally, we analyze the impact of a particular selection strategy. Therefore, a simple greedy strategy for selecting the employed agents is compared to a strategy which chooses them randomly.

3 Description of the Model

3.1 Bounded confidence model

In the bounded confidence model as described in [LZ04], there is a fixed size population of agents. Each agent has an opinion $x(t) \in [0, 1]$ at time t. There exists a global threshold u which is referred as "uncertainty level" in [AA10]. At each time step t, two randomly chosen agents x_A and x_B meet and interact if the difference between their opinions is smaller than their uncertainty level, i.e. if $|x_A(t) - x_B(t)| < u$. If the two agents interact, they change their opinions according to the following update rules:

$$x_A(t+1) = x_A(t) + \mu \cdot (x_B(t) - x_A(t))$$
$$x_B(t+1) = x_B(t) + \mu \cdot (x_A(t) - x_B(t)),$$

where the constant $\mu \in (0, 1)$ is the model's convergence factor which corresponds to the influence the two agents have on each other's opinion. In [AA10] the convergence parameter μ was chosen from the interval $(0, 0.5]$. For $\mu = 0.5$ two agents agree on the same opinion after one single interaction.

Note that the average of the opinions of two interacting agents is the same before and after an interaction, therefore the bounded confidence model cannot be used to investigate a shift in the community's opinion of a social network.

3.2 Informed agents extension

Mohammad Afshar and Masoud Asadpour describe in [AA10] an extension of the bounded confidence model. As setting they choose a graph structure where each node corresponds to an agent. Every agent has an opinion $x \in [-1, 1]$ and an uncertainty level u. The population is divided into two groups named *majority* and *informed agents*. The informed agents try to change the public opinion towards +1. Moreover, informed agents are indistinguishable from the majority, i.e. an agent A interacting with an agent B cannot decide whether B is an informed agent or not.

Two agents can only interact with each other if there is an edge connecting them in the underlying graph structure. At each time step t an agent A is chosen at random and from A's neighborhood an agent B is selected randomly. The two agents interact with each other if $|x_A(t) - x_B(t)| < u_A$, similar to the bounded confidence model.
The update rule for the opinion change is more fine grained. Each agent has a *social force* which corresponds to the will to accept a neighbors' opinion, and a *self force* which reflects the will to keep its own opinion. Informed agents have an negligible

self-force in order to be in higher conformance with their neighbors. Additionally, informed agents have a *goal force* conforming to their motivation to change other agents' opinions.

3.3 Our proposal

In our proposed model, the community is modeled as an undirected graph with n nodes. Each nodes corresponds to an agent which has an opinion between -1 and $+1$. The set of agents is partitioned into two groups: *normal* and *employed agents*. The employed agents are entities which are hired by some party to hold the opinion $+1$ and to do as much as they can to change other agents' opinions towards $+1$. For instance, a company could pay some users of a social online community like Facebook to agree about a topic and to try to convince their friends of that view. For that reason, in each interaction with another agent, an employed agent pretends to have an opinion as close to $+1$ as possible to be able to interact with that agent, i.e. if an employed agent E meets a normal agent A with opinion x_A, E pretends to have the opinion $\min(x_A + u, 1)$, where $u \in (0, 1)$ is the global threshold as described in the bounded confidence model. The initial opinions of the normal agents are chosen uniformly at random from $[-1, 1]$.

At each time step t, an agent x_A and one of its neighbors x_B is chosen at random. Depending on their types, there are different update rules to be applied:

(a) *Two normal agents meet:*
If the difference between their opinions is smaller than the uncertainty level, both agents change their opinions, i.e. if $|x_A(t) - x_B(t)| < u$ holds, their opinions are updated as follows:

$$x_A(t+1) = x_A(t) + \mu \cdot (x_B(t) - x_A(t))$$
$$x_B(t+1) = x_B(t) + \mu \cdot (x_A(t) - x_B(t))$$

(b) *A normal and an employed agent meet:*
Without loss of generality assume A is the employed agent. Agent A pretends to have an opinion $x_A(t) = \min(x_B(t) + u, 1)$. The two agents interact as described in (a) but A's opinion doesn't change, i.e. A's opinion is reset to $+1$ after the interaction.

(c) *Two employed agents meet:*
Nothing happens.

Note some differences between the informed agents described in [AA10] and the proposed employed agents: At first, employed agents are a group of explicitly hired

entities, whereas informed agents are independent individuals. Moreover, in contrast to informed agents, employed agents are not completely indistinguishable from normal agents, since normal agents could exchange information about opinions of former interaction partners and may detect employed agents. Furthermore, an employed agent satisfies the interaction condition for all its neighbors at any point in time. However, an informed agent changes his opinion in course of meetings. Hence, he may become unable to fulfill the interaction condition with some of his neighbors.

4 Implementation

For the implementation of our proposed model, the generation of the random graphs as well as for the statistics and plots, MathWorks *Matlab*[1] has been used.
The most important part of the Matlab code, written for this project, is the implementation of the simulation process of the proposed employed agent model. As we want to run this part for various parameters as well as for a large number of time steps and a moderately large number of agents, i.e. nodes in the underlying graph, the performance of this part of the implementation is crucial for the overall runtime. Furthermore, one fixed simulation setup has to be run several times to investigate the average value for that parameter setting.

The two main techniques that have been used for writing code with good performance are *vectorization*[Mat] and *parallelization*. For the latter, Matlab's *Parallel Computing Toolbox* has been used. Moreover we used Matlab's *Profiler* to find bottlenecks in our code. Despite all that, we tried to keep the code clean and well structured by following many of the advices given in Richard Johnson's MATLAB Programming Style Guidelines[Joh].

4.1 Model for graphs and generation

4.1.1 Real world graphs

As we want to simulate our model on both randomly generated and real world graphs, the first goal was to get some real world data of social networks, preferably of a social networking service like Facebook. It turned out that the data published with "Social Structure of Facebook Networks" [TKMP10] is a great resource for these purposes. Those data sets from September 2005 contain the full friendship graphs of 100 American colleges and universities; the graphs contain only intra-school links. Moreover, the datasets include further information like gender and graduation year of each person. We didn't use that information, but it could be interesting for future research. The sizes of the provided networks vary between 769 and $41,554$ nodes. We restricted our investigations to six of those networks. We chose the three networks with the smallest and the three with the largest number of nodes.

Furthermore, we decided to only consider the largest connected component for each network. One reason for this is that it's easier to define the average shortest path in a graph where each node is connected via a path to every other node. Another reason is that selecting an employed agent out of a very small connected compo-

[1] Version R2010b

nent could dramatically increase the variance of the results. An employed agent in a very small connected component can influence only a very small number of other agents, whereas an agent in a network consisting of just one connected component can at least theoretically influence (directly or indirectly through its neighborhood) every other agent's opinion. However, restricting the network to the largest component does only exclude less than 1% of the nodes for the case of our considered real world networks. Hence, it cannot change the results significantly, as with less than 1% of the nodes dropped, the worst case absolute change of the community's average opinion is less than 0.02. This can be easily calculated by assuming that the discarded agents all have opinion -1, while the majority, i.e. the agents associated with the nodes in the largest connected component all have opinion $+1$: $|(1.00 \cdot (+1) - (0.01 \cdot (-1) + 0.99 \cdot (+1))| = 0.02$.
Choosing six networks as described above and discarding nodes that were not connected to the largest component, we obtained graphs of sizes 762, 962 and 1,510 resp. 35,111, 36,364 and 41,536. These networks will be referred as small resp. large real world graphs.

4.1.2 Random graphs

The generated random graphs on which we will compare the employed agents model to the real world networks are generated using the same three different models that have been used in [AA10]: *random*, *scale-free* and *small-world*.
The random networks are generated using a variant of the Erdős-Rényi model [Wik11b], where each possible edge is included in the network with a fixed probability. The scale-free networks are generated using the Barabási-Albert model as described in [BA99]. For the small-world graphs we used the "Watts and Strogatz model" as illustrated in [WS98].

The parameters for the random graphs have been chosen such that they are similar to characteristics of the real world graphs. Doing so, for each of the three random graph models the size of the graphs to be generated is chosen as 1,000 resp. 35,000. In addition, other parameters have been chosen in such a way that characteristic values for the graph structure like average number of edges per node, average shortest path and clustering coefficient are similar to the real world networks. For a detailed list of parameters used for generation see Table 1.

Network	Parameter		Value	
	Number of nodes	n	1000	35000
Random	Probability of link creation between two nodes	p	0.0401	0.0024
Scale-free	Number of initially placed nodes	m	21	39
	Number of nodes a new added node is connected to	m_0	20	38
Small-world	Average nodal degree	k	40	76
	Rewiring probability	β	0.25	0.25

Table 1: Parameters used for graph generation

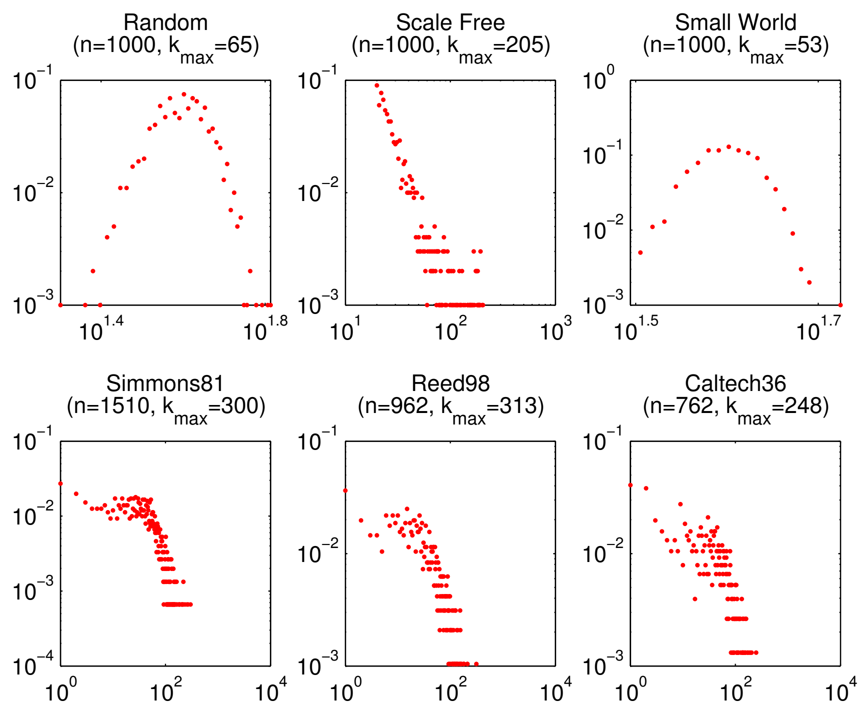

Figure 1: Degree distribution for the small networks in log-log scale (x-axis: sorted order of degree, y-axis: fraction of nodes having this degree)

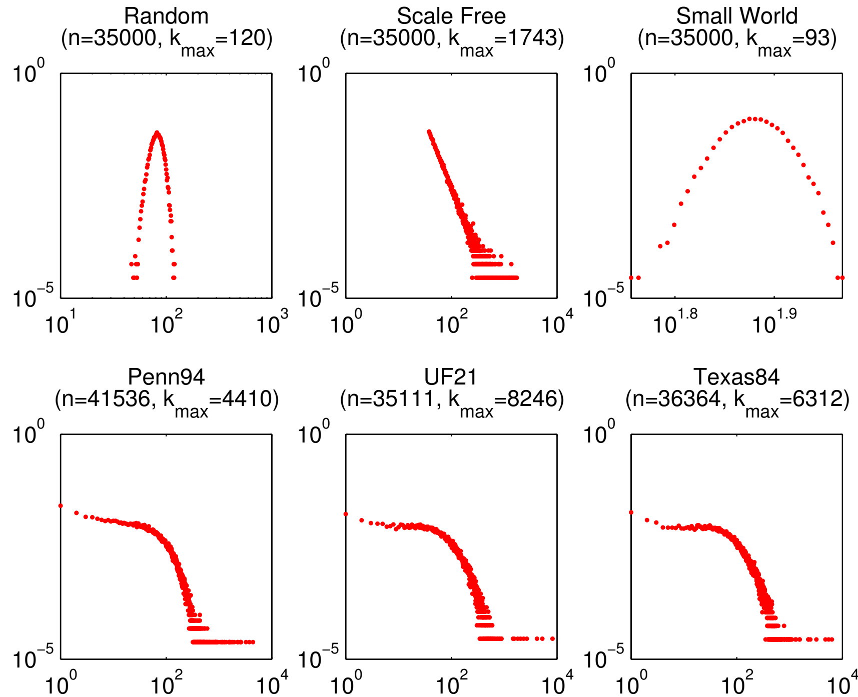

Figure 2: Degree distribution for the large networks in log-log scale (x-axis: sorted order of degree, y-axis: fraction of nodes having this degree)

4.1.3 Generation of the graphs

Loading and generating graphs of up to about 42,000 nodes, it was important to represent the graphs in a data structure with size that is not quadratic in the number of nodes[2]. For that reason we used a symmetric sparse matrix to represent the graphs.
For the real world graphs there was no generation. Moreover importing them to Matlab was easy as they each were given as a sparse matrix saved to a *.mat*-file that could be loaded by simply using Matlab's *"load(...)"* command.
Note that for the process of generating the random graphs it was not feasible to construct the whole adjacency matrix represented as a full matrix, for the reason mentioned above. Moreover *"inserting or removing a nonzero* [into a sparse matrix] *may require extensive data movement"*[GMS92], therefore insertions have been postponed by collecting non-zero entries in a column vector first and afterwards filling it

[2]Otherwise on a system where a variable of type *double* takes 8 Byte, an adjacency matrix for 42,000 nodes would need $8 \cdot 42,000^2$ Byte = 14 GB of memory and therefore not fit into main memory of a typical desktop PC

in the sparse matrix as proposed by [GMS92]. This technique has been used in the implementation of all three graph models, leading to much faster solutions then just implementing the model in a naive way.

The implementation of the generation of random graphs where each edge is added with a fixed probability can be found in the file "*generation/random_graph.m*", while the implementation of the scale-free and small-world models can be found in "*generation/scale_free.m*" resp. "*generation/small_world.m*". For further details see the fully commented implementations and the references describing the underlying models.

In total, we created six graphs, a small and a large one for each of the three types of examined random graph models. As these graphs are fixed for all simulations, we further wrote a wrapper function that returns the graph specified in its arguments. It can be found in "*utility/get_graph.m*". Once a graph has been created, it is saved to a file. The next time this wrapper is called for the same parameters, it loads and returns the already generated graph and therefore avoids the possibly time-consuming process of constructing the graph again. Furthermore this wrapper function saves both the random stream and its state for later verification that the used graphs really have been generated using the described functions. The wrapper function also works for the real world graphs for which it simply returns the graph that is loaded from disk.

4.2 Statistics on the graphs

For calculation of characteristic values of a graph like the average number of edges per node, the average shortest path or the clustering coefficient, we wrote the Matlab function *"statistics/print_statistics.m"*:

statistics/print_statistics.m

```
1  function print_statistics(A)
2  % Print statics for a undirected, loop-free graph.
3  %
4  % INPUT
5  % A: [n n]: adjacency matrix
6
7  n = size(A, 1);
8  fprintf('number of nodes = %d\n', n);
9
10 m = nnz(A);
11 assert(mod(m, 2) == 0);
```

```
12 fprintf('number of (undirected) edges = %d\n', m/2);
13
14 k = m/n;
15 fprintf('average node degree = %.4f\n', k);
16
17 maxDeg = max(full(sum(A)));
18 fprintf('maximum node degree = %d\n', maxDeg);
19
20 avgPathLength = average_path_length(A);
21 fprintf('average path length = %.4f\n', avgPathLength);
22
23 clusterCoeff = global_clustering_coefficient(A);
24 fprintf('global clustering coefficient = %.4f\n', clusterCoeff);
25
26 end % print_statistics(...)
```

To compute the *average path length* [Wik11a] and the *global clustering coefficient* [Wik11c] we wrote additional Matlab functions that can be found in "*statistics/average_path_length.m*" and "*statistics/global_clustering_coefficient.m*". To speed up computations, both implementations are using vectorization. Parallelizing loops can improve things even more. In both implementations the outermost loop is running from 1 to n and sums up over some values that are rather costly to compute but free of side-effects. The following code illustrates how this code is accelerated using Matlab's Parallel Computing Toolbox:

Listing 1: Usual for-loop

```
1 acc = 0;
2 for i=1:n
3     acc = acc + f(i);
4 end
```

Parallelized for-loop

```
1 parfor i=1:n
2     a(i) = f(i);
3 end
4 acc = sum(a);
```

Even for our small graphs with $1,000$ nodes, on a dual core machine, this technique accelerated the calculation of both the average path length and the clustering coefficient by a factor of about 1.7.

4.3 Implementation of employed agent model

For the reasons provided in the introduction of this chapter, this is the most time-critical part of the implementation. There is a straightforward implementation of the simulation part that would look like the following:

```
for t=1:maxTime
    a = randomAgent();
    b = randomNeighbor(a);
    updateOpinion(a, b);
end
```

This code won't run very fast, as every step of the simulation is dependent on the previous time step. Furthermore, there a are hidden conditional statements in *"updateOpinion(a, b)"*.

As explained before, the simulation has to be repeated several times using the same parameter setting, in particular on the same graph. This leads to a simple but effective optimization of the above given pseudo-code: instead of running a single simulation, run m simulations simultaneously. In regard to the above code, one should consider the variables a and b as vectors holding m entries instead of a single one. The *tic-toc*-plot in Figure 3 shows the influence of m on the overall running and the average running time per simulation for our final implementation:

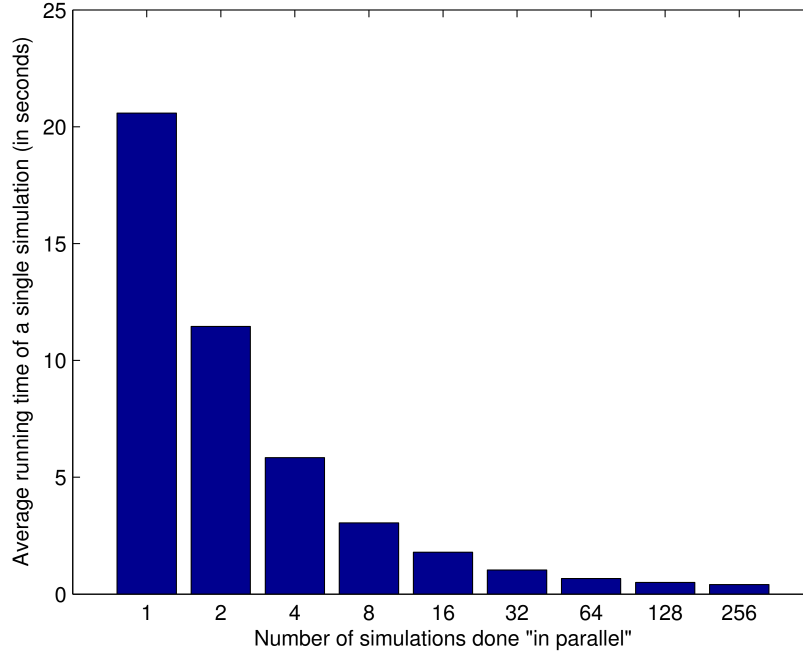

Figure 3: Impact of the parameter m, i.e. the number of simulations run "in parallel", on the average running time of a single simulation (minimum tic-toc-timing over 3 runs using *Simmons81* graph and $755,000$ time steps)

Instead of vectorizing the time-loop, there are m different simulation steps performed at every time step. These m steps are done only using vector operations, resulting in a great improvement of the average running time per simulation. Moreover, the final code updates the opinions without any conditional statements. For a detailed explanation see the comments in the provided Matlab-code. The final simulation code can be found in "*simulation/simulate.m*". A simplified version in pseudo-code is shown in Figure 4.

```
Simulate
convert the graph represented by a sparse matrix
to a data structure that allows efficiently choosing
u.a.r. among a node's adjacent nodes  (→ simulation/convert_graph.m)
for each of the m simulations
   choose n0 employed agents
   set the employed agents' opinion to +1
   choose the normal agents' opinion u.a.r.
for t = [1:dt:maxTime]
   for each of the simulations
      choose dt pairs of agents to meet
       in the next dt time steps      (→ simulation/generate_meetings.m)
   for i = [1:dt]
      simulate meeting at time t+i-1  (→ simulation/vectorized_simulate.m)
      update consensusTimeInformation
   if (consensus reached in all simulations) or (accumulated absolute
    change of opinion in the last dt steps is negligible)
      break
end
--------------------------------------------------------------------------
simulate m simulations in "parallel",
check conditions for early break every dt time steps.
```

Figure 4: Pseudo-code for *simulation/simulate.m*

Similar to what we did for the generation of the graphs, we wrote a wrapper function for the simulation that can be found in "*simulation/simulate_and_store.m*". This wrapper function writes the values returned by *simulate(...)* to a file. Furthermore it saves both the random stream and its state to guarantee reproducibility.

We don't make explicitly use of Matlab's Parallel Computing toolbox for the simulation. Instead the overall running time is decreased by running several independent instances of *simulate(...)* in parallel.

5 Simulation Results and Discussion

5.1 Experimental setup

In the experimental setup, some parameters will be fixed. One reason for fixing them is that running the simulations for varying parameters in too many dimensions would be too time consuming. Furthermore it would be much harder to visualize respectively extract results having that many variables.

We define a basic setup: a simulation is run on a fixed network, an undirected graph of n nodes representing n agents that can communicate if and only if they are connected by an edge. For an overview of the graphs used in the simulations see table 2. As described in section 3.3 the population of agents is partitioned into two groups: there are n_o employed agents and $n - n_0$ normal agents. At the beginning of each simulation the n_0 employed agents are chosen randomly. Each normal agent has an initial opinion that is chosen uniformly at random from $[-1, 1]$, while – according to our model – employed agents have an initial opinion equal to $+1$. The simulation is run for a maximal number of time steps t_{max}. We choose $t_{max} := 500 \cdot n$, i.e. on average every agent will communicate $1,000$ times with one of its neighbors. In each time step two randomly chosen agents meet and update their opinion, see section 3.3.

There are two characteristics we will analyze: the average opinion of the whole population (i.e. including the employed agents) after t_{max} time steps and the number of time steps it takes until "consensus" is reached. In [AA10] consensus is defined as *"Interpreting the opinion of each agent as his agreement (positive value) or disagreement (negative value) on a subject, the consensus can be supposed as a situation where most of the individual in the society agree or disagree on that subject. We say consensus is reached when more than 90% of the society either all have positive opinions (agree) or all have negative opinions (disagree)."* We use a slightly modified definition, only considering agreement, i.e. consensus is reached if at least 90% of the community have positive opinions.

Graph	n	m	k	k_{max}	$avgSP$	c
Random	1 000	20 169	40.34	65	2.15	0.0406
	3 5000	2 901 630	82.90	120	2.82	0.0024
Scale-free	1 000	19 790	39.58	205	2.16	0.1048
	35 000	2 658 518	75.96	1743	2.74	0.0118
Small-world	1 000	20 000	40.00	53	2.41	0.3240
	35 000	1330 000	76.00	93	2.95	0.3133
Caltech36	762	16 651	43.70	248	2.34	0.4091
Reed98	962	18 812	39.11	313	2.46	0.3184
Simmons81	1 510	32 984	43.69	300	2.58	0.3166
UF21	35 111	1 465 654	83.49	8246	2.93	0.2212
Texas84	36 364	1 590 651	87.49	6312	2.90	0.1937
Penn94	41 536	1 362 220	65.59	4410	3.13	0.2118

Table 2: Characteristics of graphs used in simulations: n: number of nodes, m: number of edges, k: average nodal degree, k_{max}: maximal nodal degree, $avgSP$: average shortest path length, c: global clustering coefficient

5.2 Influence of the parameters μ and u

For the Facebook network *Simmons81* (which has $1,510$ nodes) we choose 5% of the agents to be employed agents, i.e. 76 agents are selected. The model parameters μ and u are both varied between 0.1 and 0.5. Figure 5 resp. 6 show the average opinion achieved after $500 \cdot n = 755,000$ and $4,000 \cdot n = 6,040,000$ time steps. Furthermore the number of time steps until consensus is reached is shown in Figure 7. These and all future simulations are all averaged over 32 runs, if not stated otherwise. Note that Figure 7 is the result of simulating at most $8,000 \cdot n$ time steps; nevertheless for small values of μ and large values of u (e.g. $\mu = 0.1$, $u = 0.5$) there is at least one simulation where no consensus is reached after simulating $12,080,000$ time steps.

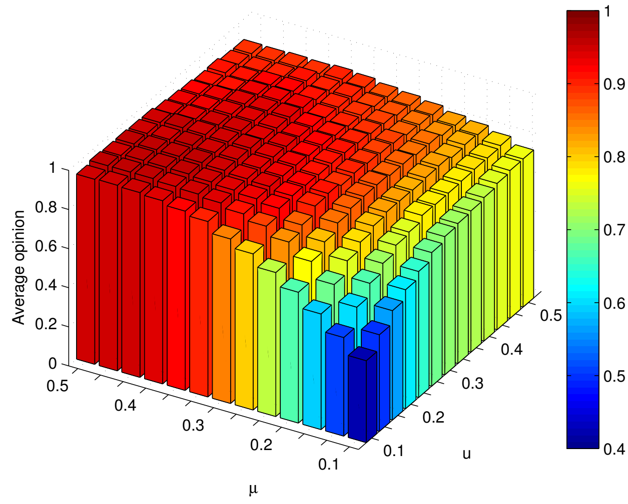

Figure 5: Average opinion after $t_{max} = 500 \cdot n = 755,000$ time steps for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph

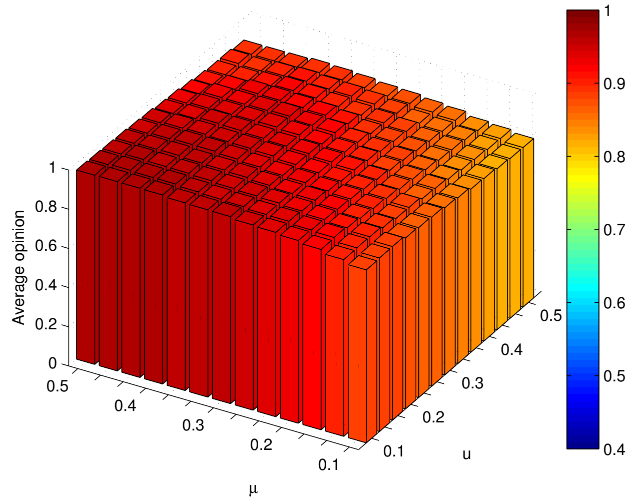

Figure 6: Average opinion after $t_{max} = 4,000 \cdot n = 1,510,000$ time steps for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph

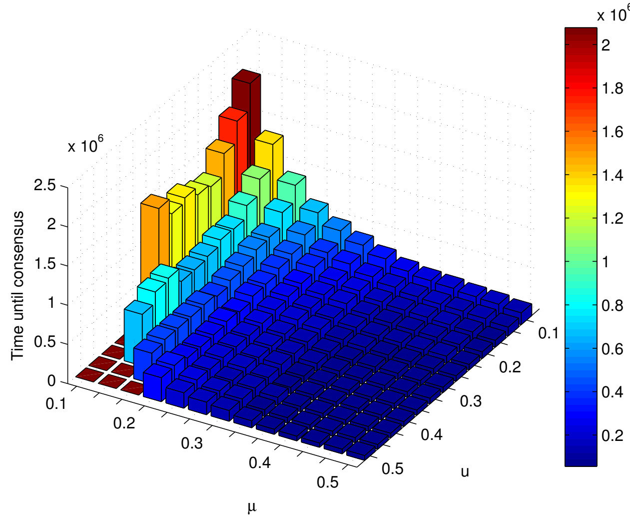

Figure 7: Time until consensus is reached for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph and $t_{max} = 8,000 \cdot n = 12,080,000$

In Figure 6 one can see that for the *Simmons81* Facebook network and a large t_{max} decreasing values of u increase the average opinion at time t_{max}, while increasing values of μ also increase the final average opinion, i.e. the average opinion and u are negatively correlated, while the average opinion and μ are positively correlated.
Furthermore, comparing Figure 5 and 6 for $\mu \geq 0.25$ there is no big difference in the average opinion after $500 \cdot n$ and $4,000 \cdot n$ time steps. Moreover, averaged over 32 simulations for $\mu \geq 0.25$ consensus is reached in less than $500 \cdot n$ steps (Fig.7). At least for $n_0 = 0.05 \cdot n$ and $\mu > 0.2$ a larger t_{max} would not change the achieved result significantly. The same simulations (but using less than 32 repetitions) gave similar results using the other real-world graphs. Note that for different choices of n_0 a maximal achievable average opinion might not be reached within $t_{max} = 500 \cdot n$ time steps, but that is not a problem: one might interpret t_{max} just as the number of times two agents communicate in a fixed period of time, e.g. in the last six months before an election an agent on average communicates with $2 \cdot 500$ other agents.

The parameters μ and u are now being fixed: $\mu = 0.25$ and $u = 0.5$. The convergence factor μ can be interpreted as follows: let two normal agents A and B with opinions $x_A < x_B$ meet. If their opinions differ by $\Delta = x_B - x_A < u$, then after interacting, A changes her opinion to $x_A + 0.25 \cdot \Delta$, while B has $x_B - 0.25 \cdot \Delta$ as his new opinion. That is their opinion will then differ by $0.5 \cdot \Delta$, i.e. the difference

between their opinions is halved. An uncertainty level of $u = 0.5$ has the effect that given an uniform distribution of the opinions (discarding employed agents), an agent with opinion x_A that satisfies $-0.5 \leq x_A \leq 0.5$ will in expectation be influenced by half of his neighbors.

Table 3 shows the parameters that have been fixed. In Figure 8, 9 and 10 one can finally see a comparison for different values of μ and u for the evolution of the average opinion resp. the fraction of people with positive opinion for a varying number of employed agents n_0.

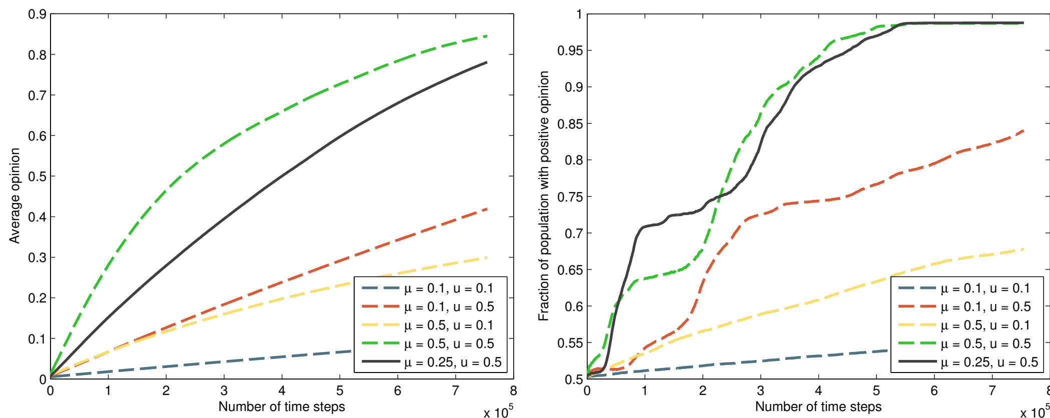

Figure 8: Average opinion and fraction of population with positive opinion for $n_0 = 0.01 \cdot n = 15$ using *Simmons81* graph

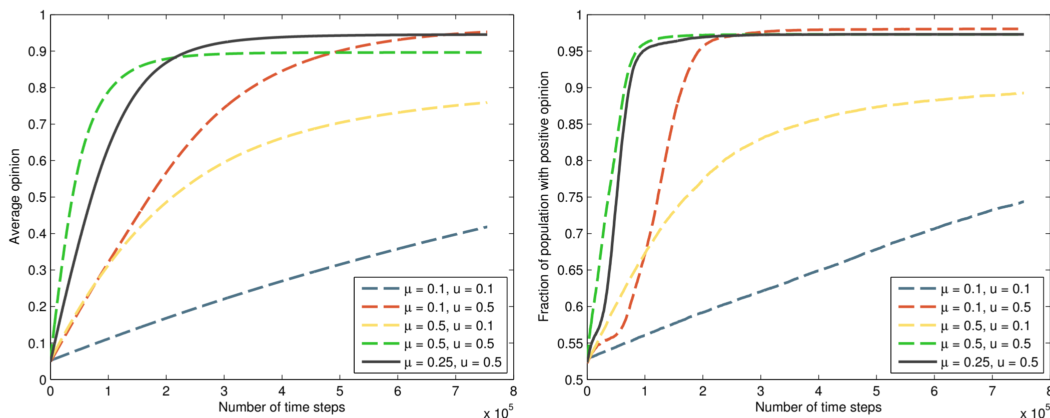

Figure 9: Average opinion and fraction of population with positive opinion for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph

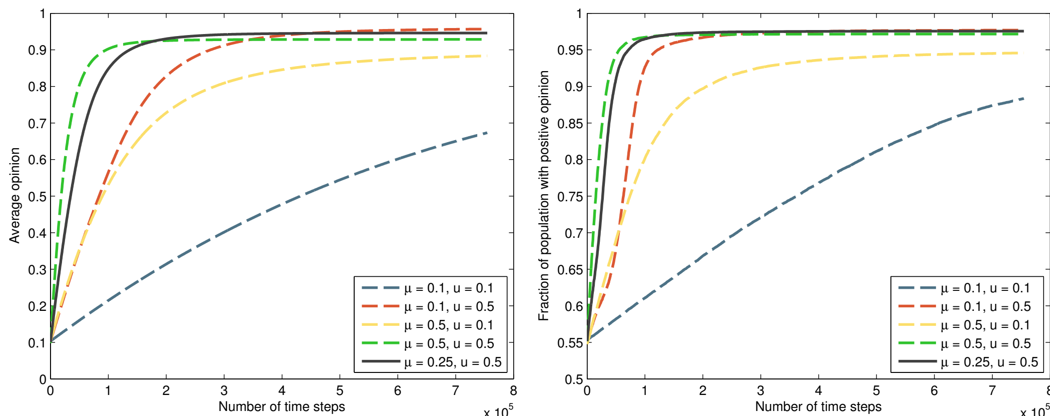

Figure 10: Average opinion and fraction of population with positive opinion for $n_0 = 0.10 \cdot n = 151$ using *Simmons81* graph

5.3 Influence of the number of employed agents

The parameters μ, u and t_{max} are fixed for the following simulations. Table 3 gives a summary. We run simulations for different values for n_0, i.e. number of employed agents, to investigate to which extend more employed agents yield to a higher average opinion or an earlier consensus time.

Parameter		Value
Convergence parameter	μ	0.25
Uncertainty level	u	0.5
Maximum number of timesteps	t_{max}	$500 \cdot n$

Table 3: Fixed parameters for following simulations

Figure 11 shows the final average opinion after t_{max} time steps for varying fractions of employed agents for both small and large networks. The fraction of employed agents has been varied between 0% and 2%. Both small and large random and real-world networks follow the same trends. An increasing fraction of employed agents implies a higher average opinion. The difference between the average opinion observed for a fixed fraction of employed agents is rather small among the various random graphs. Moreover, the average opinion achieved in the real-world networks is slightly below the average opinion investigated for the random graphs. Increasing the fraction of

employed agents to more than 1.5% doesn't yield to a considerably enhancement in the final average opinion of the population.

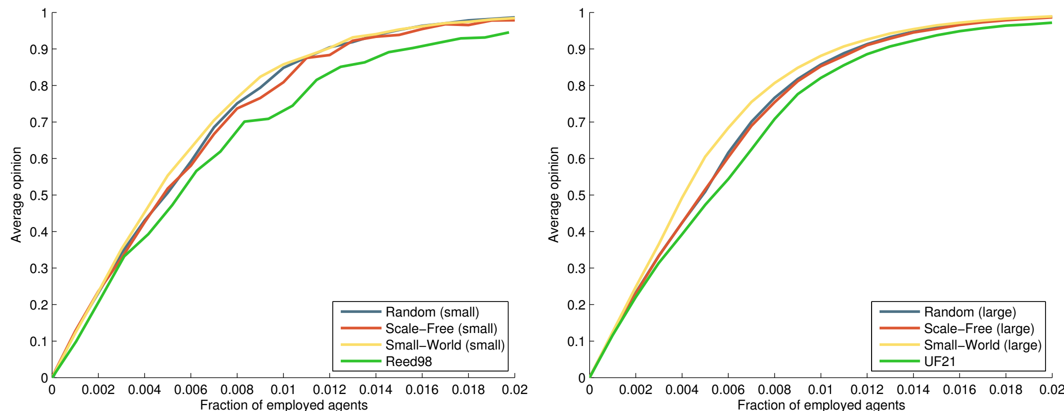

Figure 11: Average opinion after $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents

In Figure 12 the percentage of simulations that did not reach consensus after maximal t_{max} time steps for varying fractions of employed agents is shown. The maximal considered fraction of employed agents has been chosen such that for all graphs of the respective size and for all simulations consensus is reached. That is, the maximal considered fraction is 1.5% for the small and 0.5% for the large networks. For large real networks, especially large real-world networks, only a smaller fraction of employed agents is needed to achieve consensus.
Different network types behave differently. In particular, the small-world graphs tend to reach consensus for much smaller fractions of employed agents, while for the real-world graphs on average a higher fraction of employed agents is needed to decrease the number of simulations without consensus.

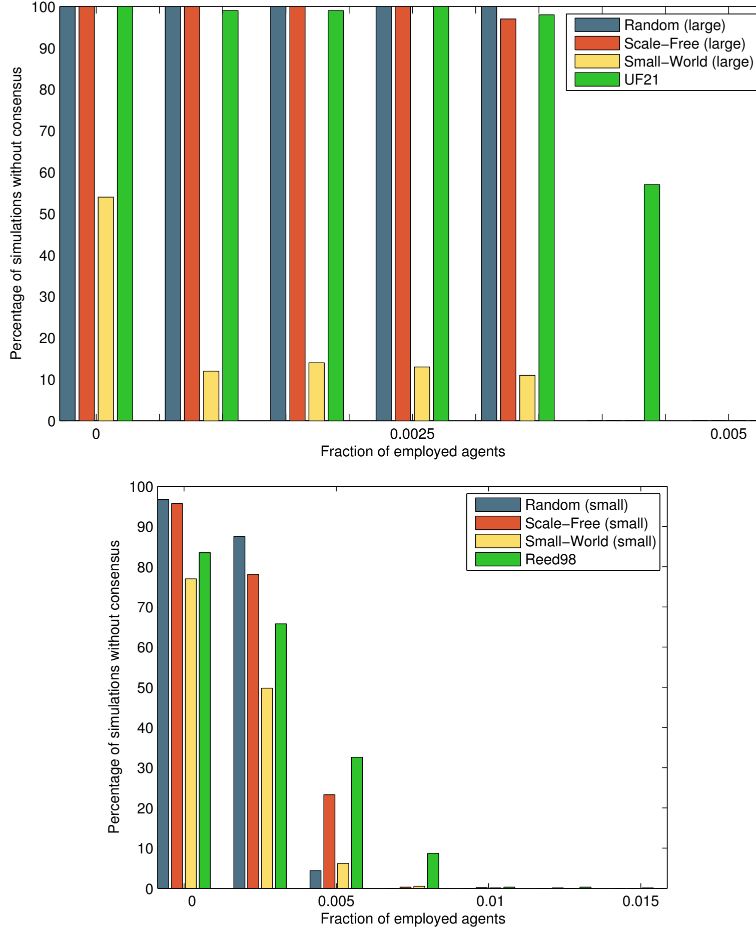

Figure 12: Percentage of simulations that did not reach consensus after maximal $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents (averaged over $1,000$ simulations for the small and 100 simulations for the large graphs)

A reason for the small-world graphs to behave differently might be that their clustering coefficient is significantly higher than the clustering coefficient of the other random networks. This seems to lead to a community structure which favors opinion formation, i.e. at the end of the process the majority of agents hold the same opinion. Moreover, for the real-world networks a higher fraction of employed agents is needed to achieve consensus compared to the random graphs. Although, those real-world networks have a higher cluster coefficient than both the random and the scale-free networks. A reason might be the distribution of the nodal degrees of the considered

real-world networks (Fig. 13 and 14). For example the small graph *Reed98* has 17.6%, the large graph *UF21* has 9.6% nodes with degree less then 10. In contrast such low-degree nodes don't exist in the considered random graphs. Having only a small number of neighbors, there are on average even less neighbors holding an opinion within the uncertainty level. Therefore an agent with just one neighbor might never change its opinion in the whole process of opinion formation.

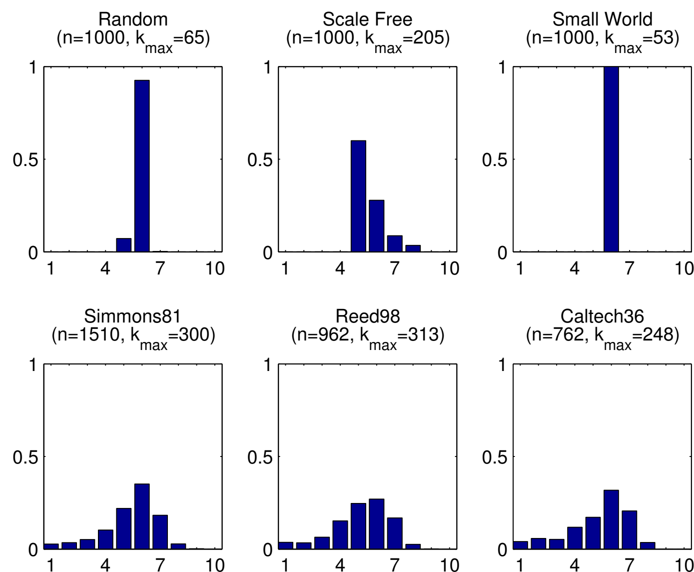

Figure 13: Histograms of grouped nodal degrees for small graphs: node k is in group i if $2^{i-1} \leq \deg(k) < 2^i$

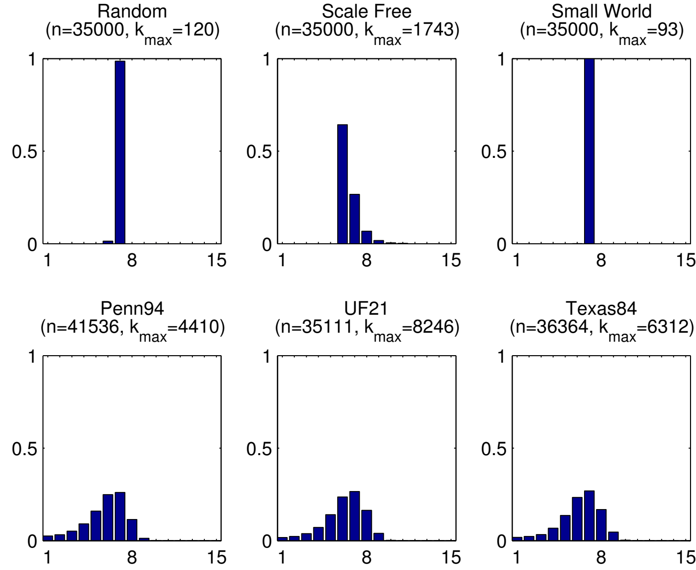

Figure 14: Histograms of grouped nodal degrees for large graphs: node k is in group i if $2^{i-1} \leq \deg(k) < 2^i$

Figure 15 shows the average time until consensus for varying fractions of employed agents. The lowest considered fractions of employed agents has been chosen such that consensus is reached in all simulations. With a higher fraction of employed agents the number of interactions needed until consensus is reached decreases, independently of the graph type. For a fixed fraction of employed agents, small-world networks need the least number of interactions to reach consensus, whereas in the real-world networks the most number of interactions are needed. Possible reasons mentioned above.

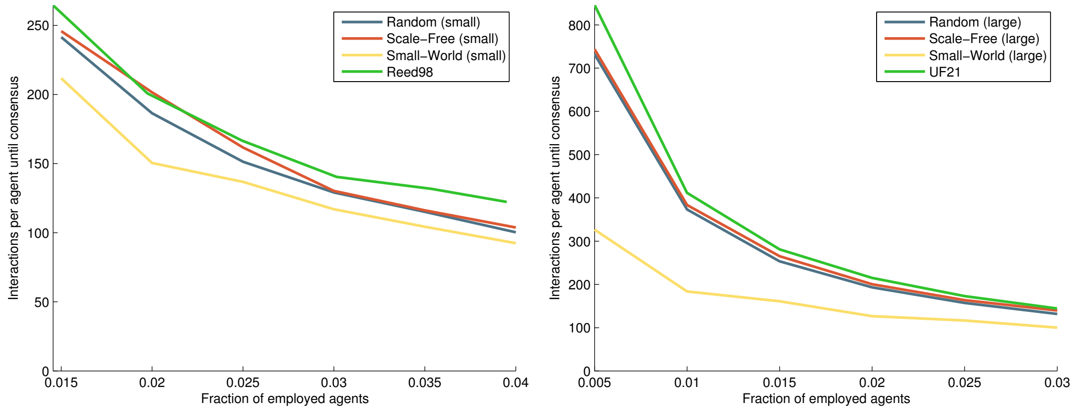

Figure 15: Average time until consensus for varying fractions of employed agents

Figure 16 and 17 illustrate how the average opinion and the fraction of population with positive opinion changes over time for varying fractions of employed agents. Scenarios with 0.5% resp. 1% employed agents are compared for the large networks. For a fixed average opinion one observes that in the setting with 1% employed agents it takes only about half the time to achieve this opinion compared to a setting with just 0.5% employed agents.

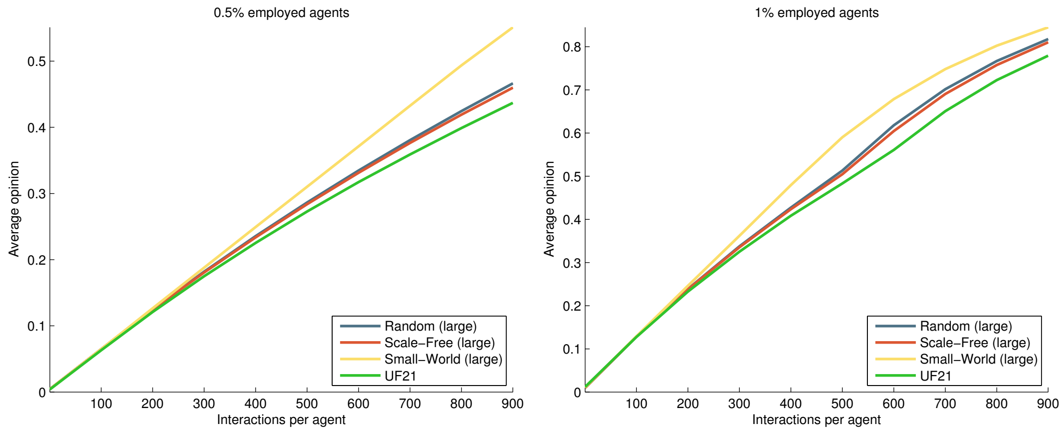

Figure 16: Propagation of opinion: interactions per agent against average opinion for 0.5% and 1% employed agents

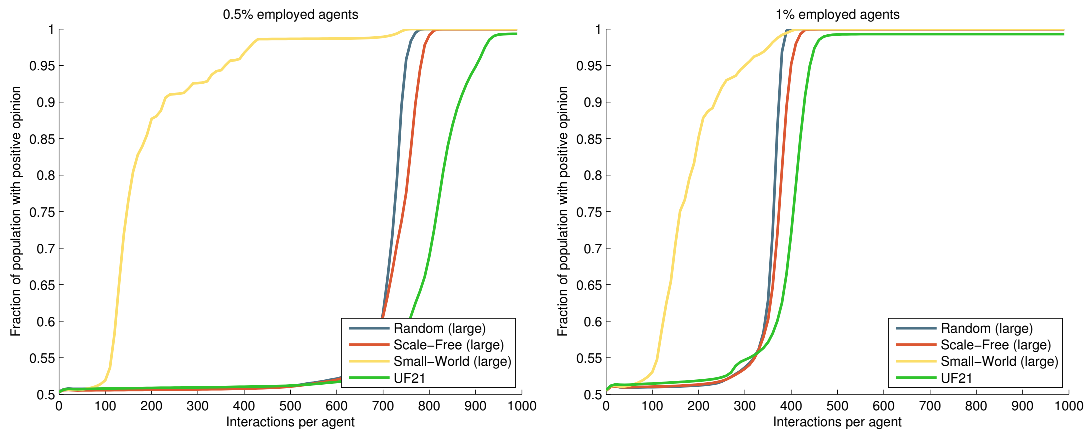

Figure 17: Propagation of opinion: interactions per agent against fraction of population with positive opinion for 0.5% and 1% employed agents

By doubling the fraction of employed agents from 0.5% to 1%, the number of time steps needed to reach consensus is halved. The reason for the large slope (around time step 400) increasing the fraction of population with positive opinion abruptly from less than 0.55 to more than 0.95 is illustrated in Figure 18. It shows the result of one single simulation for the *UF21* graph. After 400 time steps per agent only about one half of the population has a positive opinion. Most agents of the other half have a negative opinion very close to zero. Being influenced by the agents with positive opinion, their opinion is slightly increased and therefore the overall fraction of agents with positive opinion increases rapidly.

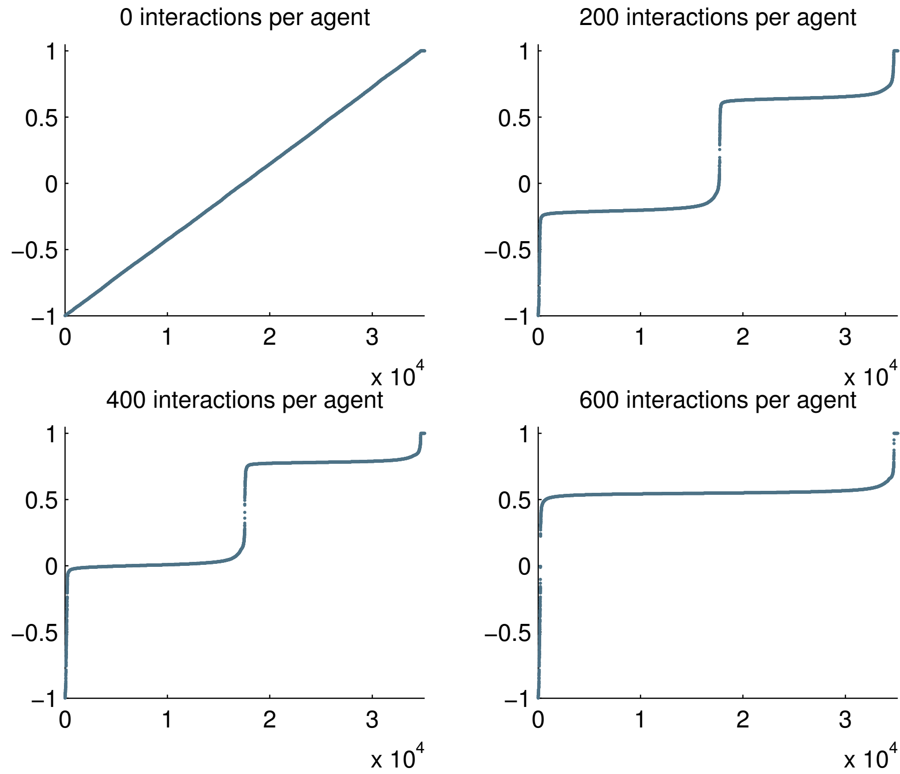

Figure 18: Distribution of opinion after 0, 200, 400 and 600 interactions per agent for one single simulation and 1% employed agents using *UF21* graph (x-axis: agents ranked by their opinion, y-axis: opinion)

In Figure 19 a comparison of the results for small and large real-world networks can be seen: Both the average opinion and the number of interactions until consensus behave similarly for small and large graphs, that is there is no major difference observable.

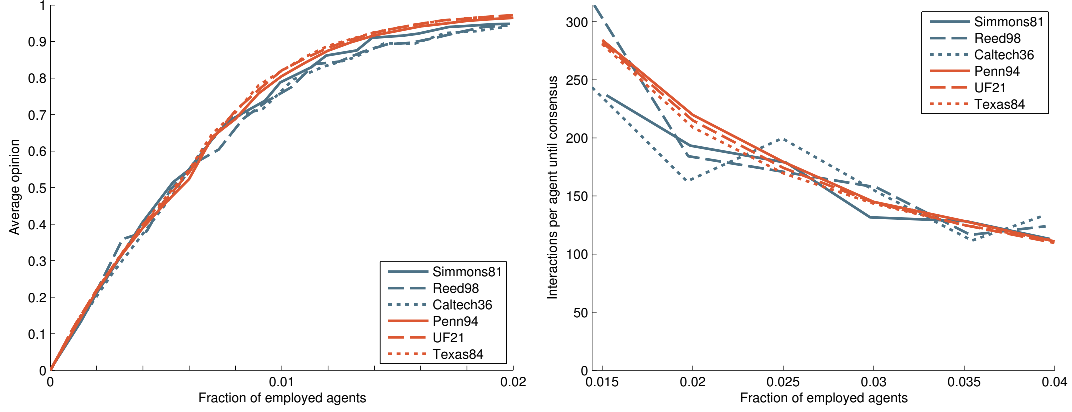

Figure 19: Direct comparison between results for small and large real-world networks: average opinion after t_{max} time steps and average time until consensus for varying fractions of employed agents

5.4 Impact of the selection strategy for employed agents

Investigating the development of the average opinion over time and the standard deviation for different time steps (Fig.20), one observes that the standard deviation for the scale-free and the real-world networks is greater than for the other network types. One possible reason might be the wide distribution of nodal degrees for these two network types[3]. Therefore, for a fixed fraction of employed agents, a strategy for selecting them may yield to a better final average opinion and consensus might be reached faster. A simple strategy for choosing n_0 employed agents out of n agents is to select the agents with the most neighbors. This strategy will be referred as "greedy strategy". For example when hiring employed agents, a party would probably prefer an agent having many social contacts over an "outsider".

[3]Scale-free networks show a power law degree distribution [Wik11d], the distributions of the real-world networks follow similar trends

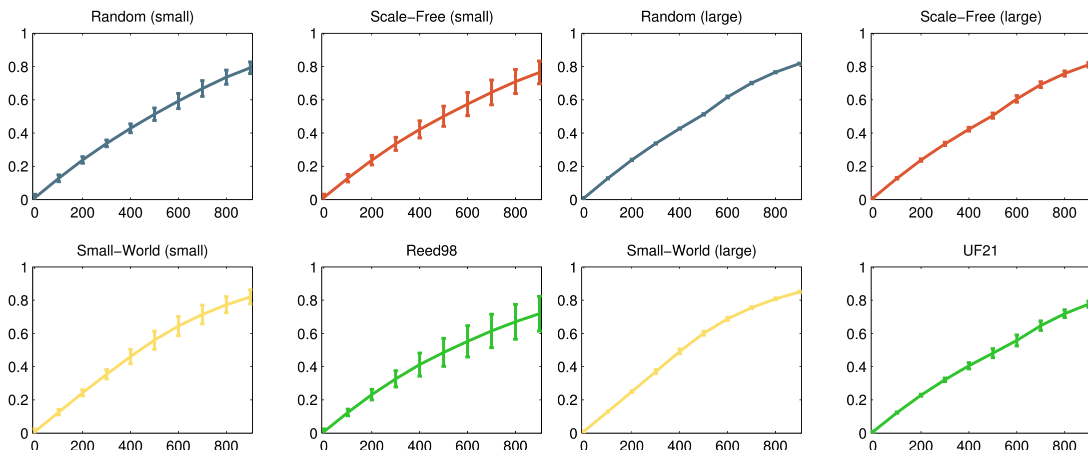

Figure 20: Propagation of opinion: interactions per agent against average opinion for 1% employed agents including vertical bars illustrating the standard deviation

In the following the previously described greedy strategy is compared with the random selection of employed agents (Fig. 21, 22, 24). The greedy strategy indeed yields to a large advance for the scale-free and real-world networks: using the new strategy the same (small) fraction of employed agents leads to a much higher average opinion, e.g. using 0.2% employed agents for the real-world network *UF21*, the final average opinion choosing the employed agents at random is 0.2 while the greedy strategy leads to an average opinion of about 0.9 (Fig. 21). In contrast to all considered random networks, using the greedy strategy on the real-world networks doesn't lead to an average opinion that is larger than 0.985 (at least when choosing 10% or less employed agents). More than that, increasing the fraction of employed agents doesn't necessarily increase the average opinion. For example in the *Reed98* network, increasing the fraction from 0.8% to 1.0% (i.e. using 10 employed agents instead of 8) decreases the average opinion from 0.961 to 0.957 (averaged over 1,000 simulations). A possible reason for this small anomaly could be that some agents increase the value of their opinion too fast to propagate their opinion change to their neighborhood. As described before, the wide distribution of nodal degrees is probably the reason why the greedy strategy performs much better for scale-free and real-world networks: choosing the employed agents greedily increases the average number of interactions per employed agent (Fig. 23). For the small-world and the random network the improvement is rather small.

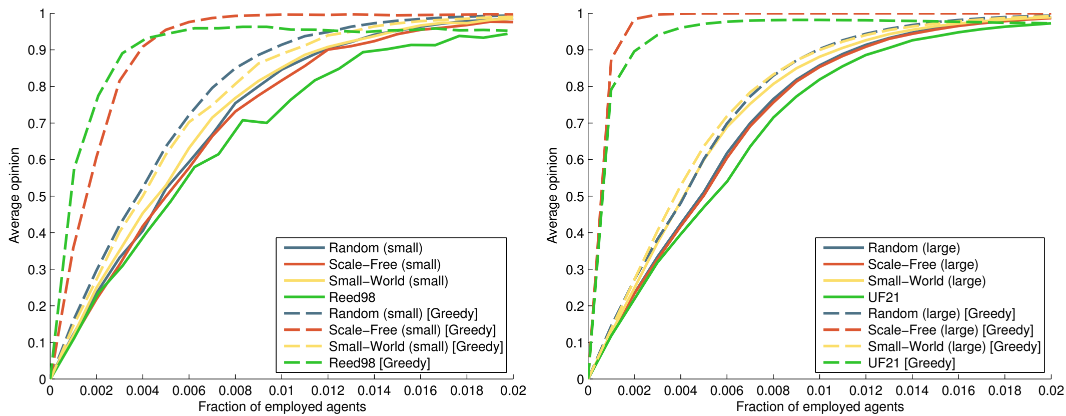

Figure 21: Comparison of greedy selection strategy against random selection of employed agents: average opinion after $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents

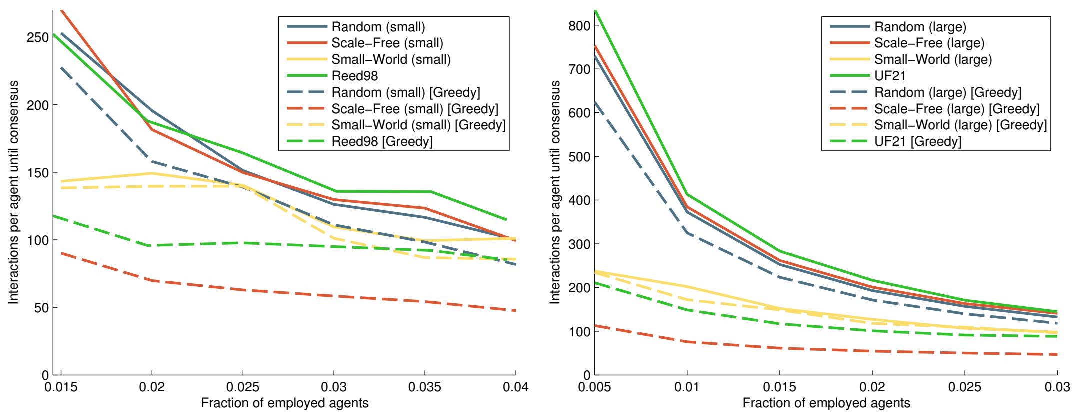

Figure 22: Comparison of greedy selection strategy against random selection of employed agents: average time until consensus for varying fractions of employed agents

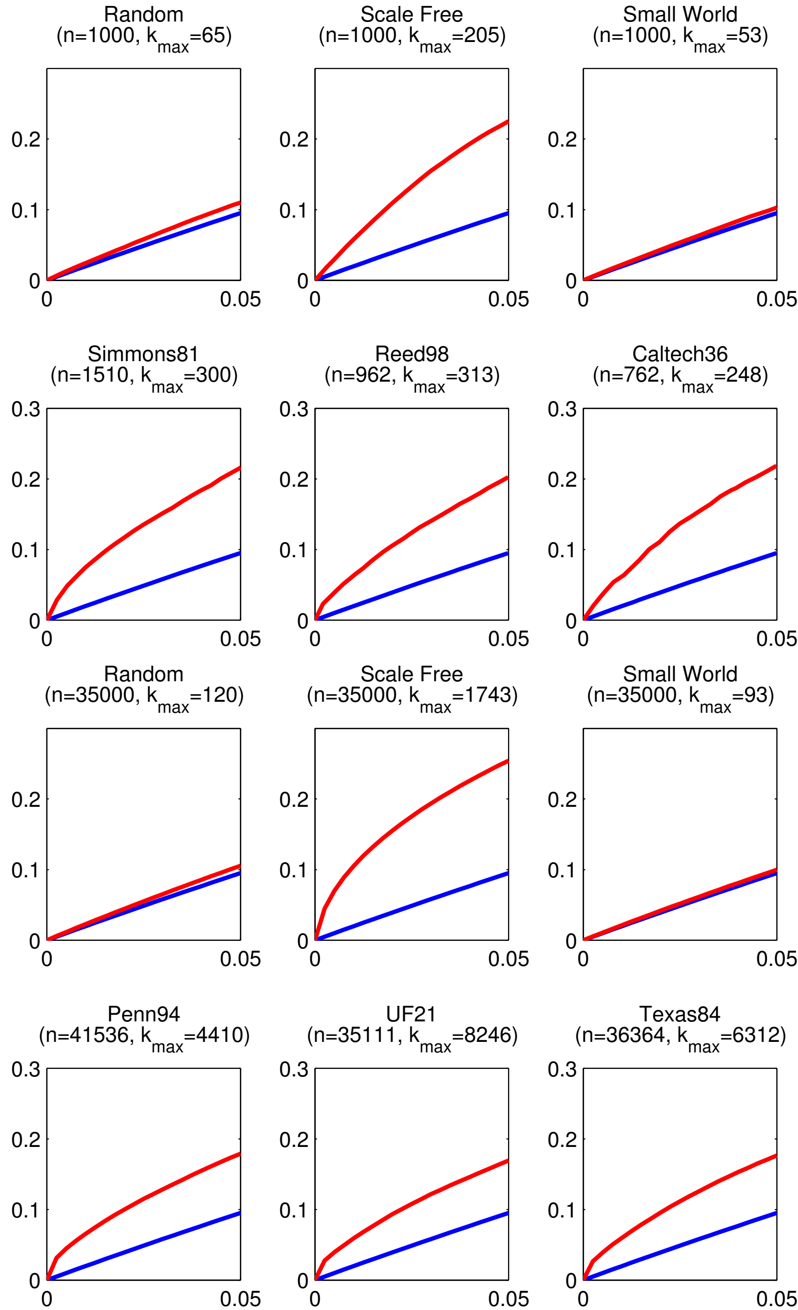

Figure 23: Probability that exactly one employed agent is involved in a single interaction for random selection (*blue*) and greedy selection strategy (*red*). (x-axis: fraction of employed agents, y-axis: probability)

Furthermore the greedy strategy leads for a fixed fraction of employed agents to a smaller number of interactions per agent until consensus. Again, the most significant decrease can be observed for the scale-free and the real-world networks. Comparing the percentage of simulations without consensus for small fractions of employed agents, one can see in Figure 24 that for 0.25%, i.e. 2 employed agents in the *Reed98* graph consensus is reached in all simulations using the greedy strategy. Notably using just a single employed agent (i.e. 0.10%) there were only 43 out of $1,000$ simulations (i.e. 4.3%) where no consensus was reached. Similarly to the small real-world network *Reed98*, in the large *UF21* graph the smallest fraction of employed agents for which all simulations reached consensus could be reduced from 0.5% to about 0.083%. Finally, Figure 25 illustrates the distribution of opinion as a function of time. In both plots two main opinion streams emerge after a short period of time. The opinions of the two streams drift towards $+1$ and join to one single main stream. The greedy strategy accelerates this process.

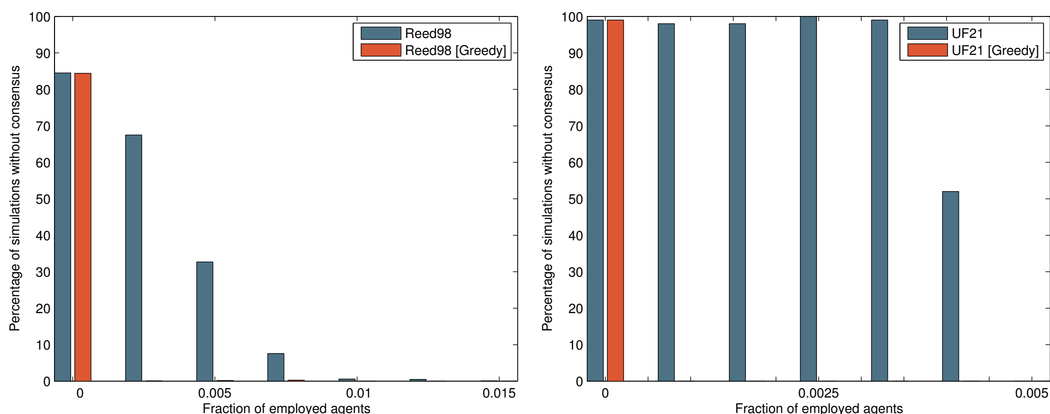

Figure 24: Comparison of greedy selection strategy against random selection of employed agents: percentage of simulations that did not reach consensus after maximal $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents (averaged over $1,000$ simulations for the small and 100 simulations for the large graphs)

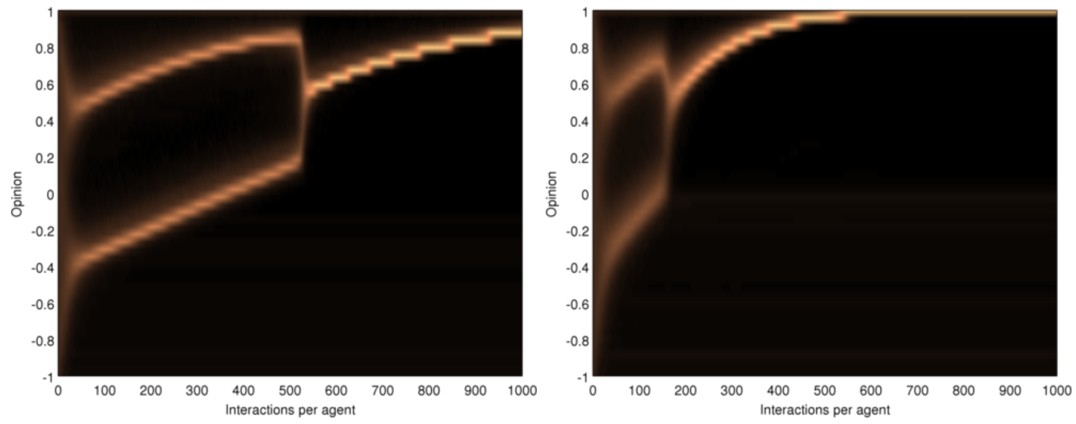

Figure 25: Distribution of opinion as a function of time for choosing employed agents randomly (left) and greedily (right). Brighter colors indicate more agents.

6 Summary and Outlook

In this report we described a continuous and network-based model for opinion formation with employed agents. Employed agents are individuals which advance a particular view and argue for that view in their neighborhood. Even a small fraction of employed agents is able to convince the majority of their opinion. The strategy for selecting the employed agents has an large impact on the propagation of opinions. The greedily chosen employed agents cause a higher final average opinion and consensus is reached faster. Using the greedy selection strategy, employed agents turned out to be quite influential. A greedily chosen single employed agent was able to cause consensus in a real-world network consisting of about $1,000$ agents.

Real-world and different types of random networks were compared using the considered model. Following a power-law distribution for its nodal degrees, the *scale-free* networks yielded to a good approximation for the real-world graphs. The lack of clustering compared to the real-world networks didn't become noticeable in our simulations. In contrast, the *small-world* graphs have higher clustering coefficients. Together with the small variance of nodal degrees, consensus was reached faster than in the real-world networks. Moreover, the considered *random* networks (generated using the Erdős-Rényi model) have low clustering and a small variance of nodal degrees. Compared to the networks having a wider degree distribution, the greedily selection of employed agents leaded to a smaller improvement. All analyzed random graphs have a small average shortest path length, similar to the real-world graphs. Although recognizing differences in these values we couldn't spot a relation to the different results. Finally, none of the considered random models produced nodes having low degrees as observed in the real-world networks. This was noticeable, as agents corresponding to those nodes can hardly be influenced.

As a possible interesting extension to our proposed model, one could allow several groups of employed agents, which compete against each other. For example, one could consider two groups of employed agents, where one group tries to influence the community's opinion towards $+1$ and the other towards -1. Moreover, one could add some properties like *social force*, *self force* and *goal force* to the agents, as described in [AA10]. In combination with a more fine grained update rule, the model would allow a more sophisticated characterization of individual agents. For instance, the model could distinguish between opinion leaders and ordinary people, which change their opinion more rapidly. Furthermore, we code assign weights to the connections between agents, as this could possibly better characterize friendships in social online networks. For example in the Facebook network, a person having a dozen friends might have greater influence on one of her friends' opinion than a

person having thousands of people on his friend list. Finally, different strategies for choosing employed agents could be analyzed. One might think of a strategy not only using local information of the network (like the node degree) but also taking the network's topology into account.

7 Appendix

7.1 References

[AA10] Mohammad Afshar and Masoud Asadpour. Opinion formation by informed agents. *Journal of Artificial Societies and Social Simulation*, 13(4):5, 2010.

[BA99] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[GMS92] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1992.

[Joh] Richard Johnson. Matlab programming style guidelines. `http://www.datatool.com/downloads/matlab_style_guidelines.pdf`.

[LZ04] Abramson G. Laguna, M. F. and D. H. Zanette. Minorities in a model for opinion formation. *Complexity*, 9:31–36, 2004.

[Mat] MathWorks. Code vectorization guide. `http://www.mathworks.com/support/tech-notes/1100/1109.html`.

[TKMP10] Amanda L. Traud, Eric D. Kelsic, Peter J. Mucha, and Mason A. Porter. Comparing community structure to characteristics in online collegiate social networks. SIAM Review, in press (arXiv:0809.0960), 2010.

[Wik11a] Wikipedia. Average path length — Wikipedia, the free encyclopedia, 2011. [Online; accessed 10-May-2011].

[Wik11b] Wikipedia. Erdös-Rényi model — Wikipedia, the free encyclopedia, 2011. [Online; accessed 10-May-2011].

[Wik11c] Wikipedia. Global clustering coefficient — Wikipedia, the free encyclopedia, 2011. [Online; accessed 10-May-2011].

[Wik11d] Wikipedia. Scale-free network — Wikipedia, the free encyclopedia, 2011. [Online; accessed 23-May-2011].

[WN02] Deffuant G. Amblard F. Weisbuch, G. and J.-P. Nadal. Meet, discuss and segregate. *Complexity*, 7(3):55–63, 2002.

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of “small-world” networks. *Nature*, 393:440–442, 1998.

7.2 List of Figures

1 Degree distribution for the small networks in log-log scale (x-axis: sorted order of degree, y-axis: fraction of nodes having this degree) . 11
2 Degree distribution for the large networks in log-log scale (x-axis: sorted order of degree, y-axis: fraction of nodes having this degree) . 12
3 Impact of the parameter m, i.e. the number of simulations run ”in parallel”, on the average running time of a single simulation (minimum tic-toc-timing over 3 runs using *Simmons81* graph and $755,000$ time steps) 16
4 Pseudo-code for *simulation/simulate.m* 17
5 Average opinion after $t_{max} = 500 \cdot n = 755,000$ time steps for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph 20
6 Average opinion after $t_{max} = 4,000 \cdot n = 1,510,000$ time steps for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph 20
7 Time until consensus is reached for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph and $t_{max} = 8,000 \cdot n = 12,080,000$ 21
8 Average opinion and fraction of population with positive opinion for $n_0 = 0.01 \cdot n = 15$ using *Simmons81* graph 22
9 Average opinion and fraction of population with positive opinion for $n_0 = 0.05 \cdot n = 76$ using *Simmons81* graph 22
10 Average opinion and fraction of population with positive opinion for $n_0 = 0.10 \cdot n = 151$ using *Simmons81* graph 23
11 Average opinion after $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents 24
12 Percentage of simulations that did not reach consensus after maximal $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents (averaged over $1,000$ simulations for the small and 100 simulations for the large graphs) 25
13 Histograms of grouped nodal degrees for small graphs: node k is in group i if $2^{i-1} \leq \deg(k) < 2^i$ 26

14 Histograms of grouped nodal degrees for large graphs: node k is in group i if $2^{i-1} \leq \deg(k) < 2^i$ 27
15 Average time until consensus for varying fractions of employed agents 28
16 Propagation of opinion: interactions per agent against average opinion for 0.5% and 1% employed agents 28
17 Propagation of opinion: interactions per agent against fraction of population with positive opinion for 0.5% and 1% employed agents 29
18 Distribution of opinion after 0, 200, 400 and 600 interactions per agent for one single simulation and 1% employed agents using *UF21* graph (x-axis: agents ranked by their opinion, y-axis: opinion) 30
19 Direct comparison between results for small and large real-world networks: average opinion after t_{max} time steps and average time until consensus for varying fractions of employed agents 31
20 Propagation of opinion: interactions per agent against average opinion for 1% employed agents including vertical bars illustrating the standard deviation 32
21 Comparison of greedy selection strategy against random selection of employed agents: average opinion after $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents 33
22 Comparison of greedy selection strategy against random selection of employed agents: average time until consensus for varying fractions of employed agents 33
23 Probability that exactly one employed agent is involved in a single interaction for random selection (*blue*) and greedy selection strategy (*red*). (x-axis: fraction of employed agents, y-axis: probability) 34
24 Comparison of greedy selection strategy against random selection of employed agents: percentage of simulations that did not reach consensus after maximal $t_{max} = 500 \cdot n$ time steps for varying fractions of employed agents (averaged over $1,000$ simulations for the small and 100 simulations for the large graphs) 35
25 Distribution of opinion as a function of time for choosing employed agents randomly (left) and greedily (right). Brighter colors indicate more agents. 36

7.3 List of Tables

1 Parameters used for graph generation 11
2 Characteristics of graphs used in simulations: n: number of nodes, m: number of edges, k: average nodal degree, k_{max}: maximal nodal degree, $avgSP$: average shortest path length, c: global clustering coefficient 19
3 Fixed parameters for following simulations 23

7.4 Source code

7.4.1 Simulation

simulation/simulate.m

```
function [opinion, consensusTime, t, intermediateOpinion, employedAgent] = ...
 simulate(A, m, maxT, dt, n0, mu0, u0, consensusFraction, epsilon, ...
 employedAgent, getIntermediateOpinion, breakOnConsensus, action)
% Perform a simulation of the "employed" agent model on a given graph.
% n: [1]: number of agents
%
% INPUT
% A: [n n]: adjacency representation of a graph describing the connections
%  between agents, i.e. A(i, j) is 1 iff there is an edge between node i and j.
%  A is assumed to be symmetric.
% m: [1]: number of simulations done in parallel
% maxT: [1]: maximal number of time steps simulated
% dt: [1]: number of time steps done in a single call to vectorized_simulate
% n0: [1]: number of employed agents, 0 <= n0 <= n.
% mu0, u0: [1] / [m 1]: model constants. Either a single constant is provided
%  and used for every simulation or a vector containing (different) constants
%  providing the u and mu-values for the different simulations
% consensusFraction: [1]: fraction of agents with positive opinion needed for
%  consensus to be reached
% employedAgent: empty / [n0 m]: if employedAgent is empty, for every
%  simulation the employed agents are selected uniformly at random. Otherwise
%  the i-th row should be a subset of size n0 of the numbers 1 to n
%  corresponding to the employed agent in simulation i
% epsilon: [1]: if average absolute change over dt simulation steps is smaller
%  than epsilion, i.e. deltaOpinion / dt < epsilon, in all m simulation then
%  stop the simulation. Use epsilon = -1 for no early break.
% getIntermediateOpinion: [1]: flag to get the intermediate opinions of all m
%  simulations after every dt time steps
% breakOnConsensus: [1]: flag to break when consensus is reached in every
%  simulation
% action: string: eval(action) is executed after each simulation step. This
%  parameter can be used to individually print or animate things without
%  changing the whole function. Note that eval(action) possibly has side-effects
%  that might change the function's behavior.
%
% OUTPUT
% opinion: [m n]: opinion(i, :) gives the final opinion of the agents in
%  simulation i
% consensusTime: [m 1]: consensusTime(i) gives the time consensus is reached in
%  simulation i. If no consensus is reached, consensusTime(i) = Inf.
% t: [1]: number of time steps simulated
% intermediateOpinion: {1 nIntermediateSteps}: if getIntermediateOpinion is set,
%  intermediateOpinion{i} holds a matrix of size [m n] representing the opinion
%  of the agents before the i-th iteration of the while-loop
% employedAgent: [n0 m]: employedAgent(:, i) gives the agents selected as
%  employed agent in simulation i

% get data structure for generation of meetings
[neighbor, nNeighbor] = convert_graph(A);
n = size(A, 1);
```

```
51
52 % initial opinion
53 opinion = 2*rand(m, n)-1; % U(-1, 1)
54
55 % simulation parameters
56 if size(mu0, 1) == 1, mu = repmat(mu0, m, 1); else mu = mu0; end
57 if size(u0, 1) == 1, u = repmat(u0, m, 1); else u = u0; end
58
59 % agent parameters
60 selfOpinionFactor = ones(m, n);
61 otherOpinionFactor = zeros(m, n);
62 opinionBias = zeros(m, n);
63 if isempty(employedAgent)
64     employedAgent = zeros(n0, m);
65     for i=1:m
66         employedAgent(:, i) = randsample(n, n0);
67     end
68 end
69 nPositiveAgent = zeros(m, 1);
70 normalAgentFlag = ones(m, n);
71
72 % select special agents and modify their parameter
73 for i = 1:m
74     opinion(i, employedAgent(:, i)) = ones(1, n0);
75     normalAgentFlag(i, employedAgent(:, i)) = zeros(1, n0);
76     nPositiveAgent(i) = sum(opinion(i, :)>0);
77     opinionBias(i, employedAgent(:, i)) = u(i);
78     selfOpinionFactor(i, employedAgent(:, i)) = zeros(1, n0);
79     otherOpinionFactor(i, employedAgent(:, i)) = ones(1, n0);
80 end
81
82 t = 0;
83 consensusTime = inf(m, 1);
84 if getIntermediateOpinion ~= 0
85     intermediateOpinion = cell(1, ceil(maxT/dt));
86 else
87     intermediateOpinion = {};
88 end
89 whileStep = 0;
90
91 while t < maxT
92     whileStep = whileStep + 1;
93     if getIntermediateOpinion ~= 0
94         intermediateOpinion{whileStep} = opinion;
95     end
96     % generate m * dt pairs of meeting agents
97     [meetingA, meetingB] = generate_meetings(neighbor, nNeighbor, m, dt);
98     % perform simulation
99     [opinion, stepConsensusTime, nPositiveAgent, deltaOpinion] = ...
100      vectorized_simulate(meetingA, meetingB, opinion, mu, u, ...
101      selfOpinionFactor, otherOpinionFactor, opinionBias, normalAgentFlag, ...
102      nPositiveAgent, consensusFraction);
103     t = t + dt;
104     noConsensusSimulation = find(stepConsensusTime==dt+1);
105     stepConsensusTime(noConsensusSimulation) = inf;
106     consensusTime = min(consensusTime, stepConsensusTime+t-dt);
107     eval(action);
108     if (breakOnConsensus && isempty(noConsensusSimulation)) ...
```

```
109            || max(deltaOpinion)/dt<epsilon
110          break;
111      end
112  end
113
114  if getIntermediateOpinion ~= 0
115      intermediateOpinion = intermediateOpinion(1:whileStep);
116  end
117
118  end % simulate(...)
```

simulation/vectorized_simulate.m

```
1  function [opinion, consensusTime, nPositiveAgent, deltaOpinion] = ...
2   vectorized_simulate(meetingA, meetingB, opinion, mu, u, selfOpinionFactor, ...
3   otherOpinionFactor, opinionBias, normalAgentFlag, nPositiveAgent, ...
4   consensusFraction)
5  % "Parallel" simulation of our "employed" agent-model for a fixed size of agents
6  % and precalculated meeting times.
7  % Performing several simulations in parallel has the great advantage that one
8  % can simulate a single time step of the simulation using vectorization
9  % technique.
10 % m: [1]: number of simulations done in parallel
11 % t: [1]: number of time steps
12 % n: [1]: number of agents
13 %
14 % INPUT
15 % meetingA, meetingB: [m t]: agent meetingA(i, j) and meetingB(i, j) meet in
16 % simulation i at time j
17 % opinion: [m n]: opinion(i, j) is the initial opinion of agent j in
18 %  simulation i
19 % mu: [m 1]: mu(i) is the convergence factor in simulation i
20 % u: [m 1]: u(i) is the uncertainty level in simulation i
21 % selfOpinionFactor, otherOpinionFactor, opinionBias: [m n]: opinion of agent j
22 %  in simulation i when meeting other agent is selfOpinionFactor(i, j) *
23 %  selfOpinion + otherOpinionFactor(i, j) * otherOpinion + opinionBias(i, j)
24 % normalAgentFlag: [m n]: normalAgentFlag(i, j) is 1 iff agent j in simulation i
25 %  is a "normal" agent i.e. not an emplyed agent
26 % nPositiveAgent: [m 1] number of agents with positive opinion (at initial time)
27 % consensusFraction: [1] fraction of (normal) agents with positive opinion s.t.
28 %  consensus is reached
29 %
30 % OUTPUT
31 % opinion: [m n]: opinion(i,j) is the opinion of agent j in simulation i after
32 %  performing the simulation
33 % consensusTime: [m 1]: consensusTime(i) gives the number of time steps T taken
34 %  until in simulation i the fraction of agents with positive opinion is greater
35 %  or equal consensusFraction, i.e., 1 <= T <= t, if no consensus is reached
36 %  then T = t+1
37 % nPositiveAgent: [m 1]: number of agents with positive opinion after performing
38 %  the simulation
39 % deltaOpinion: [m 1]: deltaOpinion(i) gives the sum of the absolute values of
40 %  changes of agents' opinion in simulation i
41
42 m = size(meetingA, 1);
43 t = size(meetingA, 2);
```

```
44  n = size(opinion, 2);
45
46  % minOpinion, maxOpinion: [m 1]: minOpinion(i) resp. maxOpinion(i) define
47  %  the maximal and minimal opinion an agent can hold in simulation i
48  minOpinion = -ones(m, 1);
49  maxOpinion = ones(m, 1);
50  deltaOpinion = zeros(m, 1);
51  consensusTime = repmat(t+1, m, 1);
52
53  % calculate one-dimensional indices into [m n]-matrix,
54  % i.e. opinion(i, meetingA(i, j)) == opinion(m*(meetingA(i, j)-1)+1)
55  meetingA = m*(meetingA-1)+repmat((1:m)', 1, t);
56  meetingB = m*(meetingB-1)+repmat((1:m)', 1, t);
57
58  for k = 1:t
59      oldOpinionA = opinion(meetingA(:, k));
60      oldOpinionB = opinion(meetingB(:, k));
61
62      % calculate opinion held of the interacting agents at time step k
63      opinionA = selfOpinionFactor(meetingA(:, k)) .* oldOpinionA ...
64               + otherOpinionFactor(meetingA(:, k)) .* oldOpinionB ...
65               + opinionBias(meetingA(:,k));
66      opinionB = selfOpinionFactor(meetingB(:, k)) .* oldOpinionB ...
67               + otherOpinionFactor(meetingB(:, k)) .* oldOpinionA ...
68               + opinionBias(meetingB(:,k));
69      opinionA = min(opinionA, maxOpinion);
70      opinionA = max(opinionA, minOpinion);
71      opinionB = min(opinionB, maxOpinion);
72      opinionB = max(opinionB, minOpinion);
73
74      % normalAgentA, normalAgentB: [m 1]: 0-1-vector. An entry is 1 iff the
75      % corresponding agent is a normal agent.
76      normalAgentA = normalAgentFlag(meetingA(:, k));
77      normalAgentB = normalAgentFlag(meetingB(:, k));
78
79      diffOpinion = opinionA-opinionB;
80
81      % changeOpinion is primarily (mu .* diffOpinion) but has entries equal to 0
82      % where the absolute value of diffOpinion is smaller than the uncertainty
83      % level (plus a small epsilon for comparing floating point numbers)
84      changeOpinion = mu .* (abs(diffOpinion)<u+1e-9) .* diffOpinion;
85
86      % changeOpinionA, changeOpinionB describe the effective change of agents'
87      % opinion at time step k, i.e. employed agents don't change their opinion
88      changeOpinionA = normalAgentA .* changeOpinion;
89      changeOpinionB = normalAgentB .* changeOpinion;
90
91      newOpinionA = oldOpinionA - changeOpinionA;
92      newOpinionB = oldOpinionB + changeOpinionB;
93
94      % accumulate opinion change
95      deltaOpinion = deltaOpinion + abs(changeOpinionA) + abs(changeOpinionB);
96
97      % update number of agents with positive opinion
98      nPositiveAgent = nPositiveAgent-(oldOpinionA>0)+(newOpinionA>0) ...
99                                     -(oldOpinionB>0)+(newOpinionB>0);
100     % update consensusTime, i.e. set it if consensus is reached for the first
101     % time
```

```
102     consensusTime = min(consensusTime, ...
103                         (nPositiveAgent<consensusFraction*n)*(t+1)+k);
104
105     opinion(meetingA(:, k)) = newOpinionA;
106     opinion(meetingB(:, k)) = newOpinionB;
107 end
108
109 end % vectorized_simulate(...)
```

simulation/convert_graph.m

```
1  function [neighbor, nNeighbor]= convert_graph(A)
2  % Convert an adjacency matrix into a concatenated vector listing all the
3  % neighbors and another vector containing the number of neighbors per node.
4  %
5  % INPUT
6  % A: [n n]
7  %  adjacency matrix
8  %
9  % OUTPUT
10 % neighbor: [m 1]: concatenated list of neighbors, m=nnz(A)
11 % nNeighbor: [n 1]: node i has neighbor(i) neighbors
12 n = size(A, 1);
13 nNeighbor = full(sum(A, 2));
14 [neighbor, ~] = ind2sub(n, full(find(A)));
15 end % convert_graph(...)
```

simulation/generate_meetings.m

```
1  function [meetingA, meetingB] = generate_meetings(neighbor, nNeighbor, m, t)
2  % Given a graph and integers m and t, return two [m t]-matrices meetingA and
3  % meetingB. For 1 <= i <= m and 1 <= j <= t, meetingA(i, j) and meetingB(i, j)
4  % represent a pair of meeting agents. This pair is generated by first choosing
5  % an agent A uniformly at random and afterwards choosing another agent B
6  % randomly from A's neighbors.
7  %
8  % INPUT
9  % neighbor: [m 1]: concatenated list of neighbors as returned by
10 %  convert_graph(A)
11 % nNeighbor: [n 1]: node i has neighbor(i) neighbors as returned by
12 %  convert_graph(A)
13 % m, t: [1]: dimensions of the matrices to be generated
14 %
15 % OUTPUT
16 % meetingA, meetingB: [m t]: matrices representing pairs of meeting agents (as
17 %  described above)
18
19 n = size(nNeighbor, 1);
20 neighborIndex = cumsum([1; nNeighbor]);
21 meetingA = randi(n, m, t);
22 meetingB = zeros(m, t);
23 for i = 1:t
24     meetingB(:, i) = neighbor(neighborIndex(meetingA(:, i)) + ...
25                       floor(rand(m, 1) .* nNeighbor(meetingA(:, i))));
```

```
26 end
27
28 end % generate_meetings(...)
```

simulation/simulate_and_store.m

```
1 function [opinion, consensusTime, t, intermediateOpinion, employedAgent, ...
2  filename] = simulate_and_store(A, m, maxT, dt, n0, mu0, u0, ...
3  consensusFraction, epsilon, employedAgent, getIntermediateOpinion, ...
4  breakOnConsensus, action, graphName)
5 % Wrapper for simulate(...). Perform a simulation of the "employed" agent model
6 % on a given graph and save input and output as well as the random stream and
7 % its state for later processing / reproducing the received results.
8 % The file is written to the directory 'simulationResult', for proper operation,
9 % this folder has to be within the Matlab path.
10 %
11 % INPUT
12 % Same as input to simulate(...). One additional argument has to be provided:
13 % graphName: string: name of the graph represented by A
14 %
15 % OUTPUT
16 % Same as output of simulate(...). One additional variable is returned:
17 % filename: string: the name of the file that input and output have been saved
18 %  to. The name of the output file is a concatenation of graphName and the
19 %  parameters n0, mu0 and u0 being used. To make sure that things work properly
20 %  using the Parallel Computing Toolbox, the filename also includes the worker
21 %  id.
22
23 muStr = 'var';
24 if size(mu0, 1) == 1, muStr = num2str(mu0); end;
25 uStr = 'var';
26 if size(u0, 1) == 1, uStr = num2str(u0); end;
27
28 worker = '';
29 w = getCurrentTask();
30 if ~isempty(w)
31     worker = ['_' num2str(get(w, 'ID'))];
32 end
33
34 % determine random stream and its state
35 stream = RandStream.getDefaultStream;
36 state = stream.State;
37
38 [opinion, consensusTime, t, intermediateOpinion, employedAgent] = ...
39  simulate(A, m, maxT, dt, n0, mu0, u0, consensusFraction, epsilon, ...
40  employedAgent, getIntermediateOpinion, breakOnConsensus, action);
41
42 for i=1:1024
43     filename = sprintf('simulationResult/%s_n0=%d_mu=%s_u=%s_[%d%s].mat', ...
44      graphName, n0, muStr, uStr, i, worker);
45     if exist(filename, 'file') ~= 2
46         save(filename, 'opinion', 'consensusTime', 't', ...
47          'intermediateOpinion', 'employedAgent', 'A', 'm', 'maxT', 'dt', ...
48          'mu0', 'u0', 'consensusFraction', 'epsilon', 'employedAgent', ...
49          'getIntermediateOpinion', 'breakOnConsensus', 'action', ...
50          'graphName', 'stream', 'state');
```

```
51          return
52      end
53  end
54
55  fprintf('Error writing parameters and output to file\n');
56
57  end % simulate_and_store(...)
```

simulation/select_greedy.m

```
1   function employedAgents = select_greedy(A, n0)
2   % Select greedily (as described in the report) the n0 agents corresponding to
3   % the nodes with the largest degrees.
4   %
5   % INPUT
6   % A: [n n]: adjacency representation of a graph describing the connections
7   %  between agents, i.e. A(i, j) is 1 iff there is an edge between node i and j.
8   %  A is assumed to be symmetric.
9   % n0: [1]: number of employed agents to be selected, n0 <= n
10  %
11  % OUTPUT
12  % employedAgents: [1 n0]: indices of the selected employed agents
13
14  [~, J] = sort(sum(A));
15  employedAgents = J(end-n0+1:end);
16
17  end % select_greedy(...)
```

7.4.2 Generation

generation/random_graph.m

```
1   function A = random_graph(n, p)
2   % Generates an undirected random graph (without self-loops) of size n (as
3   % described in the Erdoes-Renyi model)
4   %
5   % INPUT
6   % n: [1]: number of nodes
7   % p: [1]: probability that node i and node j, i != j, are connected by an edge
8   %
9   % OUTPUT
10  % A: [n n] sparse symmetric adjacency matrix representing the generated graph
11
12  % Note: A generation based on sprandsym(n, p) failed (for some values of p
13  % sprandsym was far off from the expected number of n*n*p non-zeros), therefore
14  % this longish implementation instead of just doing the following:
15  %
16  %     B = sprandsym(n, p);
17  %     A = (B-diag(diag(B))~=0);
18  %
19
20  % Idea: first generate the number of non-zero values in every row for a general
```

```
21 % 0−1−adjacency matrix. For every row this number is distributed binomially with
22 % parameters n and p.
23 %
24 % The following lines calculate "rowsize = binoinv(rand(1, n), n, p)", just in a
25 % faster way for large values of n.
26
27 % generate a vector of n values chosen u.a.r. from (0,1)
28 v = rand(1, n);
29 % Sort them and calculate the binomial cumulative distribution function with
30 % parameters n and p at values 0 to n. Afterwards match the sorted random
31 % 0−1−values to those cdf−values, i.e. associate a binomial distributed value
32 % with each value in r. Each value in v also corresponds to a value in r:
33 % permute the values in rowSize s.t. they correspond to the order given in v.
34 [r index] = sort(v); % i.e. v(index) == r holds
35 rowSize = zeros(1, n);
36 j = 0;
37 binoCDF = cumsum(binopdf(0:n, n, p));
38 for i = 1:n
39     while j<n && binoCDF(j+1)<r(i)
40         j = j + 1;
41     end
42     rowSize(i) = j;
43 end
44 rowSize(index) = rowSize;
45
46 % for every row choose the non−zero entries in it
47 nNZ = sum(rowSize);
48 I = zeros(1, nNZ);
49 J = zeros(1, nNZ);
50 j = 1;
51 for i = 1:n
52     I(j:j+rowSize(i)−1) = i;
53     J(j:j+rowSize(i)−1) = randsample(n, rowSize(i));
54     j = j + rowSize(i);
55 end
56
57 % restrict I and J to indices that correspond to entries above the main diagonal
58 % and finally construct a symmetric sparse matrix using I and J
59 upperTriu = find(I<J);
60 I = I(upperTriu);
61 J = J(upperTriu);
62 A = sparse([I;J], [J;I], ones(1, 2*size(I, 2)), n, n);
63
64 end % random_graph(...)
```

generation/scale_free.m

```
1 function A = scale_free(n, m0, m)
2 % Use the Barabasi−Albert model to generate a scale free graph of size n (as
3 % described in Albert−Laszlo Barabasi & Reka Albert: "Emergence of scaling
4 % in random networks")
5 %
6 % INPUT
7 % n: [1]: number of nodes
8 % m0: [1]: number of initially placed nodes
9 % m: [1]: number of nodes a new added node is connected to, 1 <= m < m0
```

```
10 %
11 % OUPUT
12 % A: [n n] sparse symmetric adjacency matrix representing the generated graph
13
14 % Start with a graph of size m0 and add edges to this graph. Each of these m0
15 % nodes is connected to at least m nodes.
16 B = zeros(m0, m0);
17 for i = 1:m0
18     neighbors = randsample(m0-1, m);
19     neighbors = neighbors + (neighbors>=i);
20     B(i,neighbors) = 1;
21     B(neighbors,i) = 1;
22 end
23
24 % Create a vector of edges added so far, i.e. nodes edge(2*i) and edge(2*i-1),
25 % 1 <= i <= nEdges, are connected by an edge.
26 [rows, columns] = find(triu(B));
27 nEdges = size(rows, 1);
28 edges = reshape([rows';columns'], 2*nEdges, 1);
29 edges = [edges; zeros(2*(n-m0)*m,1)];
30
31 % Add nodes m0+1:n, one at a time. Each node is connected to m existing nodes,
32 % where each of the existing nodes is chosen with a probability that is
33 % proportional to the number of nodes it is already connected to.
34 used = zeros(n, 1); % is a node already used in a timestep?
35 for i = m0+1:n
36     neighbors = zeros(1, m);
37     for j=1:m
38         k = edges(randi(2*nEdges));
39         while used(k)
40             k = edges(randi(2*nEdges));
41         end
42         used(k) = 1;
43         neighbors(j) = k;
44     end
45     used(neighbors) = 0;
46     edges(2*nEdges+1:2*nEdges+2*m) = reshape([repmat(i, 1, m); neighbors], ...
47      1, 2*m);
48     nEdges = nEdges+m;
49 end
50
51 % finally construct a symmetric adjacency matrix using the vector of edges
52 edges = edges(1:2*nEdges);
53 first = edges(1:2:end);
54 second = edges(2:2:end);
55 A = sparse([first;second], [second;first], ones(2*nEdges, 1), n, n);
56
57 end % scale_free(...)
```

generation/small_world.m

```
1 function A = small_world(n, k, beta)
2 % Generate a small world graph using the "Watts and Strogatz model" as
3 % described in Watts, D.J.; Strogatz, S.H.: "Collective dynamics of
4 % 'small-world' networks."
5 % A graph with n*k/2 edges is constructed, i.e. the nodal degree is n*k for
```

```
6  % every node.
7  %
8  % INPUT
9  % n: [1]: number of nodes of the graph to be generated
10 % k: [1]: mean degree (assumed to be an even integer)
11 % beta: [1]: rewiring probability
12 %
13 % OUPUT
14 % A: [n n] sparse symmetric adjacency matrix representing the generated graph
15
16 % Construct a regular lattice: a graph with n nodes, each connected to k
17 % neighbors, k/2 on each side.
18 kHalf = k/2;
19 rows = reshape(repmat([1:n]', 1, k), n*k, 1);
20 columns = rows+reshape(repmat([[1:kHalf] [n-kHalf:n-1]], n, 1), n*k, 1);
21 columns = mod(columns-1, n) + 1;
22 B = sparse(rows, columns, ones(n*k, 1));
23 A = sparse([], [], [], n, n);
24
25 % With probability beta rewire an edge avoiding loops and link duplication.
26 % Until step i, only the columns 1:i are generated making implicit use of A's
27 % symmetry.
28 for i = [1:n]
29     % The i-th column is stored full for fast access inside the following loop.
30     col= [full(A(i, 1:i-1))'; full(B(i:end, i))];
31     for j = i+find(col(i+1:end))'
32         if (rand()<beta)
33             col(j)=0;
34             k = randi(n);
35             while k==i || col(k)==1
36                 k = randi(n);
37             end
38             col(k) = 1;
39         end
40     end
41     A(:,i) = col;
42 end
43
44 % A is not yet symmetric: to speed things up, an edge connecting i and j, i < j
45 % implies A(i,j)==1, A(i,j) might be zero.
46 T = triu(A);
47 A = T+T';
48
49 end % small_world(...)
```

7.4.3 Statistics

statistics/print_statistics.m

```
1 function print_statistics(A)
2 % Print statics for a undirected, loop-free graph.
3 %
4 % INPUT
5 % A: [n n]: adjacency matrix
```

```
6
7  n = size(A, 1);
8  fprintf('number of nodes = %d\n', n);
9
10 m = nnz(A);
11 assert(mod(m, 2) == 0);
12 fprintf('number of (undirected) edges = %d\n', m/2);
13
14 k = m/n;
15 fprintf('average node degree = %.4f\n', k);
16
17 maxDeg = max(full(sum(A)));
18 fprintf('maximum node degree = %d\n', maxDeg);
19
20 avgPathLength = average_path_length(A);
21 fprintf('average path length = %.4f\n', avgPathLength);
22
23 clusterCoeff = global_clustering_coefficient(A);
24 fprintf('global clustering coefficient = %.4f\n', clusterCoeff);
25
26 end % print_statistics(...)
```

statistics/average_path_length.m

```
1  function lG = average_path_length(A)
2  % Calculate the average path length in a graph.
3  % (see http://en.wikipedia.org/wiki/Average_path_length for a definition)
4  %
5  % INPUT
6  % A: [n n]: adjacency matrix
7  %
8  % OUTPUT
9  % lG: [1]: average path length
10 %
11 % This function makes use of the Parallel Computing Toolbox, i.e. the outer loop
12 % can be divided among several workers.
13
14 n = size(A, 1);
15 lGs = zeros(n, 1);
16 parfor i = 1:n
17     active = i;
18     todo = ones(n,1);
19     todo(i) = 0;
20     dist = 0;
21     total = 0;
22     while ~isempty(active)
23         dist = dist + 1;
24         active = find(sum(A(:,active), 2) .* todo);
25         todo(active) = 0;
26         total = total + size(active, 1) * dist;
27     end
28     lGs(i) = total;
29 end
30 lG = sum(lGs) / (n * (n - 1));
31
32 end % average_path_length(...)
```

statistics/global_clustering_coefficient.m

```
1  function C = global_clustering_coefficient(A)
2  % Calculate the global clustering coefficient of a graph.
3  % (see http://en.wikipedia.org/wiki/Clustering_coefficient for a definition)
4  %
5  % INPUT
6  % A: [n n]: adjacency matrix
7  %
8  % OUTPUT
9  % C: [1]: global clustering coefficient
10 %
11 % This function makes use of the Parallel Computing Toolbox, i.e. the outer loop
12 % can be divided among several workers.
13
14 n = size(A, 1);
15 Cs = zeros(n, 1);
16 parfor i = 1:n
17     Cs(i) = local_clustering_coefficient(A, i);
18 end
19 C = sum(Cs) / n;
20
21 end % global_clustering_coefficient(...)
```

statistics/local_clustering_coefficient.m

```
1  function Ci = local_clustering_coefficient(A, i)
2  % Calculate the local clustering coefficient of a node in a graph.
3  % (see http://en.wikipedia.org/wiki/Clustering_coefficient for a definition)
4  %
5  % INPUT
6  % A: [n n]: adjacency matrix
7  % i: [1]: index of the node the local clustering coefficient is calculated for
8  %
9  % OUTPUT
10 % Ci: [1]: local clustering coefficient
11
12 neighbor = find(A(:, i));
13 m = size(neighbor, 1);
14 Ci = 0;
15 if m > 1
16     for j = neighbor'
17         Ci = Ci + full(sum(A(neighbor, j)));
18     end
19     Ci = Ci / (m * (m - 1));
20 end
21
22 end % local_clustering_coefficient(...)
```

7.4.4 Utility

utility/get_graph.m

```
function [A, name, stream, state] = get_graph(graphID, graphSize, ...
 generateNew, verbose)
% Function for loading, generating and storing graphs. If the requested graph
% does not exist, the function generates a new one and stores it to a mat-file.
% For proper operation, the folders 'generatedGraphs' and 'facebook100' have to
% be within the Matlab path.
%
% INPUT
% graphID: [1]: identifier for the graph type which should be loaded/generated:
%  1    <--> Random graph
%  2    <--> Scale free graph
%  3    <--> Small world graph
%  4-6 <--> Real world graph
% graphSize: [1]: determines size of graph:
%  0 <-> small graph, i.e. ~1000 nodes
%  1 <-> large graph, i.e. ~35000 nodes
% generateNew: [1]: flag to generate a new graph, i.e. the graph is newly
%  generated even if it has been stored before
% verbose: [1]: enable showing what is going on
%
% OUTPUT
% A: [n,n] (sparse): adjacency matrix of the loaded/generated graph. If the
%  graph is not connected, its largest component is returned.
% name: string: describing the returned graph
% stream: [1 RandStream]: the random stream used for generating the graph (as
%  returned by RandStream.getDefaultStream).
% state: the state of stream before the graph is generated

% Cell array for graph filenames
generatedGraphsFilenames = {{'Random_small', 'ScaleFree_small', ...
 'SmallWorld_small'}; {'Random_large', 'ScaleFree_large', 'SmallWorld_large'}};
% Cell array for real world graph filenames
realWorldGraphsFilenames = {{'Simmons81', 'Reed98', 'Caltech36'}; ...
 {'Penn94', 'UF21', 'Texas84'}};

% Parameters for graph generation. The number of nodes n, edges m and the
% average nodal degree k have been chosen such that they best match the
% corresponding characteristics of the real world graphs.
n = [1000 35000];
m = [40000 2900000];
k = [40 76];
k2 = ceil(k/2);

% Cell array with functions for graph generation.
generationFunctions = {{@(n) random_graph(n, m(1)/((n-1)*(n-1))), ...
 @(n) scale_free(n, k2(1)+1, k2(1)), @(n) small_world(n, k(1),0.25)} ...
 {@(n) random_graph(n, m(2)/((n-1)*(n-1))), @(n) scale_free(n, k2(2)+1, ...
 k2(2)), @(n) small_world(n, k(2), 0.25)}};

% determine random stream and its state
stream = RandStream.getDefaultStream;
state = stream.State;
```

```
53
54 name = '';
55 A = [];
56 if graphID <= 3
57     % Determine filename with cell array and check if file exists.
58     name = generatedGraphsFilenames{graphSize+1}{graphID};
59     filename = [name '.mat'];
60     if ~generateNew && exist(filename, 'file') == 2
61         if verbose
62             fprintf('Loading graph from: %s \n', filename);
63         end
64         load(filename, 'A', 'stream', 'state');
65     else % New graph to be generated or file does not exist.
66         if graphSize>1
67             error 'Invalid value for graphSize';
68         end;
69         if verbose
70             fprintf(['New graph with %d nodes is generated and saved to ' ...
71              '%s\n'], n(graphSize+1), filename);
72         end
73         A = generationFunctions{graphSize+1}{graphID}(n(graphSize+1));
74         oldSize = size(A, 1);
75         A = largest_connected_component(A);
76         if verbose && size(A, 1) ~= oldSize
77             fprintf(['dropped %d nodes from graph of size %d resulting in ' ...
78              'a graph of size %d\n'], oldSize-size(A, 1), oldSize, size(A, 1));
79         end
80         save(['generatedGraphs/' filename], 'A', 'stream', 'state');
81     end
82 elseif graphID <= 6 % real world graph
83     name = realWorldGraphsFilenames{graphSize+1}{graphID-3};
84     filename = [name '.mat'];
85     if verbose
86         fprintf('Loading real world graph: %s\n', filename);
87     end
88     load(filename, 'A');
89     oldSize = size(A, 1);
90     A = largest_connected_component(A);
91     if verbose && size(A, 1) ~= oldSize
92         fprintf(['dropped %d nodes from graph of size %d resulting in a ' ...
93          'graph of size %d\n'], oldSize-size(A, 1), oldSize, size(A, 1));
94     end
95 else
96     error 'Wrong graphID';
97 end
98
99 end % get_graph(...)
```

utility/largest_connected_component.m

```
1 function B = largest_connected_component(A)
2 % Returns the adjacency matrix of the largest connected component of an
3 % undirected graph.
4 %
5 % INPUT
6 % A: [n,n]: symmetric adjacency matrix with entries 0 resp. 1
```

```
7  %
8  % OUTPUT
9  % B: [m,m]: Adjacency matrix of the largest connected component of the graph
10 %  represented by the matrix A.
11
12 n = size(A, 1);
13 % vector to record visited nodes: node i was visited iff visited(i)==1
14 visited = zeros(n, 1);
15 todo = ones(n,1);
16 indexLargestComponent = 0;
17 sizeLargestComponent = 0;
18 for i = 1:n
19     % if node i wasn't visited yet, start a new component search at node i
20     if ~visited(i)
21         % vector with indices of currently active nodes
22         active = i;
23         todo(i) = 0;
24         count = 1;
25         visited(i) = i;
26         while ~isempty(active)
27             % Set active nodes to all not visited nodes reachable from ...
28             % currently active nodes.
29             active = find(sum(A(:, active), 2) .* todo);
30             count = count + length(active);
31             todo(active) = 0;
32             visited(active) = i;
33         end
34         if count>sizeLargestComponent
35             sizeLargestComponent = count;
36             indexLargestComponent = i;
37         end
38     end
39 end
40
41 % extract indices of nodes belonging to the largest connected component
42 largestComponent = find(visited==indexLargestComponent);
43 B = A(largestComponent, largestComponent);
44
45 end % largest_connected_component(...)
```